PsychoPy - Psychology software for Python

Release 2025.1.0

Open Science Tools Ltd

Apr 29, 2025

CONTENTS

A pdf copy of the current documentation is available here:

PsychoPy Manual

Contents:

CHAPTER

ONE

ABOUT

1.1 Citing

If you use this software, please cite one of the publications that describe it. For most people **the 2019 paper is probably the most relevant** (the papers from 2009, 2007 did not mention Builder at all, for instance).

- Peirce, J. W., Gray, J. R., Simpson, S., MacAskill, M. R., Höchenberger, R., Sogo, H., Kastman, E., Lindeløv, J. (2019). PsychoPy2: experiments in behavior made easy. *Behavior Research Methods.* 10.3758/s13428-018-01193-y
- Peirce, J. W., Hirst, R. J. & MacAskill, M. R. (2022). Building Experiments in PsychoPy. 2nd Edn London: Sage.
- Peirce J. W. (2009). Generating stimuli for neuroscience using PsychoPy. *Frontiers in Neuroinformatics,* **2** (10), 1-8. doi:10.3389/neuro.11.010.2008
- Peirce, J. W. (2007). PsychoPy - Psychophysics software in Python. *Journal of Neuroscience Methods,* **162** (1-2):8-13 doi:10.1016/j.jneumeth.2006.11.017

Citing these papers gives the reviewer/reader of your study information about how the system works and it attributes some credit for its original creation. Academic assessment (whether for promotion or even getting appointed to a job in the first place) prioritises publications over making useful tools for others. Citations provide a way for the developers to justify their continued involvement in the development of the package.

1.2 License for use

is licensed under a GPL3 license which means, essentially, that:

- you can use it (and adapt it) for free in your work, and you can even release those versions
- but you must include the original license
- AND you must also make your release open source using the same license

What that means is you're free to use PsychoPy's goodwill in being open source, you are required to pass on that goodwill!

CHAPTER

TWO

GENERAL ISSUES

These are issues that users should be aware of, whether they are using Builder or Coder views.

2.1 Monitor Center

🖥

provides a simple and intuitive way for you to calibrate your monitor and provide other information about it and then import that information into your experiment.

Information is inserted in the Monitor Center (Tools menu), which allows you to store information about multiple monitors and keep track of multiple calibrations for the same monitor.

For experiments written in the Builder view, you can then import this information by simply specifying the name of the monitor that you wish to use in the *Experiment settings* dialog. For experiments created as scripts you can retrieve the information when creating the `Window` by simply naming the monitor that you created in Monitor Center. e.g.:

```
from psychopy import visual
win = visual.Window([1024,768], mon='SonyG500')
```

Of course, the name of the monitor in the script needs to match perfectly the name given in the Monitor Center.

2.1.1 Real world units

One of the particular features of is that you can specify the size and location of stimuli in units that are independent of your particular setup, such as degrees of visual angle (see *Units for the window and stimuli*). In order for this to be possible you need to inform of some characteristics of your monitor. Your choice of units determines the information you need to provide:

Units	Requires
'norm' (normalised to width/height)	n/a
'pix' (pixels)	Screen width in pixels
'cm' (centimeters on the screen)	Screen width in pixels and screen width in cm
'deg' (degrees of visual angle)	Screen width (pixels), screen width (cm) and distance (cm)

2.1.2 Calibrating your monitor

can also store and use information about the gamma correction required for your monitor. If you have a Spectrascan PR650, PR655/PR670, Minolta LS100/LS110 or a CRS ColorCAL you can perform an automated calibration in which will measure the necessary gamma value to be applied to your monitor. Alternatively this can be added manually into the grid to the right of the Monitor Center. To run a calibration, connect the photometer via the serial port and, immediately after turning it on press the *Get Photometer* button in the Monitor Center.

Note that, if you don't have a photometer to hand then there is a method for determining the necessary gamma value psychophysically included in (see gammaMotionNull and gammaMotionAnalysis in the coder demos menu, under "experiment control").

The two additional tables in the Calibration box of the Monitor Center provide conversion from *DKL* and *LMS* colour spaces to *RGB*.

2.2 Units for the window and stimuli

One of the key advantages of over many other experiment-building software packages is that stimuli can be described in a wide variety of real-world, device-independent units. In most other systems you provide the stimuli at a fixed size and location in pixels, or percentage of the screen, and then have to calculate how many cm or degrees of visual angle that was.

In , after providing information about your monitor, via the *Monitor Center*, you can simply specify your stimulus in the unit of your choice and allow to calculate the appropriate pixel size for you.

Your choice of unit depends on the circumstances. For conducting demos, the two normalised units ('norm' and 'height') are often handy because the stimulus scales naturally with the window size. For running an experiment it's usually best to use something like 'cm' or 'deg' so that the stimulus is a fixed size irrespective of the monitor/window.

For all units, the centre of the screen is represented by coordinates (0,0), negative values mean down/left, positive values mean up/right.

For help understanding spatial units visually, try the builder demo "spatialUnits" under "Understanding PsychoPy" (version 2021.2).

2.2.1 Units for online experiments

If you are running a study online the easiest units to use will be those that require no monitor info. It is likely that your experiment will be run on a wide variety of devices all with differing screen resolutions. Furthermore it is going to be more difficult for you to control factors like viewing distance. Because of this it makes it difficult to use units like *deg* or *cm* - because we need to know both the participants viewing distance and the number of pixels that make up a cm on that participants screen. The easiest solution here is to use *Height units*, this means that the size of stimuli will be scaled relative to the height of that participants screen - which usually means it is possible to run studies even on smartphones!

Note

If using height units on a tablet/touchscreen device,currently 1 unit height corresponds to the height of the screen when the device is held in landscape.

Degrees of visual angle are not currently supported for online use, but you can estimate pixels per cm using a screen scaling method (this demo was shared by Wakefield Morys Carter 2021) and then use pixel units to present stimuli in cm see Li et al (2020) for more details. If you want to store the window size of your participants device in an online study, you can add a code component and use `thisExp.addData('windowSize', win.size)`.

2.2.2 Height units

With 'height' units everything is specified relative to the height of the window (note the window, not the screen). As a result, the dimensions of a screen with standard 4:3 aspect ratio will range (-0.6667,-0.5) in the bottom left to (+0.6667,+0.5) in the top right. For a standard widescreen (16:10 aspect ratio) the bottom left of the screen is (-0.8,-0.5) and top-right is (+0.8,+0.5). This type of unit can be useful in that it scales with window size, unlike *Degrees of* *visual angle* or *Centimeters on screen*, but stimuli remain square, unlike *Normalised units* units. Obviously it has the disadvantage that the location of the right and left edges of the screen have to be determined from a knowledge of the screen dimensions. (These can be determined at any point by the *Window.size* attribute.)

Spatial frequency: cycles **per stimulus** (so will scale with the size of the stimulus).

:darkgreen:`Requires :No monitor information`

2.2.3 Normalised units

In normalised ('norm') units the window ranges in both x and y from -1 to +1. That is, the top right of the window has coordinates (1,1), the bottom left is (-1,-1). Note that, in this scheme, setting the height of the stimulus to be 1.0, will make it half the height of the window, not the full height (because the window has a total height of 1:-1 = 2!). Also note that specifying the width and height to be equal will not result in a square stimulus if your window is not square - the image will have the same aspect ratio as your window. e.g. on a 1024x768 window the size=(0.75,1) will be square.

Spatial frequency: cycles **per stimulus** (so will scale with the size of the stimulus).

:darkgreen:`Requires : No monitor information`

2.2.4 Centimeters on screen

Set the size and location of the stimulus in centimeters on the screen.

Spatial frequency: cycles per cm

:darkorange:`Requires : information about the screen width in cm and size in pixels`

Assumes : pixels are square. Can be verified by drawing a stimulus with matching width and height and verifying that it is in fact square. For a *CRT* this can be controlled by setting the size of the viewable screen (settings on the monitor itself).

2.2.5 Degrees of visual angle

Use degrees of visual angle to set the size and location of the stimulus. This is, of course, dependent on the distance that the participant sits from the screen as well as the screen itself, so make sure that this is controlled, and remember to change the setting in *Monitor Center* if the viewing distance changes.

Spatial frequency: cycles per degree

:darkorange:`Requires : information about the screen width in cm and pixels and the viewing distance in cm`

There are actually three variants: 'deg', 'degFlat', and 'degFlatPos'

- **'deg'** : Most people using degrees of visual angle choose to make the assumption that a degree of visual angle spans the same number of pixels at all parts of the screen. This isn't actually true for standard flat screens - a degree of visual angle at the edge of the screen spans more pixels because it is further from the eye. For moderate eccentricities the error is small (a 0.2% error in size calculation at 3 deg eccentricity) but grows as stimuli are placed further from the centre of the screen (a 2% error at 10 deg). For most studies this form of calculation is preferred, as it does not result in a warped appearance of visual stimuli, but if you need greater precision at far eccentricities then choose one of the alternatives below.
- **'degFlatPos'** : This accounts for flat screens in calculating position coordinates of visual stimuli but leaves size and spatial frequency uncorrected. This means that an evenly spaced grid of visual stimuli will appear warped in position but will
- **'degFlat'**: This corrects the calculations of degrees for flatness of the screen for each vertex of your stimuli. Square stimuli in the periphery will, therefore, become more spaced apart but they will also get larger and rhomboid in the pixels that they occupy.

2.2.6 Pixels on screen

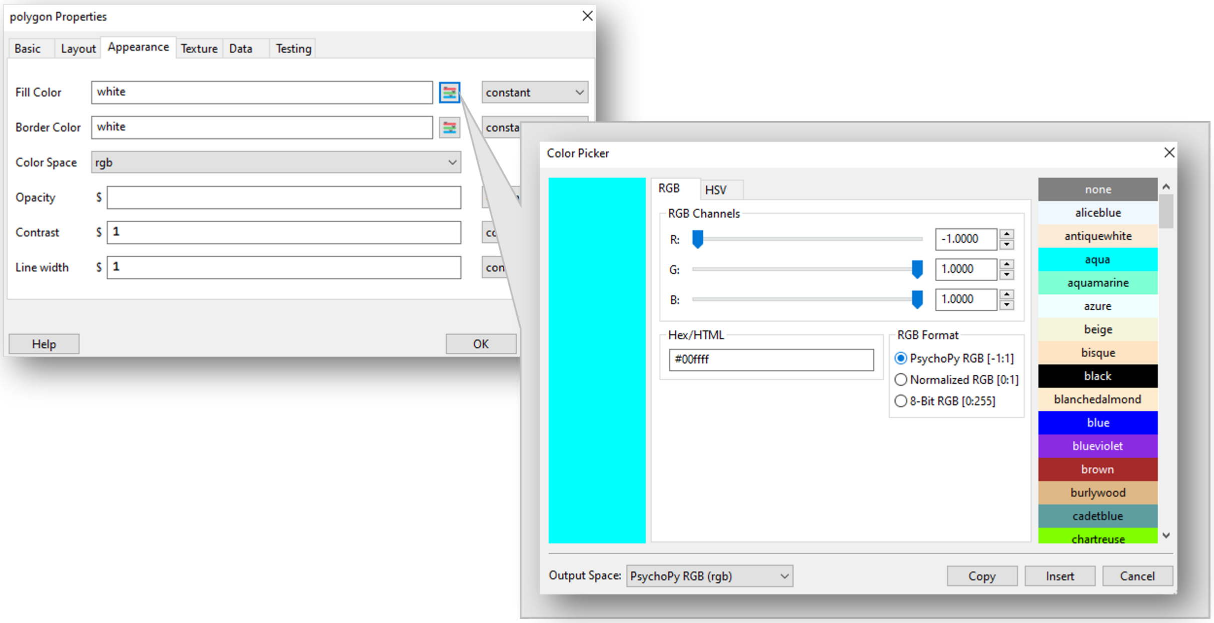
You can also specify the size and location of your stimulus in pixels. Obviously this has the disadvantage that sizes are specific to your monitor (because all monitors differ in pixel size).

Spatial frequency: `` `cycles per pixel` `` (this catches people out but is used to be in keeping with the other units. If using pixels as your units you probably want a spatial frequency in the range 0.2-0.001 (i.e. from 1 cycle every 5 pixels to one every 100 pixels).

:darkorange:`Requires : information about the size of the screen (not window) in pixels, although this can often be deduce from the operating system if it has been set correctly there.`

Assumes: nothing

2.3 Color spaces



polygon Properties

Fill Color: white — constant

Border Color: white — consta

Color Space: rgb

Opacity: $

Contrast: $ 1

Line width: $ 1

Color Picker

RGB Channels

R: -1.0000

G: 1.0000

B: 1.0000

Hex/HTML: #00ffff

RGB Format

● PsychoPy RGB [-1:1]

○ Normalized RGB [0:1]

○ 8-Bit RGB [0:255]

Output Space: PsychoPy RGB (rgb)

You can explore colors in PsychoPy Builder through accessing the color picker from any parameter that takes a color value.

The color of stimuli can be specified when creating a stimulus and when using setColor() in a variety of ways. From Builder view you can also use the color picker to pick the color you want and explore what value that color would correspond to in a variety of spaces. There are three basic color spaces that can use, RGB, DKL and LMS but colors can also be specified by a name (e.g. 'DarkSalmon') or by a hexadecimal string (e.g. '#00FF00').

examples:

```
stim = visual.GratingStim(win, color=[1,-1,-1], colorSpace='rgb') #will be red
stim.setColor('Firebrick')#one of the web/X11 color names
stim.setColor('#FFFAF0')#an off-white
stim.setColor([0,90,1], colorSpace='dkl')#modulate along S-cone axis in isoluminant plane
stim.setColor([1,0,0], colorSpace='lms')#modulate only on the L cone
stim.setColor([1,1,1], colorSpace='rgb')#all guns to max
stim.setColor([1,0,0])#this is ambiguous - you need to specify a color space
```

2.3.1 Colors by name

Any of the web/X11 color names can be used to specify a color. These are then converted into RGB space by .

These are not case sensitive, but should not include any spaces.

2.3.2 Colors by hex value

This is really just another way of specifying the r,g,b values of a color, where each gun's value is given by two hexadecimal characters. For some examples see this chart. To use these in they should be formatted as a string, beginning with # and with no spaces. (NB on a British Mac keyboard the # key is hidden - you need to press Alt-3)

2.3.3 RGB color space

This is the simplest color space, in which colors are represented by a triplet of values that specify the red green and blue intensities. These three values each range between -1 and 1.

Examples:

- `[1,1,1]` is white
- `[0,0,0]` is grey
- `[-1,-1,-1]` is black
- `[1.0,-1,-1]` is red
- `[1.0,0.6,0.6]` is pink

The reason that these colors are expressed ranging between 1 and -1 (rather than 0:1 or 0:255) is that many experiments, particularly in visual science where has its roots, express colors as deviations from a grey screen. Under that scheme a value of -1 is the maximum decrement from grey and +1 is the maximum increment above grey.

You can still specify colors in RGB from 0:1 or 0:255, but you will need to let know that this is what you're doing. To do this, set the color space to be *rgb1* for 0:1 or *rgb255* for 0:255 - if the color space is just *rgb*, then values will be from -1:1

Note that will use your monitor calibration to linearize this for each gun. E.g., 0 will be halfway between the minimum luminance and maximum luminance for each gun, if your monitor gammaGrid is set correctly.

2.3.4 HSV color space

Another way to specify colors is in terms of their Hue, Saturation and 'Value' (HSV). For a description of the color space see the Wikipedia HSV entry. The Hue in this case is specified in degrees, the saturation ranging 0:1 and the 'value' also ranging 0:1.

Examples:

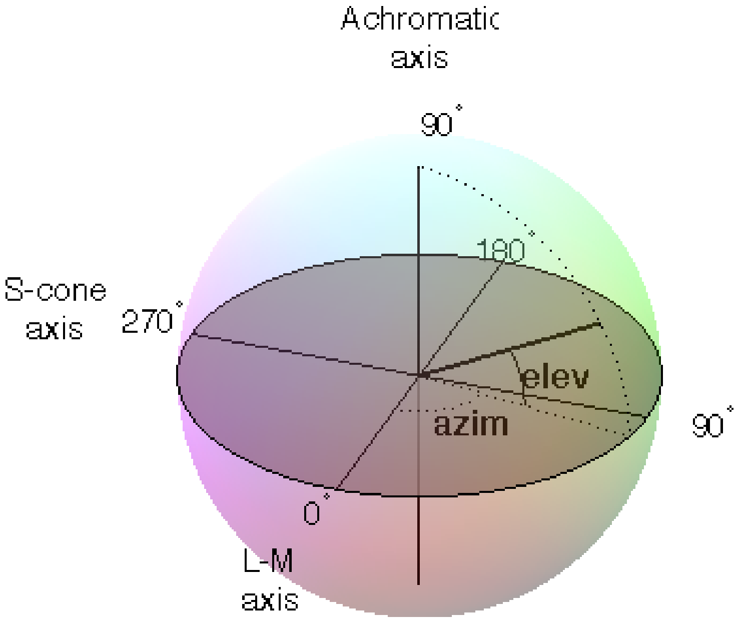
- `[0,1,1]` is red
- `[0,0.5,1]` is pink
- `[90,1,1]` is cyan
- `[anything, 0, 1]` is white
- `[anything, 0, 0.5]` is grey
- `[anything, anything,0]` is black

Note that colors specified in this space (like in RGB space) are not going to be the same another monitor; they are device-specific. They simply specify the intensity of the 3 primaries of your monitor, but these differ between monitors. As with the RGB space gamma correction is automatically applied if available.

2.3.5 DKL color space

To use DKL color space the monitor should be calibrated with an appropriate spectrophotometer, such as a PR650.

In the Derrington, Krauskopf and Lennie[1] color space (based on the Macleod and Boynton[2] chromaticity diagram) colors are represented in a 3-dimensional space using spherical coordinates that specify the *elevation* from the isoluminant plane, the *azimuth* (the hue) and the contrast (as a fraction of the maximal modulations along the cardinal axes of the space).



In these values are specified in units of degrees for elevation and azimuth and as a float (ranging -1:1) for the contrast.

Note that not all colors that can be specified in DKL color space can be reproduced on a monitor. You can see here a movie plotting colors in DKL space (showing *cartesian* coordinates, not spherical coordinates) to show the gamut of colors available on an example monitor.

Examples:

- `[90,0,1]` is white (maximum elevation aligns the color with the luminance axis)
- `[0,0,1]` is an isoluminant stimulus, with azimuth 0 (S-axis)
- `[0,45,1]` is an isoluminant stimulus,with an oblique azimuth

[1] Derrington, A.M., Krauskopf, J., & Lennie, P. (1984). Chromatic Mechanisms in Lateral Geniculate Nucleus of Macaque. Journal of Physiology, 357, 241-265.

[2] MacLeod, D. I. A. & Boynton, R. M. (1979). Chromaticity diagram showing cone excitation by stimuli of equal luminance. Journal of the Optical Society of America, 69(8), 1183-1186.

2.3.6 LMS color space

To use LMS color space the monitor should be calibrated with an appropriate spectrophotometer, such as a PR650.

In this color space you can specify the relative strength of stimulation desired for each cone independently, each with a value from -1:1. This is particularly useful for experiments that need to generate cone isolating stimuli (for which modulation is only affecting a single cone type).

2.4 Preferences

The Preferences dialog allows to adjust general settings for different parts of . The preferences settings are saved in the configuration file *userPrefs.cfg*. The labels in brackets for the different options below represent the abbreviations used in the *userPrefs.cfg* file.

In rare cases, you might want to adjust the preferences on a per-experiment basis. See the API reference for the *Preferences class here*.

2.4.1 General settings (General)

window type (winType) :
can use one of two 'backends' for creating windows and drawing; pygame, pyglet and glfw. Here you can set the default backend to be used.

units (units) :
Default units for windows and visual stimuli ('deg', 'norm', 'cm', 'pix'). See *Units for the window and stimuli*. Can be overridden by individual experiments.

full-screen (fullscr) :
Should windows be created full screen by default? Can be overridden by individual experiments.

allow GUI (allowGUI) :
When the window is created, should the frame of the window and the mouse pointer be visible. If set to False then both will be hidden.

paths (paths) :
Paths for additional Python packages can be specified. See more *information on paths here*.

flac audio compression (flac) :
Set flac audio compression.

parallel ports (parallelPorts) :
This list determines the addresses available in the drop-down menu for the /builder/components/parallelout.

2.4.2 Application settings (App)

These settings are common to all components of the application (Coder and Builder etc)

show start-up tips (showStartupTips) :
Display tips when starting .

large icons (largeIcons) :
Do you want large icons (on some versions of wx on macOS this has no effect)?

default view (defaultView) :
Determines which view(s) open when the app starts up. Default is 'last', which fetches the same views as were open when last closed.

reset preferences (resetPrefs) :
Reset preferences to defaults on next restart of .

auto-save prefs (autoSavePrefs) :
Save any unsaved preferences before closing the window.

debug mode (debugMode) :
Enable features for debugging itself, including unit-tests.

locale (locale) :
Language to use in menus etc.; not all translations are available. Select a value, then restart the app. Think about *adding translations for your language*.

2.4.3 Builder settings (Builder)

reload previous exp (reloadPrevExp) :
Select whether to automatically reload a previously opened experiment at start-up.

uncluttered namespace (unclutteredNamespace) :
If this option is selected, the scripts will use more complex code, but the advantage is that there is less of a chance that name conflicts will arise.

components folders (componentsFolders) :
A list of folder path names that can hold additional custom components for the Builder view; expects a comma-separated list.

hidden components (hiddenComponents) :
A list of components to hide (e.g., because you never use them)

unpacked demos dir (unpackedDemosDir) :
Location of Builder demos on this computer (after unpacking).

saved data folder (savedDataFolder) :
Name of the folder where subject data should be saved (relative to the script location).

Flow at top (topFlow) :
If selected, the “Flow” section will be shown topmost and the “Components” section will be on the left. Restart to activate this option.

always show readme (alwaysShowReadme) :
If selected, always shows the Readme file if you open an experiment. The Readme file needs to be located in the same folder as the experiment file.

max favorites (maxFavorites) :
Upper limit on how many components can be in the Favorites menu of the Components panel.

2.4.4 Coder settings (Coder)

code font (codeFont) :
A list of font names to be used for code display. The first found on the system will be used.

comment font (commentFont) :
A list of font names to be used for comments sections. The first found on the system will be used

output font (outputFont) :
A list of font names to be used in the output panel. The first found on the system will be used.

code font size (codeFontSize) :
An integer between 6 and 24 that specifies the font size for code display in points.

output font size (outputFontSize) :
An integer between 6 and 24 that specifies the font size for output display in points.

show source asst (showSourceAsst) :
Do you want to show the source assistant panel (to the right of the Coder view)? On Windows this provides help about the current function if it can be found. On macOS the source assistant is of limited use and is disabled by default.

show output (showOutput) :
Show the output panel in the Coder view. If shown all python output from the session will be output to this panel. Otherwise it will be directed to the original location (typically the terminal window that called application to open).

reload previous files (reloadPrevFiles) :
Should fetch the files that you previously had open when it launches?

preferred shell (preferredShell) :
Specify which shell should be used for the coder shell window.

newline convention (newlineConvention) :
Specify which character sequence should be used to encode newlines in code files: unix = n (line feed only), dos = rn (carriage return plus line feed).

2.4.5 Connection settings (Connections)

proxy (proxy) :
The proxy server used to connect to the internet if needed. Must be of the form http://111.222.333.444:5555

auto-proxy (autoProxy) :
should try to deduce the proxy automatically. If this is True and autoProxy is successful, then the above field should contain a valid proxy address.

allow usage stats (allowUsageStats) :
Allow to ping a website at when the application starts up. Please leave this set to True. The info sent is simply a string that gives the date, version and platform info. There is no cost to you: no data is sent that could identify you and will not be delayed in starting as a result. The aim is simple: if we can show that lots of people are using there is a greater chance of it being improved faster in the future.

check for updates (checkForUpdates) :
can (hopefully) automatically fetch and install updates. This will only work for minor updates and is still in a very experimental state (as of v1.51.00).

timeout (timeout) :
Maximum time in seconds to wait for a connection response.

2.4.6 Hardware settings

audioLib :
Select your choice of audio library with a list of names specifying the order they should be tried. We recommend *['PTB', 'sounddevice', 'pyo', 'pygame']* for lowest latency.

audioLatencyMode
[0, 1, 2, 3 (default), 4] Latency mode for PsychToolbox audio. See *PTB Audio Latency Modes*.

audioDriver
['portaudio'] Some of PsychoPy's audio engines provide the option not to use portaudio but go directly to another lib (e.g. to coreaudio) but some don't allow that.

audioDevice :
The name of the audio driver to use.

parallelPorts :
A list of parallel ports. The default is `['0x0378', '0x03BC']`.

qmixConfiguration :

The name of the Qmix pump configuration to use. The default is `'qmix_config'`.

2.4.7 Key bindings

There are many shortcut keys that you can use in . For instance did you realise that you can indent or outdent a block of code with Ctrl-[and Ctrl-] ?

2.5 Data outputs

There are a number of different forms of output that can generate, depending on the study and your preferred analysis software. Multiple file types can be output from a single experiment (e.g. *Excel data file* for a quick browse, *Log file* to check for error messages and *data file (.psydat)* for detailed analysis)

2.5.1 Log file

Log files are actually rather difficult to use for data analysis but provide a chronological record of everything that happened during your study. The level of content in them depends on you. See *Logging data* for further information.

2.5.2 data file (.psydat)

This is actually a *`TrialHandler`* or *`StairHandler`* object that has been saved to disk with the python cPickle module.

.psydat files can be useful for retrieving data that you forgot to explicitly tell to save. They can also be more directly used by experienced users with previous experience of python and, probably, matplotlib. The contents of the file can be explored with dir(), as any other python object.

.psydat files are ideal for batch analysis with a python script and plotting via *matplotlib*. They contain more information than the Excel or csv data files, and can even be used to (re)create those files.

Of particular interest might be the following attributes and methods of the Handler:

entries

a list of dictionaries. Each entry/dictionary in the list represents a single trial's (a single routine 'run'/iteration) data.

saveAsPickle()

a method for saving all of the entries' data in a Python pickle file

saveAsWideText()

a method for saving all of the entrie's data in a .csv file.

If you just want to recover data or first wish to try things out in a familiar format you can put all of the data in a .csv file, very similar to the .csv files that are produced by default when running experiments. The following script assumes you're using a command-line interface (e. g. Terminal on Mac, or the Command Prompt on Windows) where you've opened up a Python shell, and that you have installed as a Python package:

```python
# import PsychoPy function for loading Pickle/JSON data
from psychopy.misc import fromFile
# (replace with the file path to your .psydat file)
fpath = '/Users/my_user/myexperiments/myexperiment/participant_expname_date.psydat'
# load in the data
psydata = fromFile(fpath)
# (replace with the file path to where you want the resulting .csv
# to be saved)
save_path = '/Users/my_user/Desktop/test_out.csv'
# save the data as a .csv file, ie separating the values with a
```

(continues on next page)

(continued from previous page)

```
# comma. 'CSV' simply means 'comma-separated values'
psydata.saveAsWideText(save_path, delim=',')
```

To get started using the data directly in Python, you can try opening a psydat file and printing all its entries:

```
from psychopy.misc import fromFile
# (replace with the file path to your .psydat file)
fpath = 'path/to/file.psydat'
psydata = fromFile(fpath)
for entry in psydata.entries:
    print(entry)
```

Ideally, we should provide a demo script here for fetching and plotting some data (feel free to *contribute*).

2.5.3 Long-wide data file

This form of data file is the default data output from Builder experiments as of v1.74.00. Rather than summarising data in a spreadsheet where one row represents all the data from a single condition (as in the summarised data format), in long-wide data files the data is not collapsed by condition, but written chronologically with one row representing one trial (hence it is typically longer than summarised data files). One column in this format is used for every single piece of information available in the experiment, even where that information might be considered redundant (hence the format is also 'wide').

Although these data files might not be quite as easy to read quickly by the experimenter, they are ideal for import and analysis under packages such as R, SPSS or Matlab.

2.5.4 Excel data file

Excel 2007 files (.xlsx) are a useful and flexible way to output data as a spreadsheet. The file format is open and supported by nearly all spreadsheet applications (including older versions of Excel and also OpenOffice). N.B. because .xlsx files are widely supported, the older Excel file format (.xls) is not likely to be supported by unless a user contributes the code to the project.

Data from are output as a table, with a header row. Each row represents one condition (trial type) as given to the *TrialHandler*. Each column represents a different type of data as given in the header. For some data, where there are multiple columns for a single entry in the header. This indicates multiple trials. For example, with a standard data file in which response time has been collected as 'rt' there will be a heading *rt_raw* with several columns, one for each trial that occurred for the various trial types, and also an *rt_mean* heading with just a single column giving the mean reaction time for each condition.

If you're creating experiments by writing scripts then you can specify the sheet name as well as file name for Excel file outputs. This way you can store multiple sessions for a single subject (use the subject as the filename and a date-stamp as the sheetname) or a single file for multiple subjects (give the experiment name as the filename and the participant as the sheetname).

Builder experiments use the participant name as the file name and then create a sheet in the Excel file for each loop of the experiment. e.g. you could have a set of practice trials in a loop, followed by a set of main trials, and these would each receive their own sheet in the data file.

2.5.5 Delimited text files (.csv, .tsv, .txt)

For maximum compatibility, especially for legacy analysis software, you can choose to output your data as a delimited text file. Typically this would be comma-separated values (.csv file) or tab-delimited (.tsv file). The format of those files is exactly the same as the Excel file, but is limited by the file format to a single sheet.

2.6 Gamma correcting a monitor

Monitors typically don't have linear outputs; when you request luminance level of 127, it is not exactly half the luminance of value 254. For experiments that require the luminance values to be linear, a correction needs to be put in place for this nonlinearity which typically involves fitting a power law or gamma (γ) function to the monitor output values. This process is often referred to as gamma correction.

can help you perform gamma correction on your monitor, especially if you have one of the supported photometers/spectroradiometers.

There are various different equations with which to perform gamma correction. The simple equation (**??**) is assumed by most hardware manufacturers and gives a reasonable first approximation to a linear correction. The full gamma correction equation (**??**) is more general, and likely more accurate especially where the lowest luminance value of the monitor is bright, but also requires more information. It can only be used in labs that do have access to a photometer or similar device.

2.6.1 Simple gamma correction

The simple form of correction (as used by most hardware and software) is this:

$$L(V) = a + kV^\gamma \tag{2.1}$$

where L is the final luminance value, V is the requested intensity (ranging 0 to 1), a, k and γ are constants for the monitor.

This equation assumes that the luminance where the monitor is set to 'black' (V=0) comes entirely from the surround and is therefore not subject to the same nonlinearity as the monitor. If the monitor itself contributes significantly to a then the function may not fit very well and the correction will be poor.

The advantage of this function is that the calibrating system (in this case) does not need to know anything more about the monitor than the gamma value itself (for each gun). For the full gamma equation (**??**), the system needs to know about several additional variables. The look-up table (LUT) values required to give a (roughly) linear luminance output can be generated by:

$$LUT(V) = V^{1/\gamma} \tag{2.2}$$

where V is the entry in the LUT, between 0 (black) and 1 (white).

2.6.2 Full gamma correction

For very accurate gamma correction uses a more general form of the equation above, which can separate the contribution of the monitor and the background to the lowest luminance level:

$$L(V) = a + (b + kV)^\gamma \tag{2.3}$$

This equation makes no assumption about the origin of the base luminance value, but requires that the system knows the values of b and k as well as γ.

The inverse values, required to build the LUT are found by:

$$LUT(V) = \frac{((1-V)b^\gamma + V(b+k)^\gamma)^{1/\gamma} - b}{k} \tag{2.4}$$

This is derived below, for the interested reader. ;-)

And the associated luminance values for each point in the LUT are given by:

$$L(V) = a + (1-V)b^\gamma + V(b+k)^\gamma$$

2.6.3 Deriving the inverse full equation

The difficulty with the full gamma equation (**??**) is that the presence of the b value complicates the issue of calculating the inverse values for the LUT. The simple inverse of (**??**) as a function of output luminance values is:

$$LUT(L) = \frac{((L-a)^{1/\gamma} - b)}{k} \tag{2.5}$$

To use this equation we need to first calculate the linear set of luminance values, L, that we are able to produce the current monitor and lighting conditions and *then* deduce the LUT value needed to generate that luminance value.

We need to insert into the LUT the values between 0 and 1 (to use the maximum range) that map onto the linear range from the minimum, *m*, to the maximum *M* possible luminance. From the parameters in (**??**) it is clear that:

$$\begin{aligned} m &= a + b^\gamma \\ M &= a + (b+k)^\gamma \end{aligned} \tag{2.6}$$

Thus, the luminance value, *L* at any given point in the LUT, *V*, is given by

$$\begin{aligned} L(V) &= m + (M-m)V \\ &= a + b^\gamma + (a + (b+k)^\gamma - a - b^\gamma)V \\ &= a + b^\gamma + ((b+k)^\gamma - b^\gamma)V \\ &= a + (1-V)b^\gamma + V(b+k)^\gamma \end{aligned} \tag{2.7}$$

where V is the position in the LUT as a fraction.

Now, to generate the LUT as needed we simply take the inverse of (**??**):

$$LUT(L) = \frac{(L-a)^{1/\gamma} - b}{k} \tag{2.8}$$

and substitute our $L(V)$ values from (**??**):

$$\begin{aligned} LUT(V) &= \frac{(a + (1-V)b^\gamma + V(b+k)^\gamma - a)^{1/\gamma} - b}{k} \\ &= \frac{((1-V)b^\gamma + V(b+k)^\gamma)^{1/\gamma} - b}{k} \end{aligned} \tag{2.9}$$

2.6.4 References

2.7 OpenGL and Rendering

All rendering performed by uses hardware-accelerated OpenGL rendering where possible. This means that, as much as possible, the necessary processing to calculate pixel values is performed by the graphics card *GPU* rather than by the *CPU*. For example, when an image is rotated the calculations to determine what pixel values should result, and any interpolation that is needed, are determined by the graphics card automatically.

In the double-buffered system, stimuli are initially drawn into a piece of memory on the graphics card called the 'back buffer', while the screen presents the 'front buffer'. The back buffer initially starts blank (all pixels are set to the window's defined color) and as stimuli are 'rendered' they are gradually added to this back buffer. The way in which stimuli are combined according to transparency rules is determined by the *blend mode* of the window. At some point in time, when we have rendered to this buffer all the objects that we wish to be presented, the buffers are 'flipped' such that the stimuli we have been drawing are presented simultaneously. The monitor updates at a very precise fixed rate and the flipping of the window will be synchronised to this monitor update if possible (see *Sync to VBL and wait for VBL*).

Each update of the window is referred to as a 'frame' and this ultimately determines the temporal resolution with which stimuli can be presented (you cannot present your stimulus for any duration other than a multiple of the frame duration).

In addition to synchronising flips to the frame refresh rate, can optionally go a further step of not allowing the code to continue until a screen flip has occurred on the screen, which is useful in ascertaining exactly when the frame refresh occurred (and, thus, when your stimulus actually appeared to the subject). These timestamps are very precise on most computers. For further information about synchronising and waiting for the refresh see *Sync to VBL and wait for VBL*.

If the code/processing required to render all you stimuli to the screen takes longer to complete than one screen refresh then you will 'drop/skip a frame'. In this case the previous frame will be left on screen for a further frame period and the flip will only take effect on the following screen update. As a result, time-consuming operations such as disk accesses or execution of many lines of code, should be avoided while stimuli are being dynamically updated (if you care about the precise timing of your stimuli). For further information see the sections on *Detecting dropped frames* and *Reducing dropped frames*.

2.7.1 Fast and slow functions

The fact that modern graphics processors are extremely powerful; they can carry out a great deal of processing from a very small number of commands. Consider, for instance, the Coder demo *elementArrayStim* in which several hundred Gabor patches are updated frame by frame. The graphics card has to blend a sinusoidal grating with a grey background, using a Gaussian profile, several hundred times each at a different orientation and location and it does this in less than one screen refresh on a good graphics card.

There are three things that are relatively slow and should be avoided at critical points in time (e.g. when rendering a dynamic or brief stimulus). These are:

1. disk accesses
2. passing large amounts of data to the graphics card
3. making large numbers of python calls.

Functions that are very fast:

1. Calls that move, resize, rotate your stimuli are likely to carry almost no overhead
2. Calls that alter the color, contrast or opacity of your stimulus will also have no overhead IF your graphics card supports *OpenGL Shaders*
3. Updating of stimulus parameters for psychopy.visual.ElementArrayStim is also surprisingly fast BUT you should try to update your stimuli using *numpy* arrays for the maths rather than *for…* loops

Notable slow functions in PsychoPy calls:

1. Calls to set the image or set the mask of a stimulus. This involves having to transfer large amounts of data between the computer's main processor and the graphics card, which is a relatively time-consuming process.
2. Any of your own code that uses a Python *for…* loop is likely to be slow if you have a large number of cycles through the loop. Try to 'vectorise' your code using a numpy array instead.

2.7.2 Tips to render stimuli faster

1. Keep images as small as possible. This is meant in terms of **number of pixels**, not in terms of Mb on your disk. Reducing the size of the image on your disk might have been achieved by image compression such as using jpeg images but these introduce artefacts and do nothing to reduce the problem of send large amounts of data from the CPU to the graphics card. Keep in mind the size that the image will appear on your monitor and how many pixels it will occupy there. If you took your photo using a 10 megapixel camera that means the image is represented by 30 million numbers (a red, green and blue) but your computer monitor will have, at most, around 2 megapixels (1960x1080).
2. Try to use square powers of two for your image sizes. This is efficient because computer memory is organised according to powers of two (did you notice how often numbers like 128, 512, 1024 seem to come up when you buy your computer?). Also several mathematical routines (anything involving Fourier maths, which is used a lot in graphics processing) are faster with power-of-two sequences. For the `psychopy.visual.GratingStim` a

texture/mask of this size is **required** and if you don't provide one then your texture will be 'upsampled' to the next larger square-power-of-2, so you can save this interpolation step by providing it in the right shape initially.

3. Get a faster graphics card. Upgrading to a more recent card will cost around £30. If you're currently using an integrated Intel graphics chip then almost any graphics card will be an advantage. Try to get an nVidia or an ATI Radeon card.

2.7.3 OpenGL Shaders

You may have heard mention of 'shaders' on the users mailing list and wondered what that meant (or maybe you didn't wonder at all and just went for a donut!). OpenGL shader programs allow modern graphics cards to make changes to things during the rendering process (i.e. while the image is being drawn). To use this you need a graphics card that supports OpenGL 2.1 and will only make use of shaders if a specific OpenGL extension that allows floating point textures is also supported. Nowadays nearly all graphics cards support these features - even Intel chips from Intel!

One example of how such shaders are used is the way that colors greyscale images. If you provide a greyscale image as a 128x128 pixel texture and set its color to be red then, without shaders, needs to create a texture that contains the 3x128x128 values where each of the 3 planes is scaled according to the RGB values you require. If you change the color of the stimulus a new texture has to be generated with the new weightings for the 3 planes. However, with a shader program, that final step of scaling the texture value according to the appropriate RGB value can be done by the graphics card. That means we can upload just the 128x128 texture (taking 1/3 as much time to upload to the graphics card) and then we each time we change the color of the stimulus we just a new RGB triplet (only 3 numbers) without having to recalculate the texture. As a result, on graphics cards that support shaders, changing colors, contrasts and opacities etc. has almost zero overhead.

2.7.4 Blend Mode

A 'blend function' determines how the values of new pixels being drawn should be combined with existing pixels in the 'frame buffer'.

blendMode = 'avg'

This mode is exactly akin to the real-world scenario of objects with varying degrees of transparency being placed in front of each other; increasingly transparent objects allow increasing amounts of the underlying stimuli to show through. Opaque stimuli will simply occlude previously drawn objects. With each increasing semi-transparent object to be added, the visibility of the first object becomes increasingly weak. The order in which stimuli are rendered is very important since it determines the ordering of the layers. Mathematically, each pixel colour is constructed from opacity*stimRGB + (1-opacity)*backgroundRGB. This was the only mode available before version 1.80 and remains the default for the sake of backwards compatibility.

blendMode = 'add'

If the window *blendMode* is set to 'add' then the value of the new stimulus does not in any way *replace* that of the existing stimuli that have been drawn; it is added to it. In this case the value of *opacity* still affects the weighting of the new stimulus being drawn but the first stimulus to be drawn is never 'occluded' as such. The sum is performed using the signed values of the color representation in , with the mean grey being represented by zero. So a dark patch added to a dark background will get even darker. For grating stimuli this means that contrast is summed correctly.

This blend mode is ideal if you want to test, for example, the way that subjects perceive the sum of two potentially overlapping stimuli. It is also needed for rendering stereo/dichoptic stimuli to be viewed through colored anaglyph glasses.

If stimuli are combined in such a way that an impossible luminance value is requested of any of the monitor guns then that pixel will be out of bounds. In this case the pixel can either be clipped to provide the nearest possible colour, or can be artificially colored with noise, highlighting the problem if the user would prefer to know that this has happened.

2.7.5 Sync to VBL and wait for VBL

will always, if the graphics card allows it, synchronise the flipping of the window with the vertical blank interval (VBL aka VBI) of the screen. This prevents visual artefacts such as 'tearing' of moving stimuli. This does not, itself, indicate that the script also waits for the physical frame flip to occur before continuing. If the *waitBlanking* window argument is set to False then, although the window refreshes themselves will only occur in sync with the screen VBL, the *win.flip()* call will not actually wait for this to occur, such that preparations can continue immediately for the next frame. For rendering purposes this is actually optimal and will reduce the likelihood of frames being dropped during rendering.

By default the Window will also wait for the VBL (*waitBlanking=True*) . Although this is slightly less efficient for rendering purposes it is necessary if we need to know exactly when a frame flip occurred (e.g. to timestamp when the stimulus was physically presented). On most systems this will provide a very accurate measure of when the stimulus was presented (with a variance typically well below 1ms but this should be tested on your system).

2.8 Presenting audio stimuli

Presenting audio stimuli with low latency is more difficult than you might think! PsychoPy has historically used a range of audio libraries to try and solve this issue. In PsychoPy 3.2 we added the option of using Psych

2.8.1 Choice of audio library

If low-latency audio is important to you then **we strongly recommend you use the `ptb` library** which is a Python version of the PsychoPhysics Toolbox audio engine (PsychPortAudio).

Other options:

- *pyo* can be fast with the right hardware/OS, but not on such a range of hardware as the PTB option.
- *sounddevice* is the next best option if *pyo* isn't working for you
- *pygame* we really don't recommend

2.8.2 PsychoPy and the PTB audio engine

The PsychPortAudio engine from Psychophysics Toolbox is considerably faster than any of our previous audio library options. On most hardware it should result in sub-ms precision in audio latencies.

This is achieved by a number of means, including fine-grained control of audio driver settings and by pre-scheduling of sounds for playback.

2.9 Timing Issues and synchronisation

One of the key requirements of experimental control software is that it has good temporal precision. aims to be as precise as possible in this domain and can achieve excellent results depending on your experiment and hardware. It also provides you with a precise log file of your experiment to allow you to check the precision with which things occurred.

Many scientists have asked "Can provide sub-millisecond timing precision?". The short answer is yes it can - 's timing is as good as any software package we've tested (we've tested quite a lot).

BUT there are many components to getting good timing, and many ways that your timing could be less-than-perfect. So if timing is important to you then you should really read this entire section of the manual and you should **test your timing** using dedicated hardware (photodiodes, microphones or, ideally the Black Box Toolkit). We can't emphasise enough how many ways there are for your hardware and/or operating system to break the good timing that is providing.

2.9.1 Can deliver millisecond precision?

The simple answer is 'yes'. PsychoPy's timing is as good as (or in most cases better than) any package out there. For detailed Studies of timing see Bridges et al., 2020

The longer answer is that you should test the timing of your own experimental stimuli on your own hardware. Very often a computer is not optimally configured to produce good timing, and a poorly coded experiment could also destroy your timing (which is one reason we now recommend even good coders use *Builder*!). Many software and hardware manufacturers will suggest that the key to good timing is using computer clocks with high precision (lots of decimal places) but that is not the answer at all. The sources of error in stimulus/response timing are almost never to do with the poor precision of the clock. The following issues are extremely common and **until you actually test your experiment you don't realise that your timing is being affected by them**:

- *Additional delays caused by monitors*: e.g. the monitor taking additional time to process the image before presentation
- *Delays caused by drivers and OS*: Windows, Linux and Mac all perform further processing on the images, depending on settings and this can delay your visual stimulus delivery by a further frame interval or more
- *Delays caused by coding errors*
- *Delays caused by keyboards*
- *Audio delays*

The clocks that uses do have sub-millisecond precision but your keyboard has a latency of 4-25ms depending on your platform and keyboard. You could buy a response pad (e.g. a Labhackers Millikey) for response timing with a sub-ms precision, but note that there will still be an apparent lag that is dependent on the monitor's absolute lag and the position of the stimulus on it.

Also note that the variance in terms of response times between your participants, and from trial to trial within a participant, probably dwarfs that of your keyboard and monitor issues! That said, does aim to give you as high a temporal precision as possible and, in a well-configured system achieves this very well.

2.9.2 Computer monitors

There are several issues with monitors that you should be aware of.

1. *Monitors have fixed refresh rates*
2. *The top of the screen appears 5-15 ms before the bottom*
3. *Additional delays caused by monitors*

Monitors have fixed refresh rates

Most monitors have fixed refresh rates, typically 60 Hz for a flat-panel display. You probably knew that but it's very easy to forget that this means certain stimulus durations won't be possible. If your screen is a standard 60 Hz monitor then your frame period is 1/60 s, roughly 16.7 ms. That means you can generate stimuli that last for 16.7 ms, or 33.3 ms or 50 ms, but you **cannot** present a stimulus for 20, 40, or 60 ms.

The caveat to this is that you can now buy specialist monitors that support variable refresh rates (although not below at least 5 ms between refreshes). These are using a technology called G-Sync (nVidia) or FreeSync (everyone else) and can make use of those technologies but support isn't built in to the library. See the publication by Poth et al (2018) for example code.

The top of the screen appears 5-15 ms before the bottom

For most monitor technologies, the lines of pixels are drawn sequentially from the top to the bottom and once the bottom line has been drawn the screen is finished and the line returns to the top (the Vertical Blank Interval, VBI). Most of your frame interval is spent drawing the lines, with 1-2ms being left for the VBI. This means that the pixels at the bottom are drawn “”up to 10 ms later”’ than the pixels at the top of the screen. At what point are you going to say your stimulus ‘appeared’ to the participant?

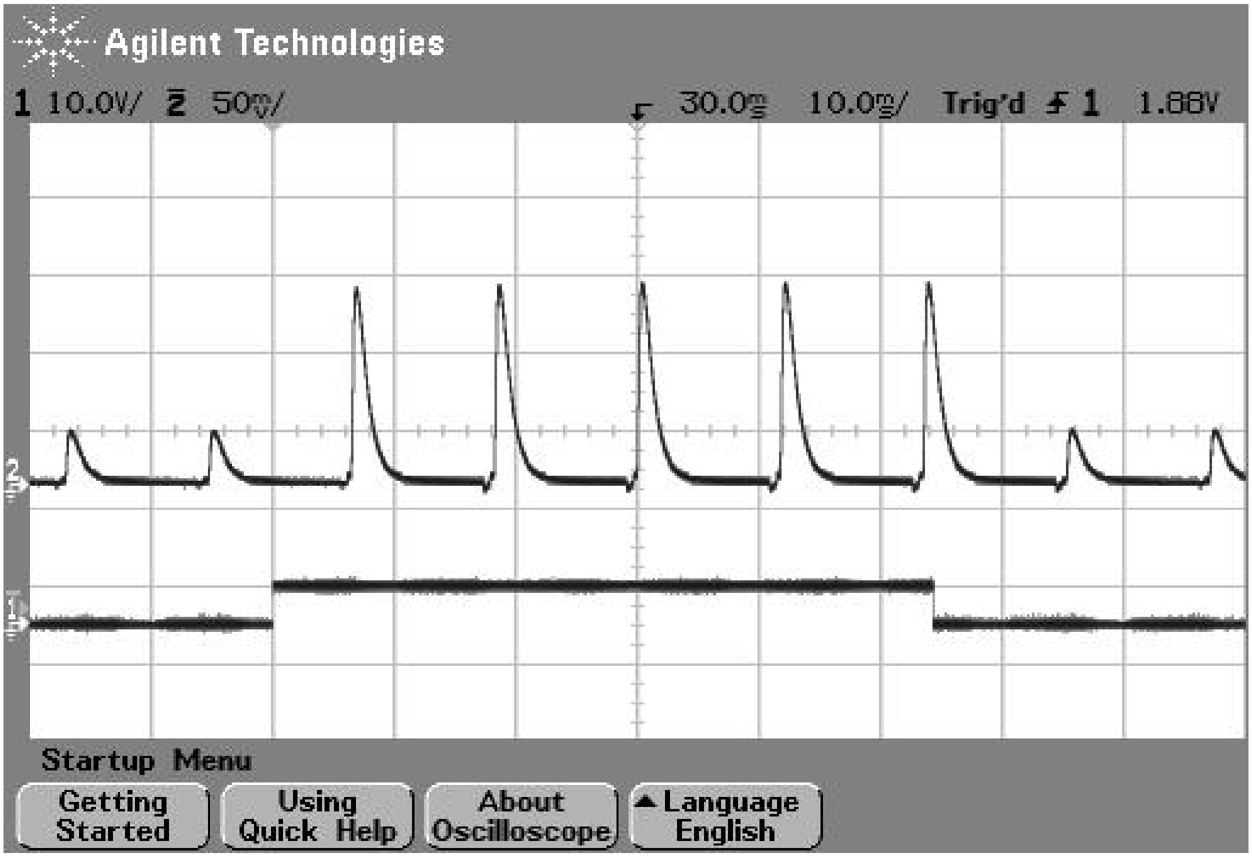


Fig. 2.1: Figure 1: photodiode trace at top of screen. The image above shows the luminance trace of a CRT recorded by a fast photo-sensitive diode at the top of the screen when a stimulus is requested (shown by the square wave). The square wave at the bottom is from a parallel port that indicates when the stimulus was flipped to the screen. Note that on a CRT the screen at any point is actually black for the majority of the time and just briefly bright. The visual system integrates over a large enough time window not to notice this. On the next frame after the stimulus ‘presentation time’ the luminance of the screen flash increased.

Additional delays caused by monitors

Monitors themselves often cause delays **on top of** the unavoidable issue of having a refresh rate. Modern displays often have features to optimize the image, which will be often labelled as modes like “Movie Mode”, Game Mode” etc. If your display has any such settings then you want to turn them off so as not to change your image. Not only do these settings entail altering the color of the pixels that your experiment generator is send to the screen (if you’ve spent time carefully calibrating your colors and then the monitor changes them it would be annoying) but these forms of “post-processing” take time and often a *variable* time.

If your monitor has any such “post-processing” enabled then you might well be seeing an additional 20-30 ms of (variable) lag added to the stimulus onset as a result. This **will not** be detected by psychoPy (or any other system) and will not show up in your log files.

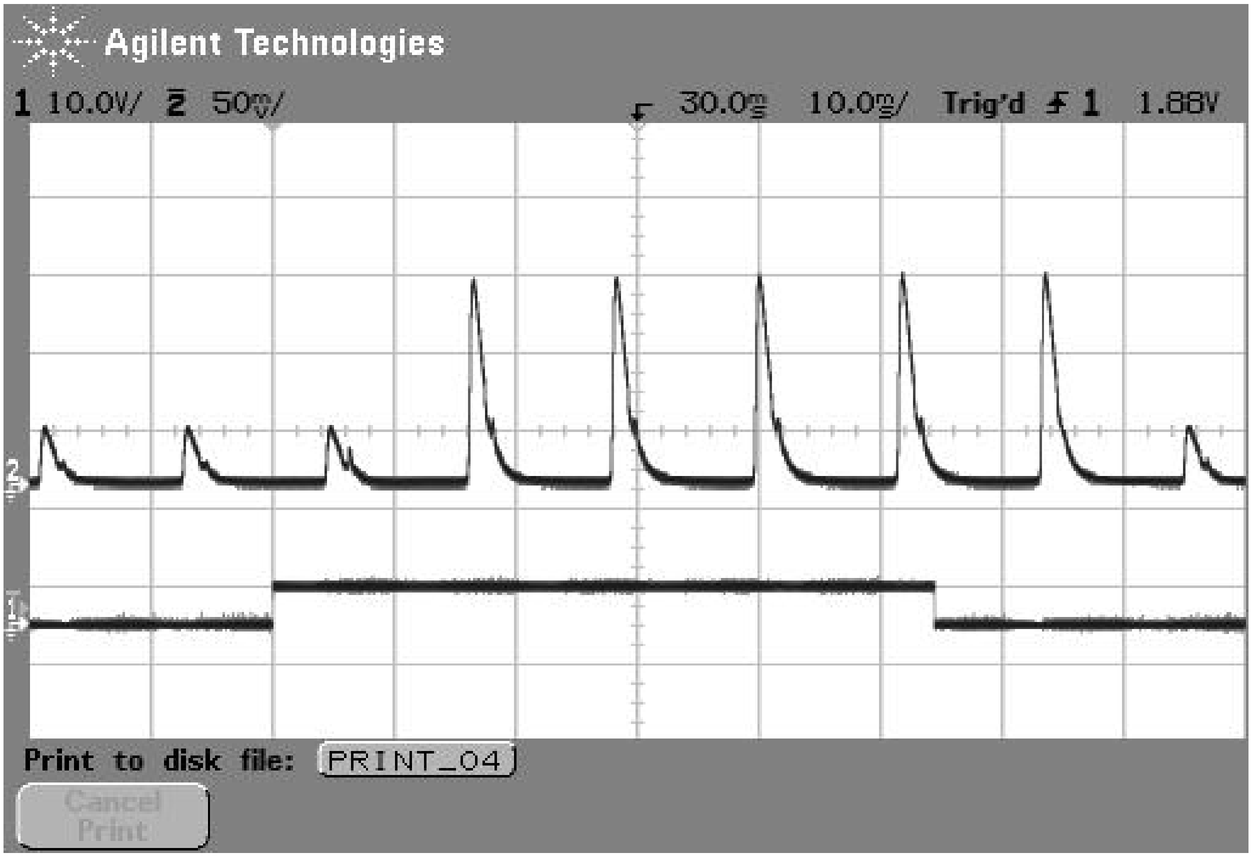


Fig. 2.2: Figure 2: photodiode trace of the same large stimulus at bottom of screen. The image above shows comes from exactly the same script as the above but the photodiode is positioned at the bottom of the screen. In this case, after the stimulus is 'requested' the current frame (which is dark) finishes drawing and then, 10ms later than the above image, the screen goes bright at the bottom.

2.9.3 Delays caused by drivers and OS

All three major operating systems are capable of introducing timing errors into your visual presentations, although these are usually observed as (relatively) constant lags. The particularly annoying factor here is that your experiment might work with very good timing for a while and then the operating system performs and automatic update and the timing gets worse! Again, the only way you would typically know about these sorts of changes is by testing with hardware.

Triple buffering: In general , and similar graphics systems, are expecting a double-buffered rendering pipeline, whereby we are drawing to one copy of the screen (the “back buffer”) and when we have finished drawing our stimuli we “flip” the screen, at which point it will wait for the next screen refresh period and become visible as the “front buffer”. Triple-buffering is a system whereby the images being rendered to the screen are put in a 3rd buffer, and the operating system can do further processing as the rendered image moves from this 3rd buffer to the back buffer. Such a system means that your images all appear exactly one frame later than expected.

Errors caused by triple buffering, either by the operating system or by the monitor, cannot be detected by and will not show up in your log files.

MacOS

The stimulus presentation on MacOS used to be very good, up until version 10.12. In MacOS 10.13 something changed and it appears that a form of triple buffering has been added and, to date, none of the major experiment generators have managed to turn this off. As a result, since MacOS 10.13 stimuli appear always to be presented a screen refresh period later than expected, resulting in a delay of 16.66 ms in the apparent response times to visual stimuli.

Windows 10

In Windows, triple buffering is something that might be turned on by default in your graphics card settings (look for 3D, or OpenGL, settings in the driver control panel to turn this off). The reason it gets used is that it often results in a more consistent frame rate for games, but having the frame appear later then expected is typically bad for experiments!

As well as the graphics card performing triple buffering, the operating system itself (via the Desktop Window Manager) does so under certain conditions: - Anytime a window is used (instead of full-screen mode) Windows 10 now uses triple buffering - having Scaling set to anything other than 100% also results in triple-buffering (presumably Microsoft renders the screen once and then scales it during the next refresh).

There are surely other settings in Windows and the graphics card that will alter the timing performance and, again, until you test these you aren’t likely to know.

Linux

In Linux, again, timing performance of the visual stimuli depends on the graphics card driver but we have also seen timing issues arising from the Window Compositor and with interactions between compositor and driver.

The real complication here is that in Linux there are many different window compositors (Compiz, XFwm, Enlightenment,. . .), as well as different options for drivers to install (e.g. for nVidia cards there is a proprietary nVidia driver as well as an open-source “Nouveau” driver which is often the default but has worse performance).

Ultimately, you need to **test the timing with hardware** and work through the driver/compositor settings to optimise the timing performance.

2.9.4 Delays caused by coding errors

It can be really easy, as a user, to introduce timing errors into your experiment with incorrect coding. Even if you really know what you’re doing, it’s easy to make a silly mistake, and if you don’t really know what you’re doing then all bets are definitely off!!

Common ways for this to happen are to forget the operations that are potentially time-consuming. The biggest of these is the loading of images from disk.

For image stimuli where the image is constant the image should be loaded from disk at the beginning of the script (Builder-generate experiments will do so automatically for you). When an image stimulus has to *change on each trial*, it must be loaded from disk at some point. That typically takes several milliseconds (possibly hundreds of milliseconds for a large image) and while that is happening the screen will not be refreshing. You need to take your image-loading time into account and allow it to occur during a static period of the screen.

In B *Builder* experiments if you set something to update "On every repeat" then it will update as that Routine begins so, if your trial Routine simply begins with 0.5s fixation period, all your stimuli can be loaded/updated in that period and you will have no further problems. Sometimes you want to load/update your stimulus explicitly at a different point in time and then you can insert a *Static Component* Component into your Builder experiment (a "Static Period" in the Python API) and then set your stimulus to update during that period (it will show up as an update option after you insert the Static Component).

The good news is that a lot of the visual timing issues caused by coding problems **are** visible in the *Log Files*, unlike the problems with hardware and operating systems introducing lags.

2.9.5 Delays caused by keyboards

Keyboards are hopeless for timing. We should expand on that. But for now, it's all you need to know! Get yourself a button box, like the LabHackers Millikey.

2.9.6 Audio delays

has a number of settings for audio and the main issue here is that the user needs to know to turn on the optimal settings.

For years we were looking for a library that provided fast reliable audio and we went through an number of libraries to optimize that (pygame was the first, with 100ms latencies, then pyo and sounddevice which were faster).

Most recently we added support for the Psychophysics Toolbox audio library (PsychPortAudio), which Mario Kleiner has ported Python in 2018. With that library we can achieve really remarkable audio timing (thanks to Mario for his fantastic work). But still there are several things you need to check to make use of this library and use it to its full potential:

- Make sure you're running with a 64bit installation of Python3. The PsychPortAudio code has not, and almost certainly will not, be built to support legacy Python installations
- Set the preferences to use it! As of version 3.2.x the PTB backend was not the default. In future versions this will probably be the default, but as of version 3.2.x you need to set to use it (we didn't want to make it the default until it had been used without issue in a number of labs in "the wild").
- Make sure that the library settings are using a high

For further information please see the documentation about the *Sound library*

2.9.7 Non-slip timing for imaging

For most behavioural/psychophysics studies timing is most simply controlled by setting some timer (e.g. a `Clock()`) to zero and waiting until it has reached a certain value before ending the trial. We might call this a 'relative' timing method, because everything is timed from the start of the trial/epoch. In reality this will cause an overshoot of some fraction of one screen refresh period (10ms, say). For imaging (EEG/MEG/fMRI) studies adding 10ms to each trial repeatedly for 10 minutes will become a problem, however. After 100 stimulus presentations your stimulus and scanner will be de-synchronised by 1 second.

There are two ways to get around this:

1. *Time by frames* If you are confident that you *aren't dropping frames* then you could base your timing on frames instead to avoid the problem.
2. *Non-slip (global) clock timing* The other way, which for imaging is probably the most sensible, is to arrange timing based on a global clock rather than on a relative timing method. At the start of each trial you add the (known)

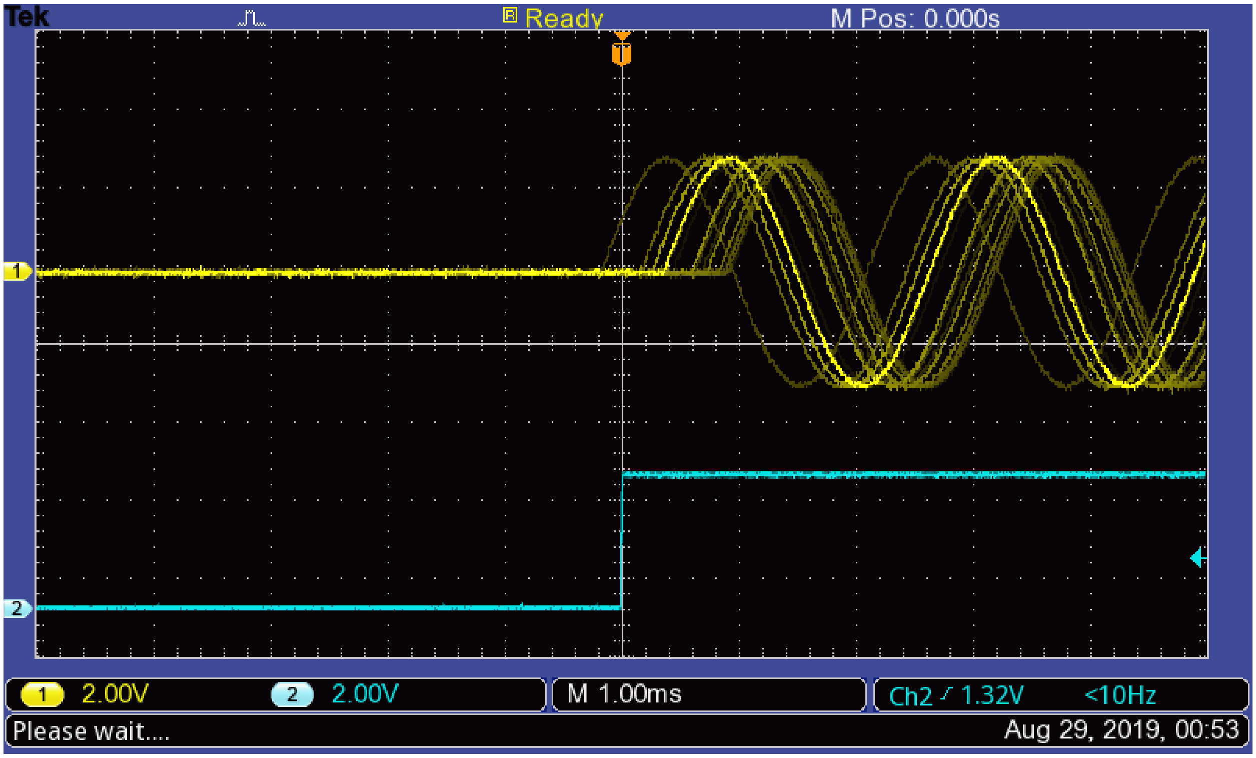


Fig. 2.3: With the new PTB library you can achieve not only sub-millisecond precision, but roughly sub-millisecond lags!! You do need to know how to configure this though and testing it can only be done with hardware.

duration that the trial will last to a *global* timer and then wait until that timer reaches the necessary value. To facilitate this, the `Clock()` was given a new *add()* method as of version 1.74.00 and a `CountdownTimer()` was also added.

The non-slip method can only be used in cases where the trial is of a known duration at its start. It cannot, for example, be used if the trial ends when the subject makes a response, as would occur in most behavioural studies.

Non-slip timing from the Builder

When creating experiments in the *Builder*, will attempt to identify whether a particular *Routine* has a known endpoint in seconds. If so then it will use non-slip timing for this Routine based on a global countdown timer called *routineTimer*. Routines that are able to use this non-slip method are shown in green in the *Flow*, whereas Routines using relative timing are shown in red. So, if you are using PsychoPy for imaging studies then make sure that all the Routines within your loop of epochs are showing as green. (Typically your study will also have a Routine at the start waiting for the first scanner pulse and this will use relative timing, which is appropriate).

2.9.8 Detecting dropped frames

Occasionally you will drop frames if you:

- try to do too much drawing
- do it in an inefficient manner (write poor code)
- have a poor computer/graphics card

Things to avoid:

- recreating textures for stimuli

- building new stimuli from scratch (create them once at the top of your script and then change them using `stim.setOri(ori)()`, *stim.setPos([x,y]. . .)*

Turn on frame time recording

The key sometimes is *knowing* if you are dropping frames. can help with that by keeping track of frame durations. By default, frame time tracking is turned off because many people don't need it, but it can be turned on any time after *`Window`* creation:

```
from psychopy import visual
win = visual.Window([800,600])
win.recordFrameIntervals = True
```

Since there are often dropped frames just after the system is initialised, it makes sense to start off with a fixation period, or a ready message and don't start recording frame times until that has ended. Obviously if you aren't refreshing the window at some point (e.g. waiting for a key press with an unchanging screen) then you should turn off the recording of frame times or it will give spurious results.

Warn me if I drop a frame

The simplest way to check if a frame has been dropped is to get to report a warning if it thinks a frame was dropped:

```
from psychopy import visual, logging
win = visual.Window([800,600])

win.recordFrameIntervals = True

# By default, the threshold is set to 120% of the estimated refresh
# duration, but arbitrary values can be set.
#
# I've got 85Hz monitor and want to allow 4 ms tolerance; any refresh that
# takes longer than the specified period will be considered a "dropped"
# frame and increase the count of win.nDroppedFrames.
win.refreshThreshold = 1/85 + 0.004

# Set the log module to report warnings to the standard output window
# (default is errors only).
logging.console.setLevel(logging.WARNING)

print('Overall, %i frames were dropped.' % win.nDroppedFrames)
```

Show me all the frame times that I recorded

While recording frame times, these are simply appended, every frame to win.frameIntervals (a list). You can simply plot these at the end of your script using matplotlib:

```
import matplotlib.pyplot as plt
plt.plot(win.frameIntervals)
plt.show()
```

Or you could save them to disk. A convenience function is provided for this:

```
win.saveFrameIntervals(fileName=None, clear=True)
```

The above will save the currently stored frame intervals (using the default filename, 'lastFrameIntervals.log') and then clears the data. The saved file is a simple text file.

At any time you can also retrieve the time of the /last/ frame flip using win.lastFrameT (the time is synchronised with logging.defaultClock so it will match any logging commands that your script uses).

'Blocking' on the VBI

As of version 1.62 'blocks' on the vertical blank interval meaning that, once Window.flip() has been called, no code will be executed until that flip actually takes place. The timestamp for the above frame interval measurements is taken immediately after the flip occurs. Run the timeByFrames demo in Coder to see the precision of these measurements on your system. They should be within 1ms of your mean frame interval.

Note that Intel integrated graphics chips (e.g. GMA 945) under win32 do not sync to the screen at all and so blocking on those machines is not possible.

2.9.9 Reducing dropped frames

There are many things that can affect the speed at which drawing is achieved on your computer. These include, but are probably not limited to; your graphics card, CPU, operating system, running programs, stimuli, and your code itself. Of these, the CPU and the OS appear to make rather little difference. To determine whether you are actually dropping frames see *Detecting dropped frames*.

Things to change on your system:

1. make sure you have a good graphics card. Avoid integrated graphics chips, especially Intel integrated chips and especially on laptops (because on these you don't get to change your mind so easily later). In particular, try to make sure that your card supports OpenGL 2.0
2. **shut down as many programs, including background processes. Although modern processors are fast and often have multiple cores, substantial disk/memory accessing can cause frame drops**
    - anti-virus auto-updating (if you're allowed)
    - email checking software
    - file indexing software
    - backup solutions (e.g. TimeMachine)
    - Dropbox
    - Synchronisation software

Writing optimal scripts

1. run in full-screen mode (rather than simply filling the screen with your window). This way the OS doesn't have to spend time working out what application is currently getting keyboard/mouse events.
2. don't generate your stimuli when you need them. Generate them in advance and then just modify them later with the methods like setContrast(), setOrientation() etc…
3. **calls to the following functions are comparatively slow; they require more CPU time than most other functions and then have to send a large amount of data to the graphics card. Try to use these methods in inter-trial intervals. This is especially true when you need to load an image from disk too as the texture.**
    - GratingStim.setTexture()
    - RadialStim.setTexture()
    - TextStim.setText()

4. if you don't have OpenGL 2.0 then calls to setContrast, setRGB and setOpacity will also be slow, because they also make a call to setTexture(). If you have shader support then this call is not necessary and a large speed increase will result.
5. avoid loops in your python code (use numpy arrays to do maths with lots of elements) *Note: numpy arrays will not work for online experiments, which use JavaScript**
6. if you need to create a large number (e.g. greater than 10) similar stimuli, then try the ElementArrayStim (currently not supported for online experiments)

Possible good ideas

It isn't clear that these actually make a difference, but they might).

1. disconnect the internet cable (to prevent programs performing auto-updates?)
2. on Macs you can actually shut down the Finder. It might help. See Alex Holcombe's timing tips page
3. use a single screen rather than two (probably there is some graphics card overhead in managing double the number of pixels?)

2.9.10 Comparing Operating Systems under

This is an attempt to quantify the ability of draw without dropping frames on a variety of hardware/software. The following tests were conducted using the script at the bottom of the page. Note, of course that the hardware fully differs between the Mac and Linux/Windows systems below, but that both are standard off-the-shelf machines.

All of the below tests were conducted with 'normal' systems rather than anything that had been specifically optimised:

- the machines were connected to network
- did not have anti-virus turned off (except Ubuntu had no anti-virus)
- they even all had dropbox clients running
- Linux was the standard (not 'realtime' kernel)

No applications were actively being used by the operator while tests were run.

In order to test drawing under a variety of processing loads the test stimulus was one of:

- a single drifting Gabor
- 500 random dots continuously updating
- 750 random dots continuously updating
- 1000 random dots continuously updating

Common settings:

- Monitor was a CRT 1024x768 100Hz
- all tests were run in full screen mode with mouse hidden

System Differences:

- the iMac was lower spec than the Windows/Linux box and running across two monitors (necessary in order to connect to the CRT)
- the Windows/Linux box ran off a single monitor

Each run below gives the number of dropped frames out of a run of 10,000 (2.7 mins at 100Hz).

_ _	Windows XP (SP3)	Windows 7 Enterprise	Mac OS X 10.6 Snow Leopard	Ubuntu 11.10
Gabor	0	5	0	0
500-dot RDK	0	5	54	3
750-dot RDK	21	7	aborted	1174
1000-dot RDK	776	aborted	aborted	aborted
GPU	Radeon 5400	Radeon 5400	Radeon *2400*	Radeon 5400
GPU driver	Catalyst 11.11	Catalyst 11.11		Catalyst 11.11
CPU	Core Duo 3GHz	Core Duo 3GHz	Core Duo 2.4GHz	Core Duo 3GHz
RAM	4GB	4GB	2GB	4GB

I'll gradually try to update these tests to include:

- longer runs (one per night!)
- a faster Mac
- a real-time Linux kernel

2.9.11 Understand and measuring your timing

There are certain steps that we strongly advise you to take before running an experiment that needs to be temporally precise in PsychoPy, or indeed any other software:

- Read this timing megastudy by Bridges et al (2020) which compares several pieces of behavioural software in terms of their temporal precision. You can find a summary of the results here: *Mega-timing study data*
- Check that your stimulus presentation monitor is not dropping frames. You can do this by running the timeByFrames.py demo. Find this demo in the *Coder* window > demos > timing. The timeByFrames demo examines the precision of your frame flips, and shows the results in a plot similar to the one below:
- Use a photodiode or other physical stimulus detector to fully understand the lag, and more importantly the variability of that lag, between any triggers that you send to indicate the start of your stimulus and when the stimulus actually starts.

2.10 Glossary

Adaptive staircase

An experimental method whereby the choice of stimulus parameters is not pre-determined but based on previous responses. For example, the difficulty of a task might be varied trial-to-trial based on the participant's responses. These are often used to find psychophysical thresholds. Contrast this with the *method of constants*.

CPU

Central Processing Unit is the main processor of your computer. This has a lot to do, so we try to minimise the amount of processing that is needed, especially during a trial, when time is tight to get the stimulus presented on every screen refresh.

CRT

Cathode Ray Tube 'Traditional' computer monitor (rather than an LCD or plasma flat screen).

csv

Comma-Separated Value files Type of basic text file with 'comma-separated values'. This type of file can be opened with most spreadsheet packages (e.g. MS Excel) for easy reading and manipulation.

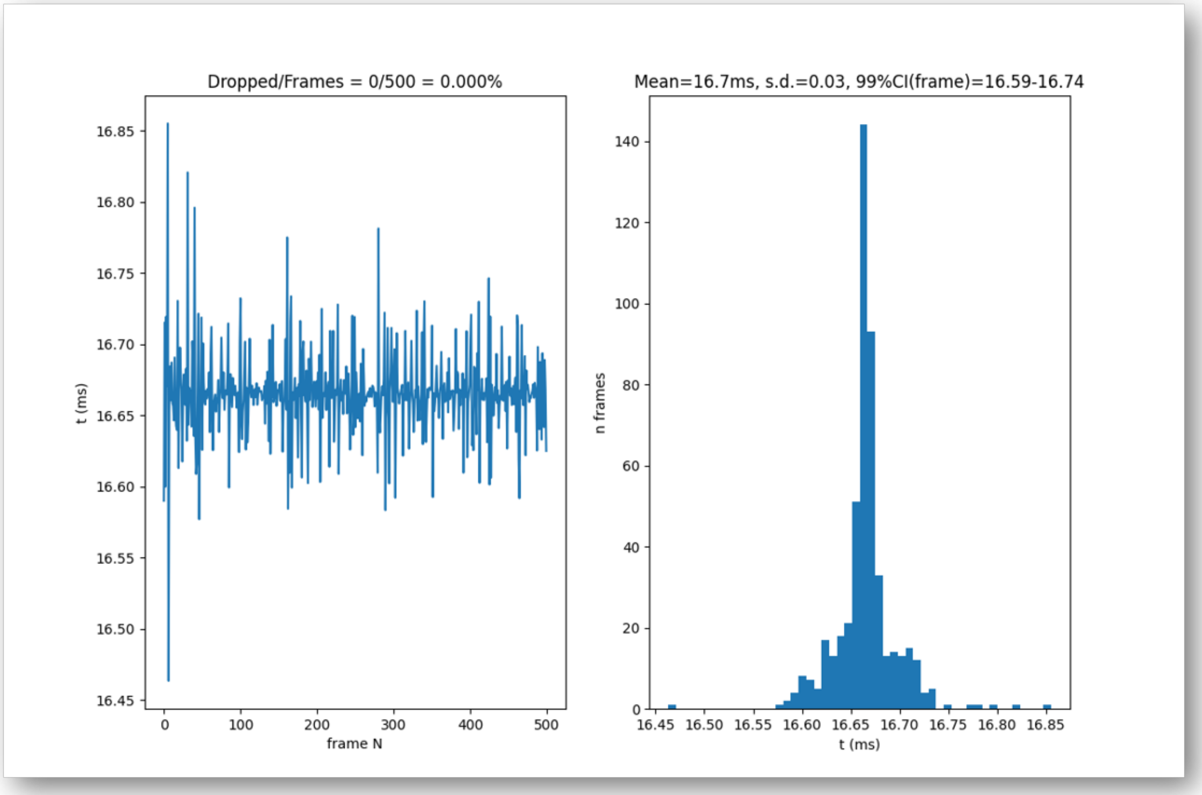


Fig. 2.4: The results here are for a 60Hz monitor, and you can see that there are no dropped frames from the left hand side of the screen, and also the timing of each frame is 16.7ms (shown on the right-hand side of the screen) which is what we would expect from a 60Hz monitor (1000ms/60 = 16.66ms).

GPU

Graphics Processing Unit is the processor on your graphics card. The GPUs of modern computers are incredibly powerful and it is by allowing the GPU to do a lot of the work of rendering that is able to achieve good timing precision despite being written in an interpreted language

Method of constants

An experimental method whereby the parameters controlling trials are predetermined at the beginning of the experiment, rather than determined on each trial. For example, a stimulus may be presented for 3 pre-determined time periods (100, 200, 300ms) on different trials, and then repeated a number of times. The order of presentation of the different conditions can be randomised or sequential (in a fixed order). Contrast this method with the *adaptive staircase*.

VBI

(**Vertical Blank Interval**, aka the Vertical Retrace, or Vertical Blank, VBL). The period in-between video frames and can be used for synchronising purposes. On a CRT display the screen is black during the VBI and the display beam is returned to the top of the display.

VBI blocking

The setting whereby all functions are synced to the VBI. After a call to *`psychopy.visual.Window.flip()`* nothing else occurs until the VBI has occurred. This is optimal and allows very precise timing, because as soon as the flip has occurred a very precise time interval is known to have occurred.

VBI syncing

(aka vsync) The setting whereby the video drawing commands are synced to the VBI. When psychopy.visual.Window.flip() is called, the current back buffer (where drawing commands are being executed) will be held and drawn on the next VBI. This does not necessarily entail *VBI blocking* (because the system may return and continue executing commands) but does guarantee a fixed interval between frames being drawn.

xlsx

Excel OpenXML file format. A spreadsheet data format developed by Microsoft but with an open (published) format. This is the native file format for Excel (2007 or later) and can be opened by most modern spreadsheet applications including OpenOffice (3.0+), google docs, Apple iWork 08.

CHAPTER

THREE

INSTALLATION

3.1 Download

Windows

Stable Tried and tested, this is the release that's been out in the wild for a bit and has already had any post-release bug fixes it needed. The best option for an install that "just works".

PsychoPy 2024.2.4 (py3.10)

Compatibility+ installer (py3.8)

PsychoPy 2024.2.4 compatibility+ (py3.8)

Beta Hot off the presses, this is the latest release with all the newest features. If you're fine with a few bugs so long as you get the cutting edge features as soon as they're ready, this is the install for you.

PsychoPy 2025.1.0beta (py3.10)

Compatibility+ installer (py3.8)

PsychoPy 2025.1.0 compatibility+ (py3.8)

MacOS

Stable Tried and tested, this is the release that's been out in the wild for a bit and has already had any post-release bug fixes it needed. The best option for an install that "just works".

PsychoPy 2024.2.4 (py3.10)

Compatibility+ installer (py3.8)

PsychoPy 2024.2.4 compatibility+ (py3.8)

Beta Hot off the presses, this is the latest release with all the newest features. If you're fine with a few bugs so long as you get the cutting edge features as soon as they're ready, this is the install for you.

PsychoPy 2025.1.0beta (py3.10)

Compatibility+ installer (py3.8)

PsychoPy 2025.1.0 compatibility+ (py3.8)

Linux

Install curl with your package manager. On most distros, curl is already installed.

1. **Download the script:**

```
curl -LOs https://github.com/wieluk/psychopy_linux_installer/releases/
↪latest/download/psychopy_linux_installer
```

2. **Make it executable:**

```
chmod +x psychopy_linux_installer
```

3. **Run the installer:**

- **GUI Mode**:

```
./psychopy_linux_installer --gui
```

Note: curl and *zenity* are required for GUI mode.

- **Command-Line Mode**:

```
./psychopy_linux_installer
```

For a list of available arguments, run:

```
./psychopy_linux_installer --help
```

For more detailed information or to report bugs, please visit the psychopy_linux_installer GitHub page.

For all versions see the PsychoPy releases on github

is distributed under the GPL3 license

3.2 Do I need the Compatibility+ installer?

While PsychoPy has supported Python 3.10 since version 2022.2.0, the *Use PsychoPy version* parameter in Experiment Settings makes it possible to run an experiment using an older version of the PsychoPy library, including versions from before Python 3.10 was supported - meaning they will fail to run if your installed Python is newer than version 3.8. The Compatibility+ installer installs PsychoPy with Python 3.8, allowing you to run these legacy experiments, but losing out on the speed and stability improvements which come with a newer version of Python.

We recommend installing via the Compatibility+ installer **only if necessary**, as besides running legacy experiments, PsychoPy will perform faster and better in Python 3.10.

3.3 Manual installations

See below for options if you don't want to use the Standalone releases:

- *pip install*
- *brew install*
- *Linux*
- *Anaconda and Miniconda*
- *Developers install*

3.3.1 pip install

Now that most python libraries can be installed using *pip* it's relatively easy to manually install and all it's dependencies to your own installation of Python.

The steps are to fetch Python. This method should work on a range of versions of Python but **we strongly recommend you use Python 3.10 or 3.8**. Older Python versions are no longer being tested and may not work correctly. Newer Python versions may not have wheels for all the necessary dependencies even though we believe that PsychoPy's code, itself, is compatible up to at least Python 3.10.

You can install and its dependencies (more than you'll strictly need, depending on the features you use) by:

```
pip install psychopy
```

If you prefer *not* to install *all* the dependencies (e.g. because the platform or Python version you're on doesn't have that dependency easily available) then you could do:

```
pip install psychopy --no-deps
```

and then install them manually. On Windows, if you need a package that isn't available on PyPI you may want to try the unofficial packages by Christoph Gohlke

3.3.2 brew install

This is a user-contributed option and may or may not work.

On a MacOS machine, *brew* can be used to install :

```
brew install --cask psychopy
```

3.3.3 Linux

We are aware that the procedure for installing on Linux is often rather painful. This is not the platform that the core PsychoPy developers currently use so support is less good than on some platforms. Feel free to jump in and help improve it as a contributor! :-)

There used to be neurodebian and Gentoo packages for but these are both badly outdated. We'd recommend you first make sure you have a compatible Python version installed (currently `>=3.8, <3.11`). If you need an older version, you can on Ubuntu for example do:

```
sudo add-apt-repository ppa:deadsnakes/ppa
sudo apt update
sudo apt install python3.10-venv python3.10-dev
python3.10 -m venv path/to/new/psychopyenv  # choose a path of interest!
source path/to/new/psychopyenv/bin/activate
```

Once you have a compatible Python activated, **copy the link to a wxPython wheel** for your platform from:

https://extras.wxpython.org/wxPython4/extras/linux/gtk3/

and having downloaded the right wheel you can then install it with something like:

```
pip install https://extras.wxpython.org/wxPython4/extras/linux/gtk3/ubuntu-22.04/
↪wxPython-4.2.1-cp310-cp310-linux_x86_64.whl
```

`wxPython>=4.0` doesn't have universal wheels yet which is why you have to find and install the correct wheel for your particular flavor of linux. If a wheel is not yet available for your platform (e.g., a new version of Linux), you will have to build it manually. For example, you can use `pip download wxPython`, extract the archive, enter the directory, and try `python setup.py bdist_wheel` to build a wheel yourself. You will likely need to install some system build dependencies. Once it builds, you can install for example with `pip install dist/wxPython*.whl`.

For some reasons wxPython (wx.html2) is using an older version of libwebkitgtk e.g. psychopy will not show up to fix this (of our own risk): sudo add-apt-repository 'deb http://archive.ubuntu.com/ubuntu bionic main universe' sudo apt install -t bionic libwebkitgtk-1.0-0

Finally, you can do:

```
# with --no-deps flag if you want to install dependencies manually
pip install psychopy
```

Building Python PsychToolbox bindings:

The PsychToolbox bindings for Python provide superior timing for sounds and keyboard responses. Unfortunately we haven't been able to build universal wheels for these yet so you may have to build the pkg yourself. That should not be hard. You need the necessary dev libraries installed first:

```
sudo apt-get install libusb-1.0-0-dev portaudio19-dev libasound2-dev
```

and then you should be able to install using pip and it will build the extensions as needed:

```
pip install psychtoolbox
```

3.3.4 Anaconda and Miniconda

Support for conda was contributed and is badly outdated but you may be able to get it working using *pip install* within your conda environment.

Generally we recommend you use StandalonePsychoPy instead, for experiment creation, as an entirely separate app, and use your conda installation for other (e.g. analysis) scripts.

Alternatively if someone wants to jump in and get things working here again that would be appreciated by other users I'm sure.

3.3.5 Developers install

Ensure you have Python 3.8 and the latest version of pip installed:

```
python --version
pip --version
```

Next, follow the instructions to fork and fetch the latest version of the repository.

From the directory where you cloned the latest repository (i.e., where setup.py resides), run:

```
pip install -e .
```

This will install all dependencies to your default Python distribution (which should be Python 3.8). Next, you should create a new shortcut linking your newly installed dependencies to your current version of in the cloned repository. To do this, simply create a new .BAT file containing:

```
"C:\PATH_TO_PYTHON3.8\python.exe C:\PATH_TO_CLONED_PSYCHOPY_REPO\psychopy\app\
↪psychopyApp.py"
```

Alternatively, you can run the psychopyApp.py from the command line:

```
python C:\PATH_TO_CLONED_PSYCHOPY_REPO\psychopy\app\psychopyApp
```

3.4 Recommended hardware

The minimum requirement for is a computer with a graphics card that supports OpenGL. Many newer graphics cards will work well. Ideally the graphics card should support OpenGL version 2.0 or higher. Certain visual functions run much faster if OpenGL 2.0 is available, and some require it (e.g. ElementArrayStim).

If you already have a computer, you can install and the Configuration Wizard will auto-detect the card and drivers, and provide more information. It is inexpensive to upgrade most desktop computers to an adequate graphics card. High-end graphics cards can be very expensive but are only needed for very intensive use.

Generally NVIDIA and ATI (AMD) graphics chips have higher performance than Intel graphics chips so try and get one of those instead.

3.4.1 Notes on OpenGL drivers

On Windows, if you get an error saying **"pyglet.gl.ContextException: Unable to share contexts"** then the most likely cause is that you need OpenGL drivers and your built-in Windows only has limited support for OpenGL (or possibly you have an Intel graphics card that isn't very good). Try installing new drivers for your graphics card **from its manufacturer's web page,** not from Microsoft. For example, NVIDIA provides drivers for its cards here

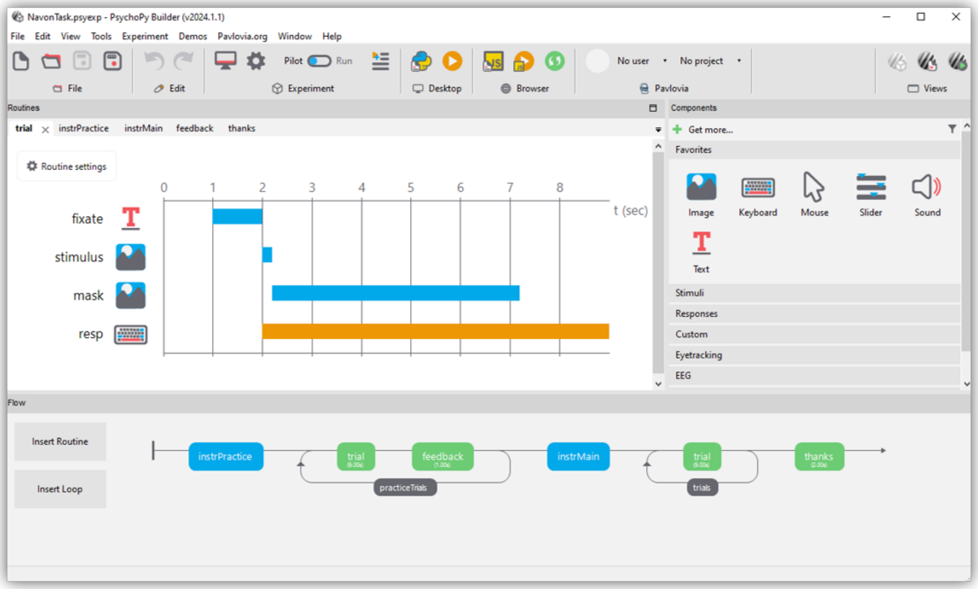
CHAPTER

FOUR

GETTING STARTED

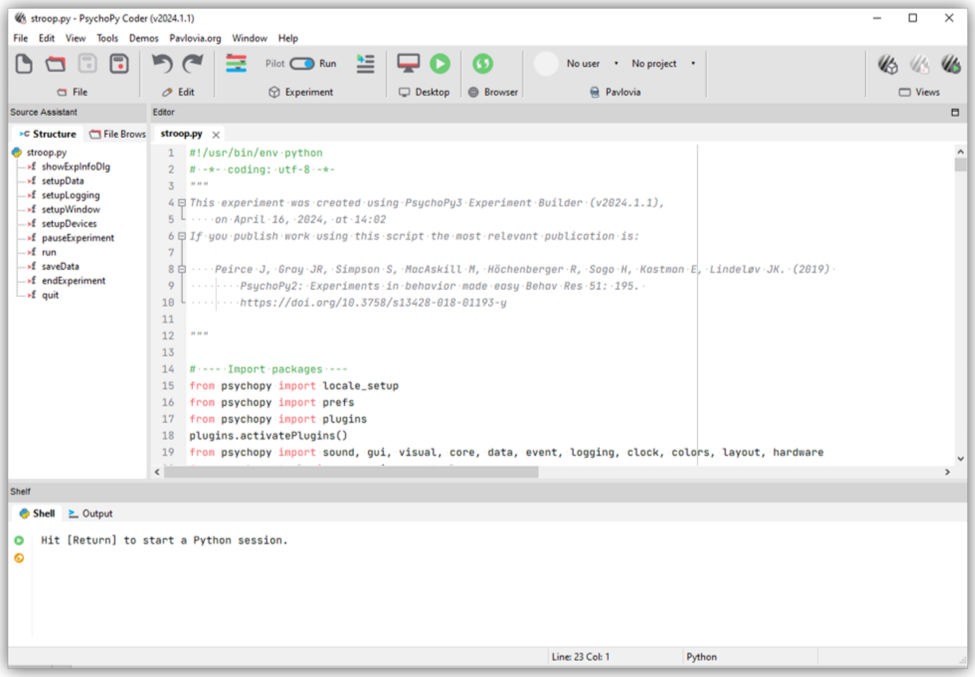
As an application, has two main views: the *Builder* view, and the *Coder* view. It also has a underlying *Reference Manual (API)* that you can call directly.

1. *Builder*. You can generate a wide range of experiments easily from the Builder using its intuitive, graphical user interface (GUI). This might be all you ever need to do. But you can always compile your experiment into a python script for fine-tuning, and this is a quick way for experienced programmers to explore some of PsychoPy's libraries and conventions. **Note: if you are taking a study online we highly advise even experienced coders use Builder view, as the JS version of your experiment will also be generated**



1. *Coder*. For those comfortable with programming, the Coder view provides a basic code editor with syntax highlighting, code folding, and so on. Importantly, it has its own output window and Demo menu. The demos illustrate how to do specific tasks or use specific features; they are not whole experiments. The *Coder tutorials* should help get you going, and the *Reference Manual (API)* will give you the details.

The Builder and Coder views are the two main aspects of the application. If you've installed the StandAlone version of on **MS Windows** then there should be an obvious link to in your > Start > Programs. If you installed the StandAlone version on **macOS** then the application is where you put it (!). On these two platforms you can open the Builder and Coder views from the View menu and the default view can be set from the preferences. **On Linux**, you can start from a command line, or make a launch icon (which can depend on the desktop and distro). If the app is started with flags —–coder (or -c), or —–builder (or -b), then the preferences will be overridden and that view will be created as the app opens.



```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
This experiment was created using PsychoPy3 Experiment Builder (v2024.1.1),
    on April 16, 2024, at 14:02
If you publish work using this script the most relevant publication is:

    Peirce J, Gray JR, Simpson S, MacAskill M, Höchenberger R, Sogo H, Kastman E, Lindeløv JK. (2019)
        PsychoPy2: Experiments in behavior made easy Behav Res 51: 195.
        https://doi.org/10.3758/s13428-018-01193-y

"""

# --- Import packages ---
from psychopy import locale_setup
from psychopy import prefs
from psychopy import plugins
plugins.activatePlugins()
from psychopy import sound, gui, visual, core, data, event, logging, clock, colors, layout, hardware
```

```
Hit [Return] to start a Python session.
```

For experienced python programmers, it's possible to use without ever opening the Builder or Coder. Install the libraries and dependencies, and use your favorite IDE instead of the Coder.

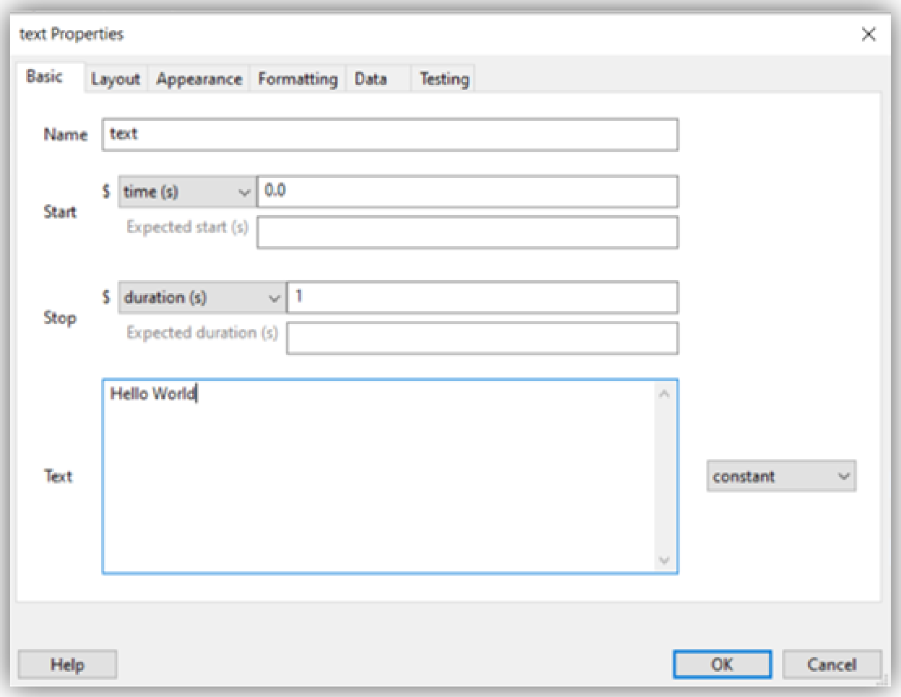
4.1 Builder

When learning a new computer language, the classic first program is simply to print or display "Hello world!". Lets do it.

4.1.1 A first program

Start , and be sure to be in the Builder view.

- If you have poked around a bit in the Builder already, be sure to start with a clean slate. To get a new Builder view, type *Ctrl-N* on Windows or Linux, or *Cmd-N* on Mac.
- Click on a Text component and a Text Properties dialog will pop up.



text Properties

Name: text

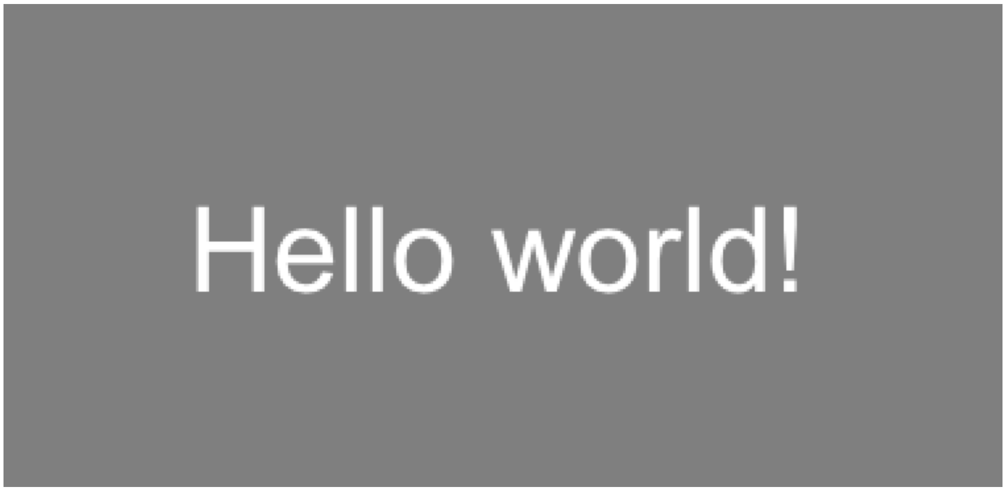
Start: $ time (s) 0.0
Expected start (s):

Stop: $ duration (s) 1
Expected duration (s):

Text: Hello World
constant

- In the *Text* field, replace the default text with your message. When you run the program, the text you type here will be shown on the screen.
- Click OK (near the bottom of the dialog box). (Properties dialogs have a link to online help—an icon at the bottom, near the OK button.)
- Your text component now resides in a routine called *trial*. You can click on it to view or edit it. (Components, Routines, and other Builder concepts are explained in the *Builder documentation*.)
- Back in the main Builder, type *Ctrl-R* (Windows, Linux) or *Cmd-R* (Mac), or use the mouse to click the *Run* icon.

Assuming you typed in “Hello world!”, your screen should have looked like this (briefly):



If nothing happens or it looks wrong, recheck all the steps above; be sure to start from a new Builder view.

What if you wanted to display your cheerful greeting for longer than the default time?

- Click on your Text component (the existing one, not a new one).
- Edit the *Stop duration (s)* to be *3.2*; times are in seconds.
- Click OK.
- And finally *Run*.

When running an experiment, you can quit by pressing the *escape* key (this can be configured or disabled). You can quit from the File menu, or typing *Ctrl-Q* / *Cmd-Q*.

4.1.2 Getting beyond Hello

To do more, you can try things out and see what happens. You may want to consult the *Builder documentation*. Many people find it helpful to explore the Builder demos, in part to see what is possible, and especially to see how different things are done.

A good way to develop your own first experiment is to base it on the Builder demo that seems closest. Copy it, and then adapt it step by step to become more and more like the program you have in mind. Being familiar with the Builder demos can only help this process.

You could stop here, and just use the Builder for creating your experiments. It provides a lot of the key features that people need to run a wide variety of studies. But it does have its limitations. When you want to have more complex designs or features, you'll want to investigate the Coder. As a segue to the Coder, lets start from the Builder, and see how Builder programs work.

4.2 Builder-to-coder

Whenever you run a Builder experiment, will first translate it into python code, and then execute that code.

To get a better feel for what was happening "behind the scenes" in the Builder program above:

- In the Builder, load or recreate your "hello world" program.
- Instead of running the program, explicitly convert it into python: Type *F5*, or click the *Compile* icon:

The view will automatically switch to the Coder, and display the python code. If you then save and run this code, it would look the same as running it directly from the Builder.

It is always possible to go from the Builder to python code in this way. You can then edit that code and run it as a python program. However, you **cannot go from code back to a Builder representation** editing in coder is a one-way street, so, in general, we advise compiling to code is good for understanding what exists but, where possible, make code tweaks in builder itself using code components.

To switch quickly between Builder and Coder views, you can type *Ctrl-L* / *Cmd-L*.

4.3 Coder

Being able to inspect Builder-generated code is nice, but it's possible to write code yourself, directly. With the Coder and various libraries, you can do virtually anything that your computer is capable of doing, using a full-featured modern programming language (python).

For variety, lets say hello to the Spanish-speaking world. knows Unicode (UTF-8).

If you are not in the Coder, switch to it now.

- Start a new code document: *Ctrl-N* / *Cmd-N*.
- Type (or copy & paste) the following:

```
from psychopy import visual, core

win = visual.Window()
msg = visual.TextStim(win, text=u"\u00A1Hola mundo!")

msg.draw()
win.flip()
core.wait(1)
win.close()
```

- Save the file (the same way as in Builder).
- Run the script.

Note that the same events happen on-screen with this code version, despite the code being much simpler than the code generated by the Builder. (The Builder actually does more, such as prompt for a subject number.)

Coder Shell

The shell provides an interactive python interpreter, which means you can enter commands here to try them out. This provides yet another way to send your salutations to the world. By default, the Coder's output window is shown at the bottom of the Coder window. Click on the Shell tab, and you should see python's interactive prompt, >>>:

```
PyShell in |PsychoPy| - type some commands!

Type "help", "copyright", "credits" or "license" for more information.
>>>
```

At the prompt, type:

```
>>> print(u"\u00A1Hola mundo!")
```

You can do more complex things, such as type in each line from the Coder example directly into the Shell window, doing so line by line:

```
>>> from psychopy import visual, core
```

and then:

```
>>> win = visual.Window()
```

and so on—watch what happens each line:

```
>>> msg = visual.TextStim(win, text=u"\u00A1Hola mundo!")
>>> msg.draw()
>>> win.flip()
```

and so on. This lets you try things out and see what happens line-by-line (which is how python goes through your program).

CHAPTER

FIVE

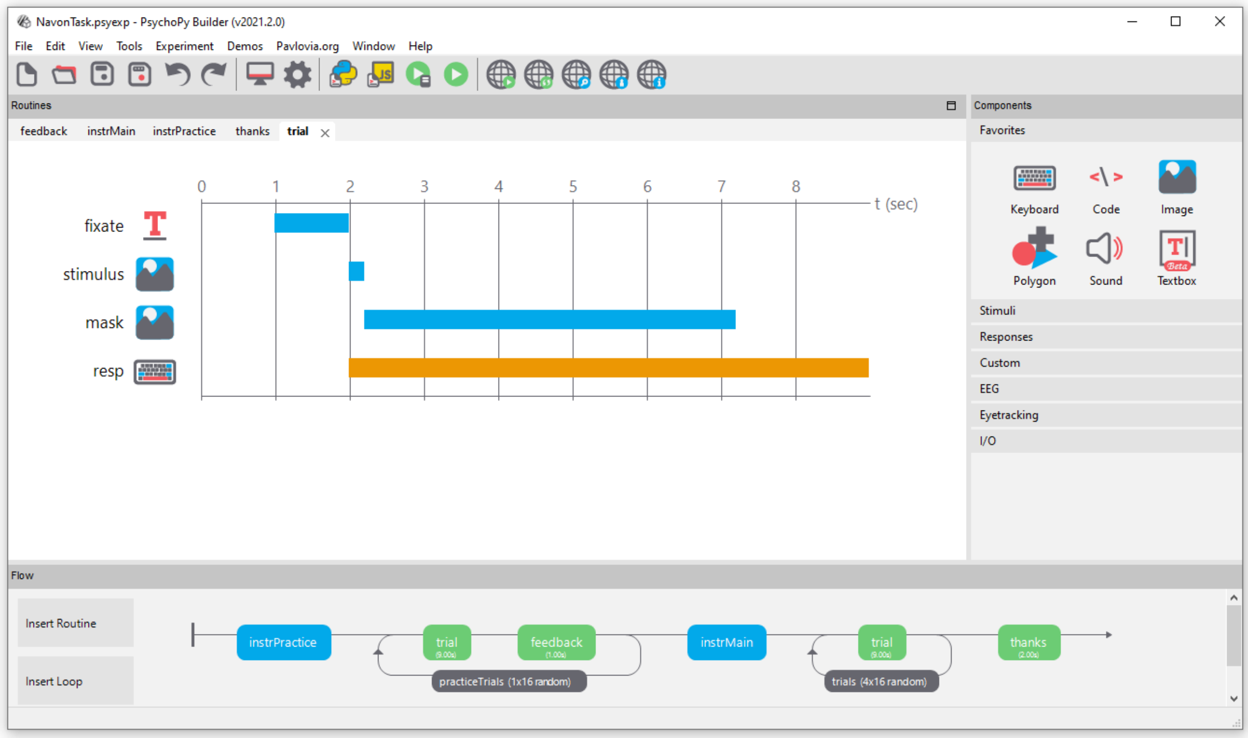
BUILDER

5.1 Building experiments in a GUI

Making your experiments using the builder is the approach that we generally recommend. Why would we (a team of programmers) recommend using a GUI?:

- It's much faster to make experiments
- Your experiment will be less likely to have bugs (experiments coded from scratch can very easily contain errors - even when made by the best of programmers!).
- You can easily make an experiment to run **online in a browser**. builder view is writing you a python script "under the hood" of your experiment, but if you want to run an experiment online it can also compile a javascript version of your task using PsychoPy's sister library PsychoJS. Remember that PsychoJS is younger than - so remember to check the status of online options *before* making an experiment you plan to run online! The easiest way to host a study online from is through the platform, and builder has inbuilt integration to interact with this platform.

There are a number of tutorials on how to get started making experiments in builder on the PsychoPy Youtube channel as well as several written tutorials and Experiment Recipes. You can also find a range of materials for teaching using builder view.



Contents:

5.1.1 Builder concepts

Routines and Flow

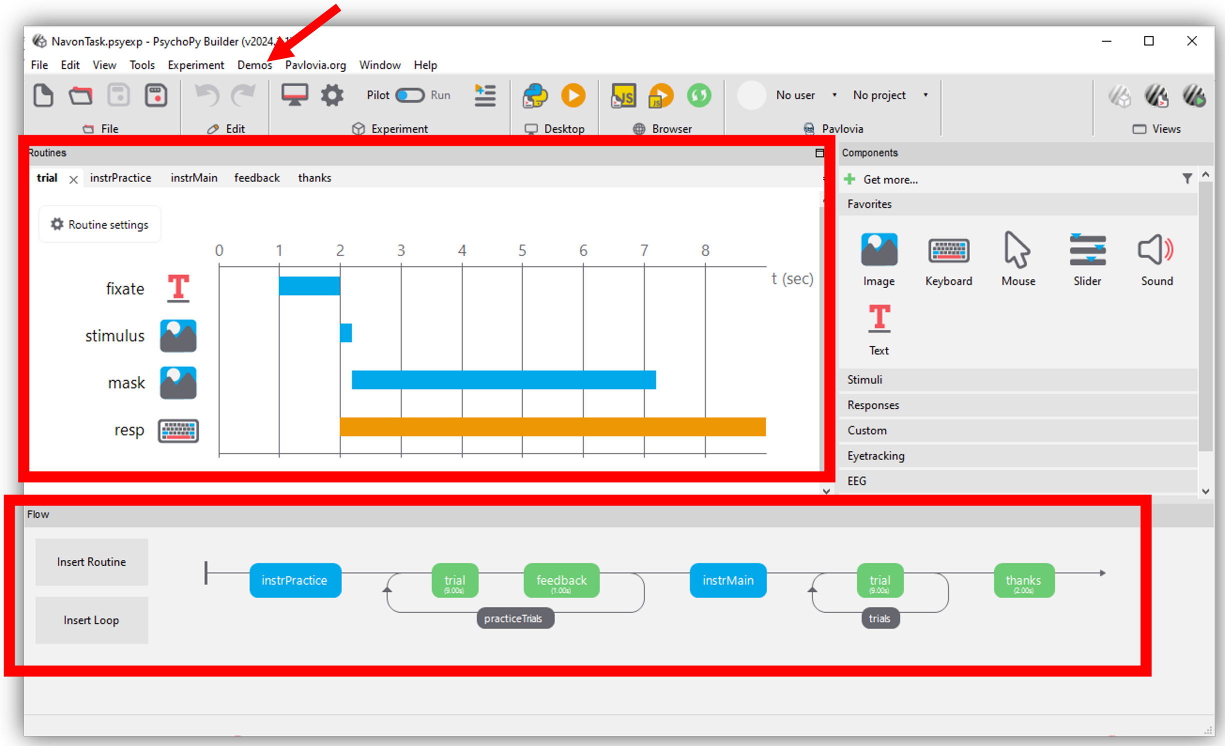
The Builder view of the application is designed to allow the rapid development of a wide range of experiments for experimental psychology and cognitive neuroscience experiments.

The Builder view comprises two main panels for viewing the experiment's *Routines* (upper left) and another for viewing the *Flow* (lower part of the window).

An experiment can have any number of *Routines*, describing the timing of stimuli, instructions and responses. These are portrayed in a simple track-based view, similar to that of video-editing software, which allows stimuli to come on go off repeatedly and to overlap with each other.

The way in which these *Routines* are combined and/or repeated is controlled by the *Flow* panel. All experiments have exactly one *Flow*. This takes the form of a standard flowchart allowing a sequence of routines to occur one after another, and for loops to be inserted around one or more of the *Routines*. The loop also controls variables that change between repetitions, such as stimulus attributes.

If it is your first time opening , we highly recommend taking a look at the large number of inbuilt demos that come with . This can be done through selecting *Demos > unpack demos* within your application. Another good place to get started is to take a look at the many openly available demos at pavlovia.org you can view an intro to Pavlovia at our Youtube channel.



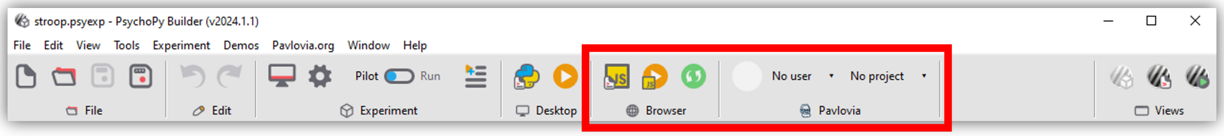
The |PsychoPy| builder, the Routines panel an the Flow are highlighted, if you are new to |PsychoPy|, we recommend starting by unpacking your demos and exploring the example tasks

The components panel

You can add components to an experiment by selecting components from the *Components panel*. This is currently divided into 7 sections:

- *Favorites* - your commonly used components
- *Stimuli* - components used to present a stimulus (e.g. a visual image or shape, or an auditory tone or file)
- *Responses* - stimulu used to gather responses (e.g. keyboards or mouse components - amongst many others!)
- *Custom* - builder can be used to make a fair few complex experiments now, but for added flexibility, you can add code components at any point in an experiment (e.g. for providing response-dependant feedback).
- *EEG* - can actually be used with a range of EEG devices. Most of these are interacted with through delivering a trigger through the parallel port (see I/O below), or serial port (see ../api/serial.html). However, Builder has inbuilt support (i.e. no need for code snippets) for working with Emotiv EEG, you can view a Youtube tutorial on how to use Emotiv EEG with PsychoPy here.
- *Eyetracking* - 2021.2 released inbuilt supprort for eyetrackers! had supported eye tracker research for a while, but not via components in builder. You can learn more about these from the more specific components.html info.
- *I/O* - I/O stands for "input/output" under the hood this is ../api/iohub.html, this is useful for if you are working with external hardware devices requiring communication via the parallel port (e.g. EEG).

Making experiments to go online



Buttons to interact with pavlovia.org from your experiment builder

Before making an experiment to go online, it is a good idea to check the status of online options - remember PsychoJS (the javascript sister library of) is younger that - so not everything can be done online yet! but for most components there are prototype work arounds to still make things possible (e.g. RDKs and staircases). You can learn more about taking experiments online from builder via the online documentation.

5.1.2 Routines

An experiment consists of one or more Routines. A Routine might specify the timing of events within a trial or the presentation of instructions or feedback. Multiple Routines can then be combined in the *Flow*, which controls the order in which these occur and the way in which they repeat.

To create a new Routine, you can either select “Insert Routine” from within your flow panel or use the Experiment menu. The display size of items within a routine can be adjusted (see the View menu).

Within a Routine there are a number of components. These components determine the occurrence of a stimulus, or the recording of a response. Any number of components can be added to a Routine. Each has its own line in the Routine view that shows when the component starts and finishes in time, and these can overlap.

For now the time axis of the Routines panel is fixed, representing seconds (one line is one second). If you choose to present your stimuli based on another timing unit, e.g. number of frames (more precise), you can specify the “Expected duration” within the component - that will mean that this component still appears on your routine timeline

5.1.3 Flow

In the Flow panel a number of *Routines* can be combined to form an experiment. For instance, your study may have a *Routines* that presented initial instructions and waited for a key to be pressed, followed by a *Routines* that presented one trial which should be repeated 5 times with various different parameters set. All of this is achieved in the Flow panel. You can adjust the display size of the Flow panel (see View menu).

Adding Routines

The *Routines* that the Flow will use should be generated first (although their contents can be added or altered at any time). To insert a *Routines* into the Flow click the appropriate button in the left of the Flow panel or use the Experiment menu. A dialog box will appear asking which of your *Routines* you wish to add. To select the location move the mouse to the section of the flow where you wish to add it and click on the black disk.

Loops

Loops control the repetition of *Routines* and the choice of stimulus parameters for each. To insert a loop use the button on the left of the Flow panel, or the item in the Experiment menu of the Builder. The start and end of a loop is set in the same way as the location of a *Routines* (see above). Loops can encompass one or more *Routines* and other loops (i.e. they can be nested).

As with components in *Routines*, the loop must be given a name, which must be unique and made up of only alphanumeric characters (underscores are allowed). I would normally use a plural name, since the loop represents multiple repeats of something. For example, *trials*, *blocks* or *epochs* would be good names for your loops.

It is usually best to use trial information that is contained in an external file (.xlsx or .csv). When inserting a *loop* into the *flow* you can browse to find the file you wish to use for this. An example of this kind of file can be found in the

Stroop demo (trialTypes.xlsx). The column names are turned into variables (in this case text, letterColor, corrAns and congruent), these can be used to define parameters in the loop by putting a $ sign before them e.g. *$text*.

As the column names from the input file are used in this way they must have legal variable names i.e. they must be unique, have no punctuation or spaces (underscores are ok) and must not start with a digit.

The parameter *Is trials* exists because some loops are not there to indicate trials *per se* but a set of stimuli within a trial, or a set of blocks. In these cases we don't want the data file to add an extra line with each pass around the loop. This parameter can be unchecked to improve (hopefully) your data file outputs. [Added in v1.81.00]

Loop types

You can use a number of different "Loop Types" in , this controls the way in which the trials you have fed into the "Conditions" field are presented. Imagine you have a conditions file that looks like this:

```
letter
a
b
c
```

After saving this as a spreadsheet (.xlsx or .csv), we could then add this to the "Conditions" field of our loop. Let's imagine we want to present each letter twice, so we set *nReps* to 2. We could then use the following Loop Types:

- **Random** - present a - b in a random order, because we have nReps at 2, this would be repeated twice e.g. `[c, a, b, a, c, b]`
- **Full Random** - present a - b in a random order but also take into account the number of nReps. Here, imagine that rather than having 3 items in the bag that we sample from, and repeat this twice, we instead have 6 items int he bag that are randomly sampled from. This would mean that with fullRandom, but not random, it would be possible to get the following order of trials e.g. `[a, a, b, c, c, b]` - notice that a was sampled twice in the first 2 trials.
- **sequential** - present the rows in the order they are set i nt he spreadsheet. Currently does not have inbuilt support for specific randomisation constraints, so if you need a specific pseudorandom order, preset this in your spreadsheet file and use a "sequential" loopType.
- **staircase** - for use with adaptive procedures, create an output variable called `level` that can then be used to set the parameter of a stimulus (e.g. it's opacity) in an adaptive fashion. This allows researchers to converge upon a participants threshold by adjusting the value of `level` in accordance with performance.
- **interleaved staircases** - for use with multiple staircases that are interleaved. This can also be used to implement other staircasing algorithms such as QUEST (Watson and Pelli, 1983) via `QuestHandler`.

Using a Staircase

Using a staircase procedure to control your loop allows the implementation of adaptive methods. That is, aspects of a trial can depend on (or "adapt to") how a subject has responded earlier in the study. This could be, for example, simple up-down staircases where an intensity value is varied trial-by-trial according to certain parameters, or a stop-signal paradigm to assess impulsivity.

To use a staircase, you'll need to set a 'correct' and an 'incorrect' response to the stimuli in your experiment. This is because the estimate produced by staircases is dependent on your participant's responses; the value will decrease when a participant is 'correct' and increase when the participant is 'incorrect'.

There are currently three types of staircase in PsychoPy:

- Simple
- QUEST

- QUEST Plus

You can add just one of these staircases, or you can choose to interleave two or more.

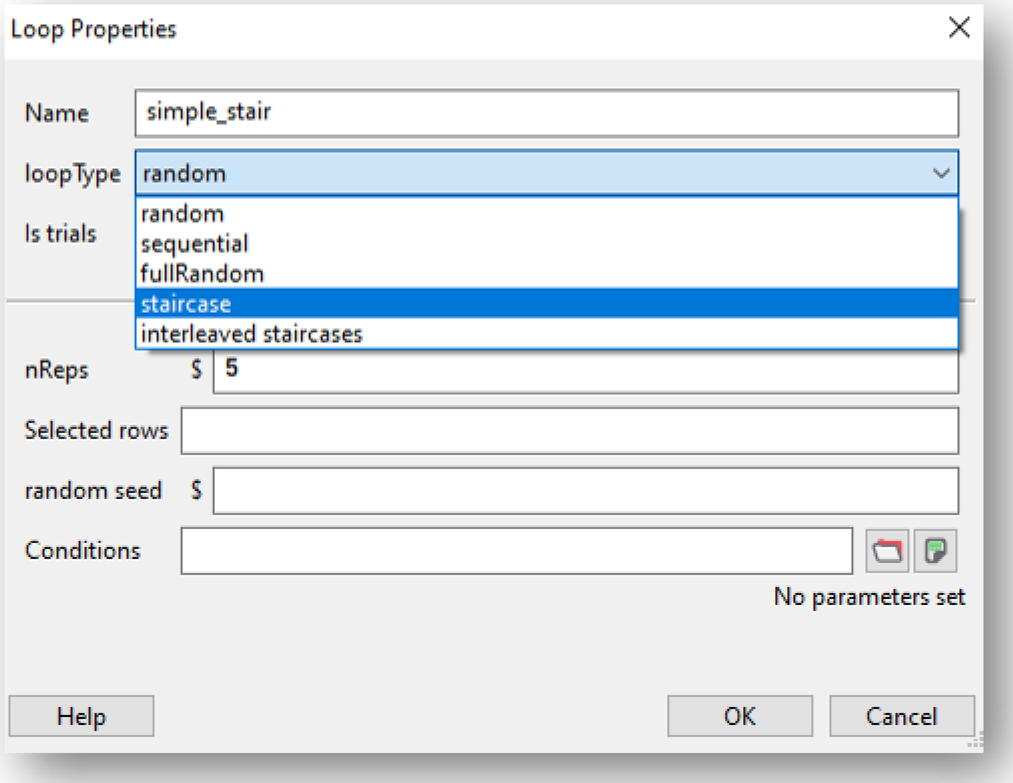
Only QUEST is currently supported for online use.

5.1.4 Using a simple staircase

A simple staircase allows you to input the step sizes that you want the staircase to take when a user gets an answer correct or incorrect.

To add one simple staircase

- To add just one simple staircase, you'll firstly need to add a loop around the routines you want to repeat. Then, from the loop type drop-down list select "staircase":



Loop Properties

Name: simple_stair

loopType: random

Is trials

nReps $ 5

Selected rows

random seed $

Conditions

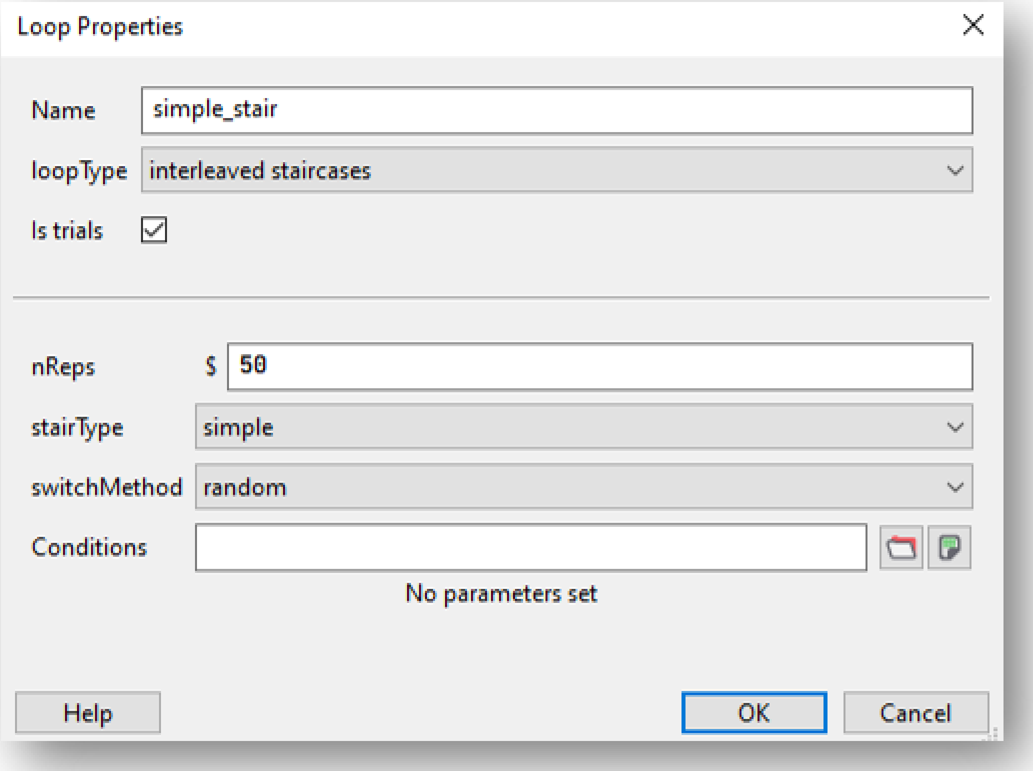
No parameters set

- You'll now see a list of parameters:
  - nReps: The minimum number of trials in the staircase. If the staircase has not reached the required number of reversals then it will continue.
  - start value: The initial value for the staircase.
  - max value: The largest legal value for the staircase, which can be used to prevent it reaching impossible contrast values, for instance.
  - min value: The smallest legal value for the staircase.
  - step sizes: The size of steps as a single value or a list. For a single value the step size is fixed. For a list the step size will progress to the next value at each reversal.

  - step type: The type of steps that should be taken each time:
    - 'lin' - This will simply add or subtract that amount at each step.
    - 'log' - This will add or subtract a certain number of log units at each step (note that this will prevent your value ever reaching zero or less).
    - 'db' - This will add or subtract a certain number of decibels at each step (note that this will prevent your value ever reaching zero or less).
  - N up: The number of 'incorrect' (or 0) responses before the staircase level increases.
  - N down: The number of 'correct' (or 1) responses before the staircase level decreases.
  - nReversals: The minimum number of reversals (i.e., times that the staircase changes direction when an answer is correct/incorrect) that must occur before the staircase ends.
- Complete these fields as required for your experiment and click OK to save the loop.
- Now that you have your conditions set up, you will need to *use* the estimates that are being produced by the staircase to control the particular aspect of the stimulus that you're investigating (contrast, or opacity for example). To do this, simply use the variable *$level* as the value for that parameter and set every repeat!

To add more than one simple staircase

- To add more than one staircase, add a loop in the same way as above but select "Interleaved staircases" from the loop type drop-down list, then "simple" from the stair type drop-down:

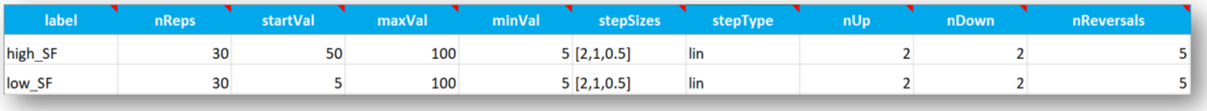


Name: simple_stair
loopType: interleaved staircases
Is trials: ☑
nReps: $ 50
stairType: simple
switchMethod: random
Conditions:
No parameters set

- Set nReps to the minimum number of trials to run.
- Next, use the switch method drop-down list to select whether you want to switch between your staircases sequentially or randomly. Let's imagine that you have four staircases that you want to interleave. Choosing sequential

  would mean that on the first trial staircase one is used, on the next trial staircase two is used, then staircase three followed by staircase four. Then we go back to staircase one on the fifth trial. Choosing random would randomly choose from one of the four staircases to use on each trial.

- Now, you'll need to create a conditions file that contains the following column headers:
  - label: So that you can distinguish between the different staircases in your data output, add a *label* column containing a name for each of your staircases.
  - nReps: The minimum number of trials in the staircase. If the staircase has not reached the required number of reversals then it will continue.
  - startVal: The initial value for the staircase.
  - maxVal: The largest legal value for the staircase, which can be used to prevent it reaching impossible contrast values, for instance.
  - minVal: The smallest legal value for the staircase.
  - stepSizes: The size of steps as a single value or a list. For a single value the step size is fixed. For a list the step size will progress to the next value at each reversal.
  - stepType: The type of steps that should be taken each time:
    - 'lin' - This will simply add or subtract that amount at each step.
    - 'log' - This will add or subtract a certain number of log units at each step (note that this will prevent your value ever reaching zero or less).
    - 'db' - This will add or subtract a certain number of decibels at each step (note that this will prevent your value ever reaching zero or less).
  - nUp: The number of 'incorrect' (or 0) responses before the staircase level increases.
  - nDown: The number of 'correct' (or 1) responses before the staircase level decreases.
  - nReversals: The minimum number of reversals (i.e., times that the staircase changes direction when an answer is correct/incorrect) that must occur before the staircase ends.
- You'll then need to input values for each of your staircases. For example:



label	nReps	startVal	maxVal	minVal	stepSizes	stepType	nUp	nDown	nReversals
high_SF	30	50	100	5	[2,1,0.5]	lin	2	2	5
low_SF	30	5	100	5	[2,1,0.5]	lin	2	2	5

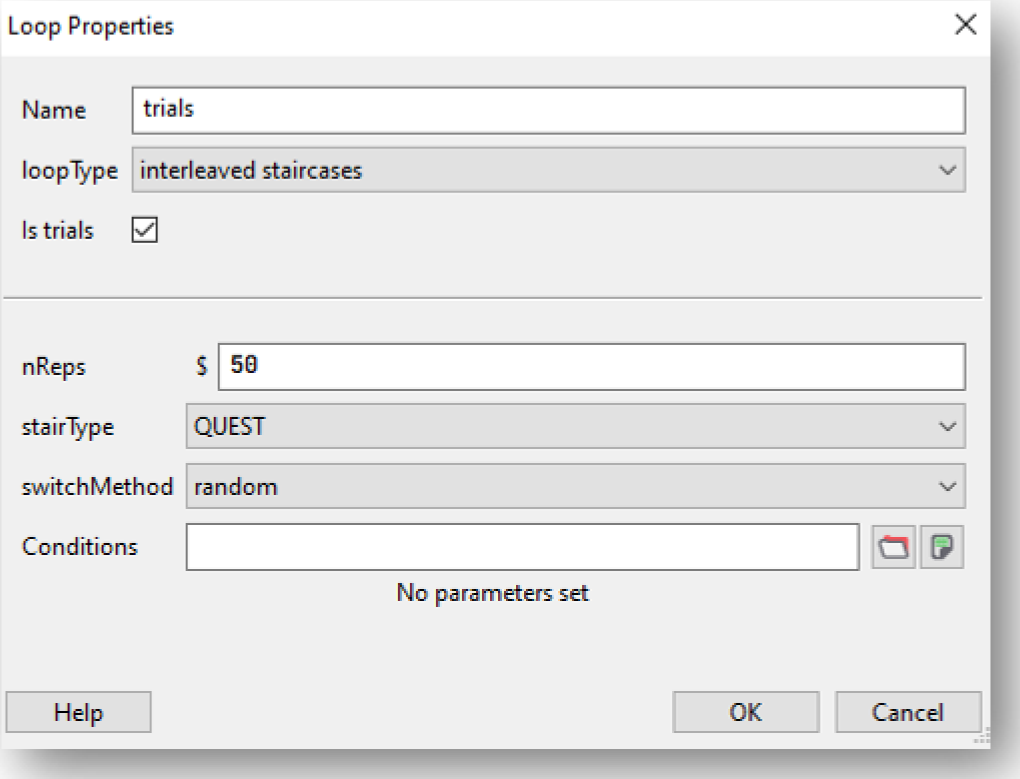
Fig. 5.1: This example has two staircases, one that will start with a high spatial frequency and one that will start with a low spatial frequency.

- Use the variable *$level* in exactly the same way as you would with one staircase - this will update on every repeat automatically.

5.1.5 Using a QUEST staircase

Rather than setting the step sizes manually, as with a simple staircase, the QUEST staircase procedure produces estimates that are based on the stimuli and the observer's responses in the preceding trials. Watson and Pelli (1983) reported QUEST which uses a Bayesian method to estimate the position of the psychometric function. For full information please see their paper in the first instance.

- To add a QUEST staircase, you'll firstly need to add a loop around the routines you want to repeat. Then, from the loop type drop-down list select "Interleaved staircases" and "QUEST" from the stair type drop-down:



Loop Properties

Name: trials

loopType: interleaved staircases

Is trials ☑

nReps $ 50

stairType: QUEST

switchMethod: random

Conditions:

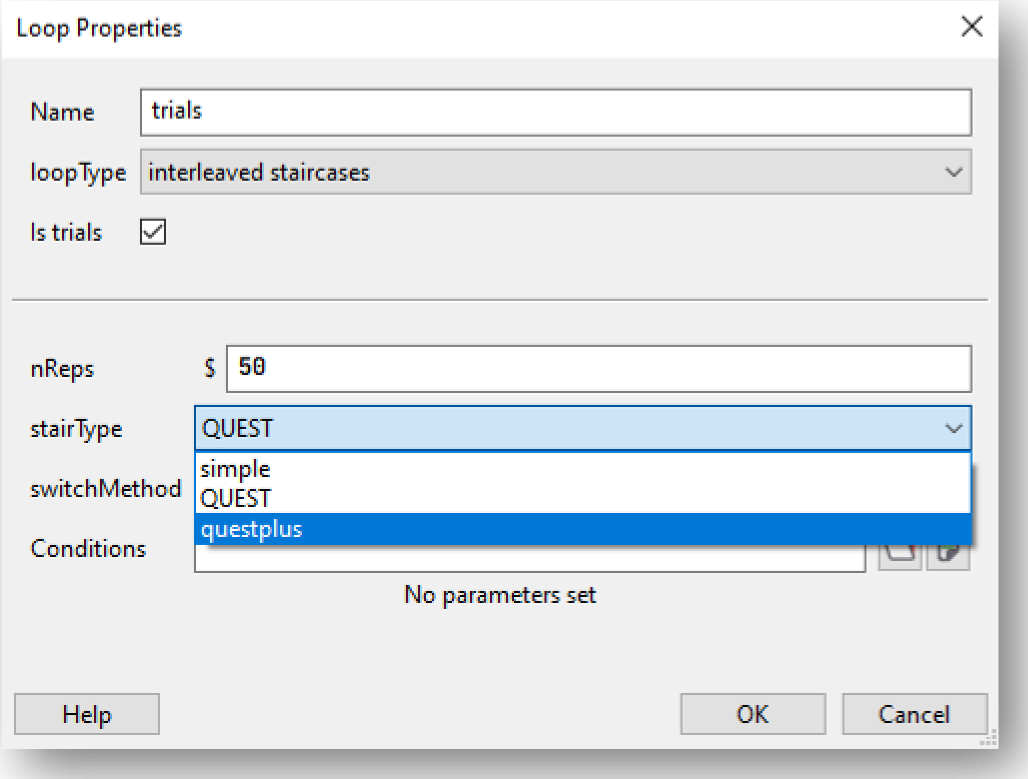
No parameters set

- Set nReps to the minimum number of trials to run.
- If you're using more than one staircase, use the switch method drop-down list to select whether you want to switch between your staircases sequentially or randomly. If you're only using one staircase you can just leave this set to the default value.
- Now, you'll need to create a conditions file that contains the following column headers/variables:
  - label: The label given to the staircase.
  - startVal: The initial value for the staircase.
  - startValSd: Standard deviation of your starting guess threshold. Be generous with the SD as QUEST will have trouble finding the true threshold if it's more than one SD from your initial guess.
  - pThreshold: Your threshold criterion expressed as probability of response==1. Typical values for pThreshold are: 0.82 which is equivalent to a 3 up 1 down standard staircase; 0.63 which is equivalent to a 1 up 1 down standard staircase; 0.5 in a yes-no task and 0.75 in a 2-AFC task
  - method: The method used to determine the next threshold estimate to test. Choose from 'mean', 'mode' or 'quantile'. The default value is quantile.
  - beta: This controls the steepness of the psychometric function (or slope).
  - delta: This is the lapse rate - the fraction of trials that the participant lapses attention and guesses blindly. The default value is 0.01.
  - gamma: The value that is scored while the participant is guessing. Watson and Pelli (1983) state that "The parameter gamma specifies the probability of a success at zero intensity: for two-alternative forced choice it is 0.5, for n-alternative forced choice it is n to the -1 ; for yes-no, it is the false alarm rate."
  - grain: Grain: This is the quantization (step size) of the internal table, e.g., 0.01.
  - minVal **Use this along with maxVal when running the staircase locally (i.e., not online)**: The minimum value that the staircase will return (good for preventing impossible contrast values, for instance).
  - maxVal **Use this along with minVal when running the staircase locally (i.e., not online)**: The maximum value that the staircase will return (good for preventing impossible contrast values, for instance).
  - range (**Use this when running the staircase online**): This is the intensity difference between the largest and smallest value, centered on startVal. Be generous with the range so that you don't exclude possible values for the threshold estimate.
- Complete these fields as required for your experiment and click OK to save the loop.
- Add as many staircases as you need to the conditions file.
- Now that you have your conditions set up, you will need to *use* the estimates that are being produced by the staircase to control the particular aspect of the stimulus that you're investigating (contrast, or opacity for example). To do this, simply use the variable *$level* as the value for that parameter and set every repeat!

5.1.6 Using a QUEST Plus staircase

QUEST Plus is an extension of the original QUEST procedure set out by Watson and Pelli (1983), by Watson (2017). Read the paper here for a complete explanation of the QUEST Plus procedure.

- To add a QUEST Plus staircase, you'll firstly need to add a loop around the routines you want to repeat. Then, from the loop type drop-down list select "Interleaved staircases" and "QUEST Plus" from the stair type drop-down:
- Now, you'll need to create a conditions file that contains the following column headers/variables:
  - label: The label given to the staircase.
  - nTrials: The number of trials to run.



Loop Properties

Name: trials

loopType: interleaved staircases

Is trials ☑

nReps $ 50

stairType: QUEST

switchMethod

Conditions

No parameters set

  - intensityVals: The complete set of stimulus levels. These do not have to just be intensity, they can be contrasts, durations or weights etc.
  - thresholdVals: The complete set of possible threshold values.
  - slopeVals: The complete set of possible slope values.
  - lowerAsymptoteVals: The complete set of possible values of the lower asymptote. This corresponds to false-alarm rates in yes-no tasks, and to the guessing rate in n-AFC tasks. Therefore, when performing an n-AFC experiment, the collection should consist of a single value only (e.g., *[0.5]* for 2-AFC, *[0.33]* for 3-AFC, *[0.25]* for 4-AFC, etc.).
  - lapseRateVals: The complete set of possible lapse rate values. The lapse rate defines the upper asymptote of the psychometric function, which will be at *1 - lapse rate*.
  - responseVals: The complete set of possible response outcomes. Currently, only two outcomes are supported: the first element must correspond to a successful response/stimulus detection, and the second one to an unsuccessful or incorrect response. For example, in a yes-no task, you would use *['Yes', 'No']*, and in an n-AFC task,`['Correct', 'Incorrect']`; or, alternatively, you could use *[1, 0]* in both cases.
  - prior: The prior probabilities to assign to the parameter values.
  - startIntensity: The very first intensity (or stimulus level) to present.
  - stimScale: The scale on which the stimulus intensities (or stimulus levels) are provided. Currently supported are the log scale, *log10*; decibels, *dB*; and a linear scale, *linear*.
  - stimSelectionMethod: How to select the next stimulus. *minEntropy* will select the stimulus that will minimize the expected entropy. *minNEntropy* will randomly pick pick a stimulus from the set of stimuli that will produce the smallest, 2nd-smallest, . . . , N-smallest entropy. This can be used to ensure some variation in the stimulus selection (and subsequent presentation) procedure. The number *N* will then have to be specified via the *stimSelectionOption* parameter.
  - stimSelectionOptions: This parameter further controls how to select the next stimulus in case *stimSelectionMethod=minNEntropy*. The dictionary supports two keys:*N* and *maxConsecutiveReps*. *N* defines the number of "best" stimuli (i.e., those which produce the smallest *N* expected entropies) from which to randomly select a stimulus for presentation in the next trial. *maxConsecutiveReps* defines how many times the exact same stimulus can be presented on consecutive trials. For example, to randomly pick a stimulus from those which will produce the 4 smallest expected entropies, and to allow the same stimulus to be presented on two consecutive trials max, use *stimSelectionOptions=dict(N=4, maxConsecutiveReps=2)*. To achieve reproducible results, you may pass a seed to the random number generator via the *randomSeed* key.
  - paramEstimationMethod: How to calculate the final parameter estimate. *mean* returns the mean of each parameter, weighted by their respective posterior probabilities. *mode* returns the the parameters at the peak of the posterior distribution.
- Complete these fields as required for your experiment and click OK to save the loop.
- Add as many staircases as you need to the conditions file.
- Now that you have your conditions set up, you will need to *use* the estimates that are being produced by the staircase to control the particular aspect of the stimulus that you're investigating (contrast, or opacity for example). To do this, simply use the variable *$level* as the value for that parameter and set every repeat!

In the standard *Loop types* you would use all the rows/conditions within your conditions file. However there are often times when you want to select a subset of your trials before randomising and repeating.

The parameter *Select rows* allows this. You can specify which rows you want to use by inserting values here:

- *0,2,5* gives the 1st, 3rd and 6th entry of a list - Python starts with index zero)
- *$random(4)*10* gives 4 indices from 0 to 9 (so selects 4 out of 10 conditions)

- *5:10* selects the 6th to 10th rows
- *$myIndices* uses a variable that you've already created

Note in the last case that *5:8* isn't valid syntax for a variable so you cannot do:

```
myIndices = 5:8
```

but you can do:

```
myIndices = slice(5,8) #python object to represent a slice
myIndices = "5:8" #a string that PsychoPy can then parse as a slice later
myIndices = "5:8:2" #as above but
```

Note that uses Python's built-in slicing syntax (where the first index is zero and the last entry of a slice doesn't get included). You might want to check the outputs of your selection in the Python shell (bottom of the Coder view) like this:

```
>>> range(100)[5:8] #slice 5:8 of a standard set of indices
[5, 6, 7]
>>> range(100)[5:10:2] #slice 5:8 of a standard set of indices
[5, 7, 9, 11, 13, 15, 17, 19]
```

Check that the conditions you wanted to select are the ones you intended!

Once you have a loop around the routine you want to repeat, you can use the variables created in your conditions file to update any parameter within your routine. For example, let's say that you have a conditions file that looks like this:

```
letter
a
b
c
```

You could then add a Text component and in the *text* field type `$letter` and then set the corresponding dropsown box to "set every repeat". This indicates that you want the value of this parameter to change on each iteration of your loop, and the value of that parameter on each loop will correspond to the value of "letter" drawn on each trial.

Note

You only need to use the $ sign if that field name does not already contain a $ sign! You also don't need several dollar signs in a field e.g. you wouldn't set the position of a stimulus on each repeat using `($myX, $myY)` instead you would just use `$(myX, myY)` - this is because the dollar sign indicates that this field will now accept python code, rather than that this value corresponds to a variable.

5.1.7 Blocks of trials and counterbalancing

Note that for PsychoPy version 2024.1 onwards you can use a a Standalone Counterbalance Routine for counterbalancing.

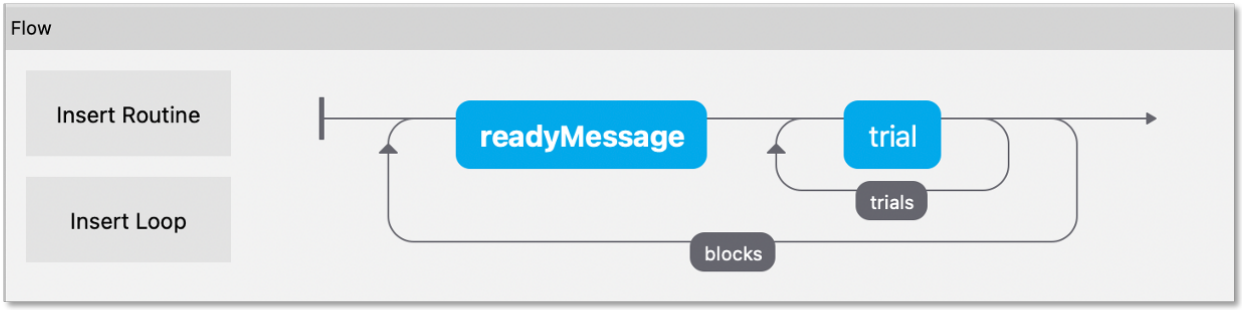
Many people ask how to create blocks of trials, how to randomise them, and how to counterbalance their order. This isn't all that hard, although it does require a bit of thinking!

Blocking similar conditions

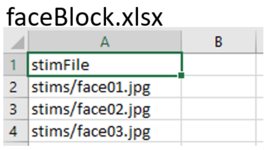
The key thing to understand is that you should not create different Routines for different trials in your blocks (if at all possible). Try to define your trials with a single Routine. For instance, let's imagine you're trying to create an experiment that presents a block of pictures of houses or a block of faces. It would be tempting to create a Routine called *presentFace* and another called *presentHouse* but you actually want just one called *presentStim* (or just *trial*) and then set that to differ as needed across different stimuli.

This example is included in the Builder demos, as of 1.85, as "images_blocks".

You can add a loop around your trials, as normal, to control the trials within a block (e.g. randomly selecting a number of images) but then you will have a second loop around this to define how the blocks change. You can also have additional Routines like something to inform participants that the next block is about to start.

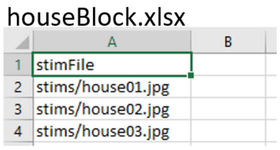


So, how do you get the block to change from one set of images to another? To do this create three spreadsheets, one for each block, determining the filenames within that block, and then another to control which block is being used:



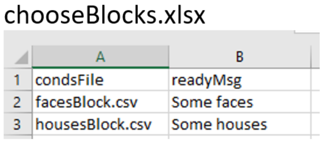
faceBlock.xlsx

	A	B
1	stimFile	
2	stims/face01.jpg	
3	stims/face02.jpg	
4	stims/face03.jpg	



houseBlock.xlsx

	A	B
1	stimFile	
2	stims/house01.jpg	
3	stims/house02.jpg	
4	stims/house03.jpg	



chooseBlocks.xlsx

	A	B
1	condsFile	readyMsg
2	facesBlock.csv	Some faces
3	housesBlock.csv	Some houses

Setting up the basic conditions. The facesBlock, and housesBlock, files look more like your usual conditions files. In this example we can just use a variable *stimFile* with values like *stims/face01.jpg* and *stims/face02.jpg* while the housesBlock file has *stims/house01.jpg* and *stims/house02.jpg*. In a real experiment you'd probably also have response keys andsuchlike as well.

So, how to switch between these files? That's the trick and that's what the other file is used for. In the *chooseBlocks.xlsx* file you set up a variable called something like *condsFile* and that has values of *facesBlock.xlsx* and *housesBlock.xlsx*. In the outer (blocks) loop you set up the conditions file to be *chooseBlocks.xlsx* which creates a variable *condsFile*. Then, in the inner (trials) loop you set the conditions file not to be any file directly but simply *$condsFile*. Now, when starts this loop it will find the current value of *condsFile* and insert the appropriate thing, which will be the name of an conditions file and we're away!

Your *chooseBlocks.xlsx* can contain other values as well, such as useful identifiers. In this demo you could add a value *readyText* that says "Ready for some houses", and "Ready for some faces" and use this in your get ready Routine.

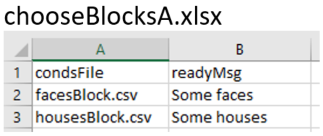
Variables that are defined in the loops are available anywhere within those. In this case, of course, the values in the outer loop are changing less often than the values in the inner loop.

Counterbalancing similar conditions

Counterbalancing is simply an extension of blocking. Until now, we have a *randomised block design*, where the order of blocks is set to random. At the moment we also only have one repeat of each block, but we could also present more than one repeat of each block by controlling the number of rows assigned to each block in our 'chooseBlocks' file.

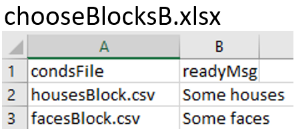
In a counterbalanced design you want to control the order explicitly and you want to provide a different order for different groups of participants. Maybe group A always gets faces first, then houses, and group B always gets houses first, then faces.

Now we need to create further conditions files, to specify the exact orders we want, so we'd have something like *chooseBlockA.xlsx* and *chooseBlockB.xlsx*:



chooseBlocksA.xlsx

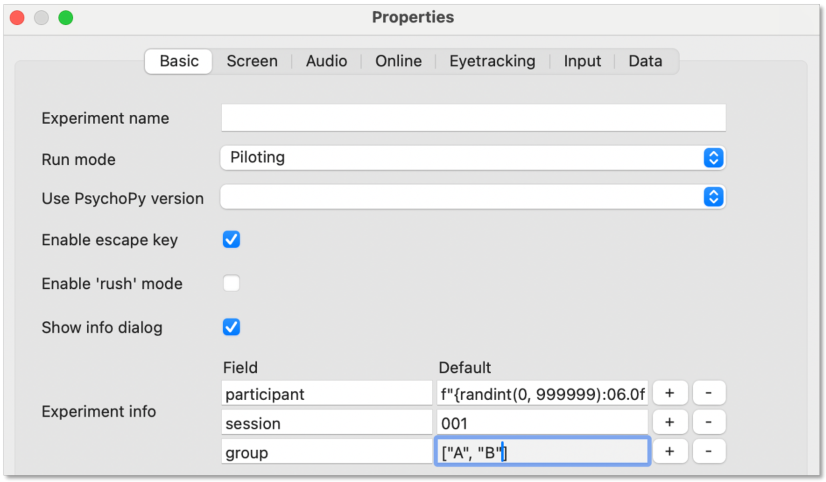
	A	B
1	condsFile	readyMsg
2	facesBlock.csv	Some faces
3	housesBlock.csv	Some houses



chooseBlocksB.xlsx

	A	B
1	condsFile	readyMsg
2	housesBlock.csv	Some houses
3	facesBlock.csv	Some faces

The last part of the puzzle is how to assign participants to groups. For this you *could* write a Code Component that would generate a variable for you (*if…..: groupFile = "groupB.xlsx"*) but the easiest thing is probably that you, the experimenter, chooses this using the GUI we present at the start of the experiment. So, we add a field to our GUI using experiment settings:



Experiment name:
Run mode: Piloting
Use PsychoPy version:
Enable escape key: ☑
Enable 'rush' mode: □
Show info dialog: ☑
Experiment info:

Field	Default
participant	f"{randint(0, 999999):06.0f
session	001
group	["A", "B"]

Note that entering a *list* as the default input will present us with a dropdown in our GUI.

Finally, we set parameters of our *blocks* loop to use the method 'sequential' (because we are using a predefined order) and we enter the following into the conditions field: ` $"chooseBlocks"+expInfo['group']+".xlsx" ` This will concatenate the string "chooseBlocks" with our selected group ("A" or "B") and the required file extension (in this case "xlsx") in order to select the correct order.

Even though our outer loop is now sequential, your inner loop still probably wants to be random (to shuffle the image order within a block).

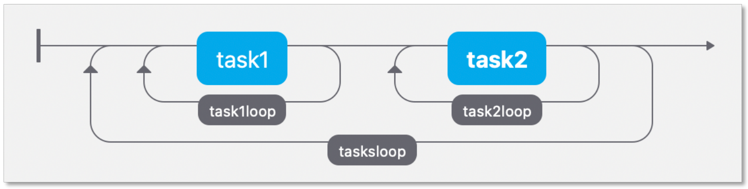
Counterbalancing different subtasks

What happens in situations where we have totally different tasks or trial types we need to counterbalance (e.g. an auditory stroop and an n-back task). The following method is an extension of the logic used in the 'branchedExp' demo available in builderview. You can also download an example .

What is key to understanding how this method works is that in a loop "nReps" (or "Num. Repeats" - depending on what version of PsychoPy you have) controls how many times that loop will be presented. **Setting nReps to 0 means**

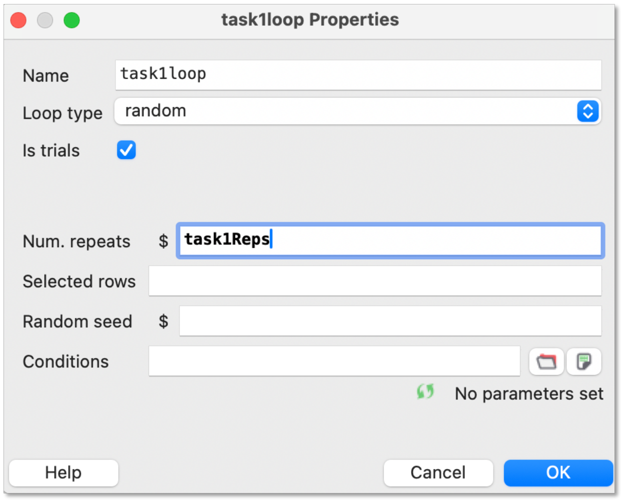
everything in that loop will be skipped - handy eh! We can use this for turning routines "off and on" when working through a set of trials or subtasks.

So, imagine we have 2 "subtasks" or "trial types". Our flow might look something like this:



Here we have 2 totally different tasks/trials, each with its own loop. Now imagine one participant is presented with these tasks using the order Task1 -> Task2 whilst another is presented with Task2 -> Task1.

The loop surrounding each task will look something like this (although here I have stripped the parameters to the bare minimum, you will likely have a conditions file):



task1loop Properties

Name: task1loop

Loop type: random

Is trials: ☑

Num. repeats: $ task1Reps

Selected rows:

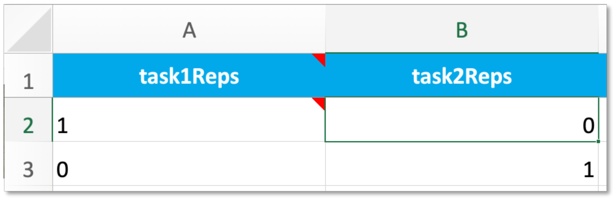
Random seed: $

Conditions:

No parameters set

Where the number of times that block is repeated (or occurs at all!) is determined by the outer loop (e.g. Task1 nReps = 'task1Reps', Task2 nReps = 'task2Reps').

For our outer loop we will use conditions files that look something like this:



	A	B
1	task1Reps	task2Reps
2	1	0
3	0	1

Each row corresponds to how many times a subtask routine (or set of routines) will be repeated per iteration of the outer loop.

What about going **online** ? Well, things are more difficult there, but not impossible let's talk about *Counterbalancing online*

5.1.8 Components

The following Components are available from Builder:

Aperture Component

This component can be used to filter the visual display, as if the subject is looking at it through an opening (i.e. add an image component, as the background image, then add an aperture to show part of the image). Currently, in builder, only circular apertures are supported (you can change the shape by specifying your aperture in a code component- we are hoping to make it easier to do this through builder soon!). Moreover, only one aperture is enabled at a time. You can't "double up": a second aperture takes precedence. Currently this component **does not run online** (see the status of online options, but you can achieve something similar online using an image with a mask: see an example demo here with corresponding PsychoPy experiment files here or by using the MouseView plugin.

Categories:
Stimuli

Works in:
PsychoPy

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name
Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start
When the Aperture Component should start, see *Defining the onset/duration of components*.

Expected start (s)
If you are using frames to control timing of your stimuli, you can add an expected start time to display the component timeline in the routine.

Start type
How do you want to define your start point?

Options:

- time (s)
- frame N
- condition

Stop
When the Aperture Component should stop, see *Defining the onset/duration of components*.

Expected duration (s)
If you are using frames to control timing of your stimuli, you can add an expected duration to display the component timeline in the routine.

Stop type
How do you want to define your end point?

Options:

- duration (s)
- duration (frames)
- time (s)
- frame N

- condition

Shape

What shape is this? With 'regular polygon…' you can set number of vertices and with 'custom polygon…' you can set vertices

Options:

- Line
- Triangle
- Rectangle
- Circle
- Cross
- Star
- Arrow
- Regular polygon…
- Custom polygon…

Num. vertices

How many vertices in your regular polygon?

Vertices

What are the vertices of your polygon? Should be an nx2 array or a list of [x, y] lists

Layout

How should the stimulus be laid out on screen? Padding, margins, size, position, etc.

Size

How big is the aperture? (a single number for diameter)

Position [x,y]

Where is the aperture centred?

Spatial units

Spatial units for this stimulus (e.g. for its *position* and *size*), see *Units for the window and stimuli* for more info.

Options:

- from exp settings
- deg
- cm
- pix
- norm
- height
- degFlatPos
- degFlat

Anchor

Which point in this stimulus should be anchored to the point specified by *Position [x,y]*?

Options:

  - center
  - top-center
  - bottom-center
  - center-left
  - center-right
  - top-left
  - top-right
  - bottom-left
  - bottom-right

Orientation
Orientation of this stimulus (in deg)

Options:

  - -360
  - 360

Draggable?
Should this stimulus be moveble by clicking and dragging?

Data

What information about this Component should be saved?

Save onset/offset times
Store the onset/offset times in the data file (as well as in the log file).

Sync timing with screen refresh
Synchronize times with screen refresh (good for visual stimuli and responses based on them)

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Component
Disable this Component

Validate with…
Name of the Validator Routine to use to check the timing of this stimulus. Options are generated live, so will vary according to your setup.

See also

API reference for *`Aperture`*

Audio Validator Routine

Use a sound sensor to confirm that audio stimuli are presented when they should be.

Categories:
Validation

Works in:
PsychoPy

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name
Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Find best threshold?
Run a brief Routine to find the best threshold for the sound sensor at experiment start?

Threshold (*if :ref:`audiovalidatorroutine-findthreshold` isn't ==True*)
Volume threshold at which the sound sensor should register a positive, units go from 0 (least volume) to 255 (most volume).

Device

Information about the device associated with this Component. Keyboards, speakers, microphones, etc.

Device name
A name to refer to this Component's associated hardware device by. If using the same device for multiple components, be sure to use the same name here.

Sound sensor type
Type of sound sensor to use.

Sound sensor channel
If relevant, a channel number attached to the sound sensor, to distinguish it from other sound sensors on the same port. Leave blank to use the first sound sensor which can detect the Window.

Microphone
What microphone device to use?

Decibel range (*if :ref:`audiovalidatorroutine-devicebackend` == 'microphone'*)
Range of possible decibels to expect mic responses to be in, by default (0, 1)

Sampling window (*if :ref:`audiovalidatorroutine-devicebackend` == 'microphone'*)
How long (s) to average samples from the microphone across? Larger sampling windows reduce the chance of random spikes, but also reduce sensitivity.

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Routine
Disable this Routine

Brush Component

A freehand drawing tool using the mouse.

Categories:
Responses

Works in:
PsychoPy, PsychoJS

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name
Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start
When the Brush Component should start, see *Defining the onset/duration of components*.

Expected start (s)
If you are using frames to control timing of your stimuli, you can add an expected start time to display the component timeline in the routine.

Start type
How do you want to define your start point?

Options:

- time (s)
- frame N
- condition

Stop
When the Brush Component should stop, see *Defining the onset/duration of components*.

Expected duration (s)
If you are using frames to control timing of your stimuli, you can add an expected duration to display the component timeline in the routine.

Stop type
How do you want to define your end point?

Options:

- duration (s)
- duration (frames)
- time (s)
- frame N
- condition

Press button
Should the participant have to press a button to paint (True), or should it be always on (False)?

Appearance

How should the stimulus look? Colors, borders, styles, etc.

Brush size
Width of the brush's line (always in pixels and limited to 10px max width)

Brush color
Fill color of this brush

Color space
In what format (color space) have you specified the colors? See *Color spaces*

Options:

- rgb
- dkl
- lms
- hsv

Opacity
Vary the transparency, from 0.0 (invisible) to 1.0 (opaque)

Contrast
Contrast of the stimulus (1.0=unchanged contrast, 0.5=decrease contrast, 0.0=uniform/no contrast, -0.5=slightly inverted, -1.0=totally inverted)

Data

What information about this Component should be saved?

Save onset/offset times
Store the onset/offset times in the data file (as well as in the log file).

Sync timing with screen refresh
Synchronize times with screen refresh (good for visual stimuli and responses based on them)

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Component
Disable this Component

Validate with…
Name of the Validator Routine to use to check the timing of this stimulus. Options are generated live, so will vary according to your setup.

See also

API reference for `Brush`

Button Box Component

Button Box: Get input from a button box

Categories:
Responses

Works in:
PsychoPy

Note: Since this is still in beta, keep an eye out for bug fixes.

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name
Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start
When the Button Box Component should start, see *Defining the onset/duration of components*.

Expected start (s)
If you are using frames to control timing of your stimuli, you can add an expected start time to display the component timeline in the routine.

Start type
How do you want to define your start point?

Options:

- time (s)
- frame N
- condition

Stop
When the Button Box Component should stop, see *Defining the onset/duration of components*.

Expected duration (s)
If you are using frames to control timing of your stimuli, you can add an expected duration to display the component timeline in the routine.

Stop type
How do you want to define your end point?

Options:

- duration (s)
- duration (frames)
- time (s)
- frame N
- condition

Force end of Routine
Should a response force the end of the Routine (e.g end the trial)?

Device

Information about the device associated with this Component. Keyboards, speakers, microphones, etc.

Device label
A label to refer to this Component's associated hardware device by. If using the same device for multiple components, be sure to use the same label here.

Device backend
What kind of button box is it? What package/plugin should be used to talk to it?

Buttons
Keys to treat as buttons (in order of what button index you want them to be). Must be the same length as the number of buttons.

Data

What information about this Component should be saved?

Register button press on…
When should the button press be registered? As soon as pressed, or when released?

Options:

- Press
- Release

Store
Choose which (if any) responses to store at the end of a trial

Options:

- Last button
- First button
- All buttons
- Nothing

Allowed buttons
A comma-separated list of button indices (should be whole numbers), leave blank to listen for all buttons.

Store correct
Do you want to save the response as correct/incorrect?

Correct answer
What is the 'correct' key? Might be helpful to add a correctAns column and use $correctAns to compare to the key press.

Save onset/offset times
Store the onset/offset times in the data file (as well as in the log file).

Sync timing with screen refresh
Synchronize times with screen refresh (good for visual stimuli and responses based on them)

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Component
Disable this Component

Button Component

This component allows you to show a static textbox which ends the routine and/or triggers a "callback" (some custom code) when pressed. The nice thing about the button component is that you can allow mouse/touch responses with a single component instead of needing 3 separate components i.e. a textbox component (to display as a "clickable" thing), a mouse component (to click the textbox) and a code component (not essential, but for example to check if a clicked response was correct or incorrect).

Categories:
Responses

Works in:
PsychoPy, PsychoJS

Note: Since this is still in beta, keep an eye out for bug fixes.

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name
Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start
When the Button Component should start, see *Defining the onset/duration of components*.

Expected start (s)
If you are using frames to control timing of your stimuli, you can add an expected start time to display the component timeline in the routine.

Start type
How do you want to define your start point?

Options:

- time (s)
- frame N
- condition

Stop
When the Button Component should stop, see *Defining the onset/duration of components*.

Expected duration (s)
If you are using frames to control timing of your stimuli, you can add an expected duration to display the component timeline in the routine.

Stop type
How do you want to define your end point?

Options:

- duration (s)
- duration (frames)
- time (s)
- frame N
- condition

Force end of Routine
Should a response force the end of the Routine (e.g end the trial)?

Button text
The text to be displayed

Callback function
Code to run when button is clicked

Run once per click

Should the callback run once per click (True), or each frame until click is released (False)

Layout

How should the stimulus be laid out on screen? Padding, margins, size, position, etc.

Size [w,h]

Size of this stimulus (either a single value or x,y pair, e.g. 2.5, [1,2]

Position [x,y]

Position of this stimulus (e.g. [1,2])

Spatial units

Spatial units for this stimulus (e.g. for its *position* and *size*), see *Units for the window and stimuli* for more info.

Options:

- from exp settings
- deg
- cm
- pix
- norm
- height
- degFlatPos
- degFlat

Anchor

Which point in this stimulus should be anchored to the point specified by *Position [x,y]*?

Options:

- center
- top-center
- bottom-center
- center-left
- center-right
- top-left
- top-right
- bottom-left
- bottom-right

Orientation

Orientation of this stimulus (in deg)

Options:

- -360
- 360

Padding

Defines the space between text and the textbox border

Appearance

How should the stimulus look? Colors, borders, styles, etc.

Text color
Foreground color of this stimulus (e.g. $[1,1,0], red)

Fill color
Fill color of this stimulus (e.g. $[1,1,0], red)

Border color
Border color of this stimulus (e.g. $[1,1,0], red)

Color space
In what format (color space) have you specified the colors? See *Color spaces* for more info.

Options:

- rgb
- dkl
- lms
- hsv

Opacity
Vary the transparency, from 0.0 (invisible) to 1.0 (opaque)

Border width
How wide should the textbox outline be? Width is specified in chosen spatial units, see _units

Contrast
Contrast of the stimulus (1.0=unchanged contrast, 0.5=decrease contrast, 0.0=uniform/no contrast, -0.5=slightly inverted, -1.0=totally inverted)

Formatting

How should this stimulus handle text? Font, spacing, orientation, etc.

Font
What font should the text be displayed in? Locally, can be a font installed on your computer, saved to the "fonts" folder in your user folder, or the name of a Google Font. Online, can be any web safe font or a font file added to your resources list in *Experiment settings*.

Letter height
Specifies the height of the letter (the width is then determined by the font)

Bold
Should text be bold?

Italic
Should text be italic?

Data

What information about this Component should be saved?

Save onset/offset times
Store the onset/offset times in the data file (as well as in the log file).

Sync timing with screen refresh
Synchronize times with screen refresh (good for visual stimuli and responses based on them)

Record clicks

What clicks on this button should be saved to the data output?

Options:

- first click
- last click
- every click
- none

Time relative to

What should the values of mouse.time should be relative to?

Options:

- button onset
- experiment
- routine

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Component

Disable this Component

Validate with…

Name of the Validator Routine to use to check the timing of this stimulus. Options are generated live, so will vary according to your setup.

See also

API reference for `ButtonStim`

Camera Component

This component provides a way to use the webcam to record participants during an experiment.

Note: For online experiments, the browser will notify participants to allow use of webcam before the start of the task.

When recording via webcam, specify the starting time relative to the start of the routine (see *start* below) and a stop time (= duration in seconds). A blank duration evaluates to recording for 0.000s.

The resulting video files are saved in .mp4 format if recorded locally and saved in .webm if recorded online. There will be one file per recording. The files appear in a new folder within the data directory in a folder called data_cam_recorded. The file names include the unix (epoch) time of the onset of the recording with milliseconds, e.g., *recording_cam_2022-06-16_14h32.42.064.mp4*.

Note: For online experiments, the recordings can only be downloaded from the "Download results" button from the study's Pavlovia page.

For a demo in builder mode, after unpacking the demos, click on Demos > Feature Demos > camera. For a demo in coder mode, click on Demos > hardware > camera.py

Categories:

Responses

Works in:
PsychoPy, PsychoJS

Note: Since this is still in beta, keep an eye out for bug fixes.

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name
Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start
When the Camera Component should start, see *Defining the onset/duration of components*.

Expected start (s)
If you are using frames to control timing of your stimuli, you can add an expected start time to display the component timeline in the routine.

Start type
How do you want to define your start point?

Options:

- time (s)
- frame N
- condition

Stop
When the Camera Component should stop, see *Defining the onset/duration of components*.

Expected duration (s)
If you are using frames to control timing of your stimuli, you can add an expected duration to display the component timeline in the routine.

Stop type
How do you want to define your end point?

Options:

- duration (s)
- duration (frames)
- time (s)
- frame N
- condition

Device

Information about the device associated with this Component. Keyboards, speakers, microphones, etc.

Device label
A label to refer to this Component's associated hardware device by. If using the same device for multiple components, be sure to use the same label here.

Backend

Python package to use behind the scenes.

Options:

- FFPyPlayer
- OpenCV

Video device

What device would you like to use to record video? This will only affect local experiments - online experiments ask the participant which device to use.

Video device

What device would you like to use to record video? This will only affect local experiments - online experiments ask the participant which device to use.

Resolution

Resolution (w x h) to record to, leave blank to use device default.

Resolution

Resolution (w x h) to record to, leave blank to use device default.

Frame rate

Frame rate (frames per second) to record at, leave blank to use device default.

Frame rate

Frame rate (frames per second) to record at, leave blank to use device default. For some cameras, you may need to use *camera.CAMERA_FRAMERATE_NTSC* or *camera.CAMERA_FRAMERATE_NTSC / 2*.

Audio

Microphone device label

A label to refer to this Component's associated microphone device by. If using the same device for multiple components, be sure to use the same label here.

Microphone

What microphone device would you like the use to record? This will only affect local experiments - online experiments ask the participant which mic to use. Options are generated live, so will vary according to your setup.

Channels

Record two channels (stereo) or one (mono, smaller file). Select 'auto' to use as many channels as the selected device allows.

Options:

- auto
- mono
- stereo

Sample rate (hz)

How many samples per second (Hz) to record at

Max recording size (kb)

To avoid excessively large output files, what is the biggest file size you are likely to expect?

Data

What information about this Component should be saved?

Save onset/offset times
Store the onset/offset times in the data file (as well as in the log file).

Sync timing with screen refresh
Synchronize times with screen refresh (good for visual stimuli and responses based on them)

Save file?
Save webcam output to a file?

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Component
Disable this Component

Code Component

This Component can be used to insert short pieces of python code into your experiments. This might be create a variable that you want for another *Component*, to manipulate images before displaying them, to interact with hardware for which there isn't yet a pre-packaged component in (e.g. writing code to interact with the serial/parallel ports). See *code uses* below.

Be aware that the code for each of the components in your *Routine* are executed in the order they appear on the *Routine* display (from top to bottom). If you want your *Code Component* to alter a variable to be used by another component immediately, then it needs to be above that component in the view. You may want the code not to take effect until next frame however, in which case put it at the bottom of the *Routine*. You can move *Components* up and down the *Routine* by right-clicking on their icons.

Within your code you can use other variables and modules from the script. For example, all routines have a stopwatch-style `Clock` associated with them, which gets reset at the beginning of that repeat of the routine. So if you have a *Routine* called trial, there will be a `Clock` called trialClock and so you can get the time (in sec) from the beginning of the trial by using:

```
currentT = trialClock.getTime()
```

To see what other variables you might want to use, and also what terms you need to avoid in your chunks of code, *compile your script* before inserting the code object and take a look at the contents of that script.

Note that this page is concerned with *Code Components* specifically, and not all cases in which you might use python syntax within the Builder. It is also possible to put code into a non-code input field (such as the duration or text of a *Text Component*). The syntax there is slightly different (requiring a *$* to trigger the special handling, or *\$* to avoid triggering special handling). The syntax to use within a Code Component is always regular python syntax.

Parameters

Code type:
What type of code will you write?

- *Py* - Python code only (for local use)
- *JS* - Javascript only (for online use)
- *Auto -> JS* - Write in python code on the left and this will be auto translated to Javascript on the right.

- *Both* - write both Python and Javascript, but independently of one another (Python will be executed when you run the task locally, JS will be executed when you run the task online)

Within a *Code Component* you can write code to be executed at 6 different points within the experiment. You can use as many or as few of these as you need for any *Code Component*:

Before Experiment:
Things that need to be done just once, like importing a supporting module, which do not need the experiment window to exist yet.

Begin Experiment:
Things that need to be done just once, like initialising a variable for later use, which may need to refer to the experiment window.

Begin Routine:
Certain things might need to be done at the start of a *Routine* e.g. at the beginning of each trial you might decide which side a stimulus will appear.

Each Frame:
Things that need to updated constantly, throughout the experiment. Note that these will be executed exactly once per video frame (on the order of every 10ms), to give dynamic displays. Static displays do not need to be updated every frame.

End Routine:
At the end of the *Routine* (e.g. the trial) you may need to do additional things, like checking if the participant got the right answer

End Experiment:
Use this for things like saving data to disk, presenting a graph(?), or resetting hardware to its original state.

Example code uses

1. Set a random location for your target stimulus

There are many ways to do this, but you could add the following to the *Begin Routine* section of a *Code Component* at the top of your *Routine*. Then set your stimulus position to be *$(targetX, 0)* and set the correct answer field of a *Keyboard Component* to be *$corrAns* (set both of these to update on every repeat of the Routine).:

```
if random()>0.5:
    targetX=-0.5 #on the left
    corrAns='left'
else:
    targetX=0.5#on the right
    corrAns='right'
```

2. Create a patch of noise

As with the above there are many different ways to create noise, but a simple method would be to add the following to the *Begin Routine* section of a *Code Component* at the top of your *Routine*. Then set the image as *$noiseTexture*.:

```
noiseTexture = random.rand((128,128)) * 2.0 - 1
```

Note

Don't expect all code components to work online. Remember that code components using specific python libraries such as numpy won't smoothly translate. You might want to view the PsychoPy to Javascript crib sheet for useful info on using code components for online experiments.

3. Send a feedback message at the end of the experiment

Make a new routine, and place it at the end of the flow (i.e., the end of the experiment). Create a *Code Component* with this in the *Begin Experiment* field:

```
expClock = core.Clock()
```

and put this in the *Begin routine* field:

```
msg = "Thanks for participating - that took' + str(expClock.getTime()/60.0)) + 'minutes␣
↪in total'
```

Next, add a *Text Component* to the routine, and set the text to *$msg*. Be sure that the text field's updating is set to "Set every repeat" (and not "Constant").

4. End a loop early.

Code components can also be used to control the end of a loop. For example imagine you want to end when a key response has been made 5 times:

```
if key_resp.keys: # if a key response has been made
    if len(key_resp.keys) ==5: # if 5 key presses have been made
        continueRoutine = False # end the current routine
        trials.finished = True # exit the current loop (if your loop is called "trials"
```

What variables are available to use?

The most complete way to find this out for your particular script is to *compile it* and take a look at what's in there. Below are some options that appear in nearly all scripts. Remember that those variables are Python objects and can have attributes of their own. You can find out about those attributes using:

```
dir(myObject)
```

Common variables:

- expInfo: This is a Python Dictionary containing the information from the starting dialog box. e.g. That generally includes the 'participant' identifier. You can access that in your experiment using *exp['participant']*
- t: the current time (in seconds) measured from the start of this Routine
- frameN: the number of /completed/ frames since the start of the Routine (=0 in the first frame)
- win: the `Window` that the experiment is using

Your own variables:

- anything you've created in a Code Component is available for the rest of the script. (Sometimes you might need to define it at the beginning of the experiment, so that it will be available throughout.)
- the name of any other stimulus or the parameters from your file also exist as variables.
- most Components have a *status* attribute, which is useful to determine whether a stimulus has *NOT_STARTED*, *STARTED* or *FINISHED*. For example, to play a tone at the end of a Movie Component (of unknown duration) you could set start of your tone to have the 'condition'

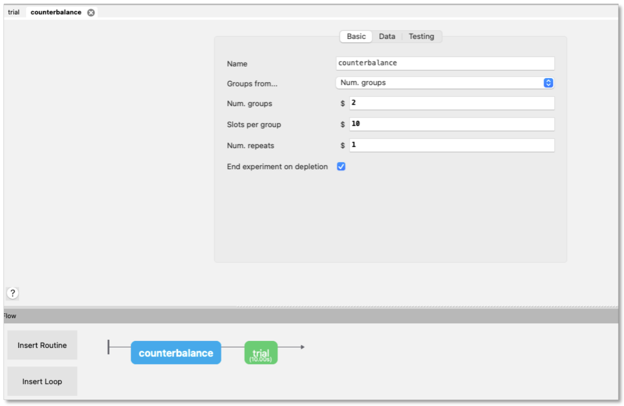
```
myMovieName.status==FINISHED
```

Selected contents of the numpy library and numpy.random are imported by default. The entire numpy library is imported as *np*, so you can use a several hundred maths functions by prepending things with 'np.':

- random() , randint() , normal() , shuffle() options for creating arrays of random numbers.
- *sin()*, *cos()*, *tan()*, and *pi*: For geometry and trig. By default angles are in radians, if you want the cosine of an angle specified in degrees use *cos(angle*180/pi)*, or use numpy's conversion functions, *rad2deg(angle)* and *deg2rad(angle)*.
- linspace(): Create an array of linearly spaced values.
- *log()*, *log10()*: The natural and base-10 log functions, respectively. (It is a lowercase-L in log).
- *sum()*, *len()*: For the sum and length of a list or array. To find an average, it is better to use *average()* (due to the potential for integer division issues with *sum()/len()*).
- *average()*, *sqrt()*, *std()*: For average (mean), square root, and standard deviation, respectively. **Note:** Be sure that the numpy standard deviation formula is the one you want!
- np.______: Many math-related features are available through the complete numpy libraries, which are available within psychopy builder scripts as 'np.'. For example, you could use *np.hanning(3)* or *np.random.poisson(10, 10)* in a code component.

Counterbalance Routine

The counterbalance standalone routine is available to use locally and online. This component allows you to automatically assign participants to groups based on defined number of groups and how many participants you want per group (slots). You can find the component in the "Custom" section of Builder. Once you add the component, do remember to select "Insert Routine" > name of your counterbalance routine and insert it into your flow!



Name: counterbalance

Groups from...: Num. groups

Num. groups: $ 2

Slots per group: $ 10

Num. repeats: $ 1

End experiment on depletion ☑

Categories:
Custom

Works in:
PsychoPy, PsychoJS

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name
Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Groups from…
Specify groups using an Excel file (for fine tuned control), specify as a variable name, or specify a number of groups to create equally likely groups with a uniform cap.

Options:

- Num. groups: Specify the number of groups and what their caps are. However, this cap is the same as for every group. The groups and caps you use here are only *reflected for local use*. For **Pavlovia**, you would need to set the number of groups and their caps via Shelf. Click here for an example on how to do that.
- Conditions file (local only): Allows maximum flexibility in setting up groups. By using an excel spreadsheet, the probability of each group occuring, slots per group and any other additional parameters can be speficied. *Note: This is currently not supported for online studies*
- Variable (local only): Similar to "Conditions file", but using a variable name rather than a file name (the variable should contain the same kind of information as would come from reading a conditions file via e.g. *pandas.read_csv <https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html>*_)

Conditions (*if :ref:`_counterbalanceroutine-specMode` is "Conditions file"*)
Name of a file specifying the parameters for each group (.csv, .xlsx, or .pkl). Browse to select a file. Right-click to preview file contents, or create a new file.

Num. groups (*if :ref:`_counterbalanceroutine-specMode` is "Num. groups"*)
Number of groups to use.

Slots per group (*if :ref:`_counterbalanceroutine-specMode` is "Num. groups"*)
Max number of participants in each group for each repeat.

Num. repeats (*if :ref:`_counterbalanceroutine-specMode` is "Num. groups"*)
How many times to run slots down to depletion?

End experiment on depletion
When all slots and repetitions are depleted, should the experiment end or continue with .finished on this Routine as True?

Conditions (*if :ref:`_counterbalanceroutine-specMode` is "Variable"*)
Name of a variable specifying the parameters for each group. Should be a list of dicts, like the output of data.conditionsFromFile

Data

What information about this Component should be saved?

Save data
Save chosen group and associated params this repeat to the data file

Save remaining cap
Save the remaining cap for the chosen group this repeat to the data file

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Routine
Disable this Routine

Dots Component

The Dots Component allows you to present a Random Dot Kinematogram (RDK) to the participant of your study. Note that this component is **not yet supported for online use** (see status of online options) but users have contributed work arounds for use online. These are fields of dots that drift in different directions and subjects are typically required to identify the 'global motion' of the field.

There are many ways to define the motion of the signal and noise dots. In the way the dots are configured follows Scase, Braddick & Raymond (1996). Although Scase et al (1996) show that the choice of algorithm for your dots actually makes relatively little difference there are some **potential** gotchas. Think carefully about whether each of these will affect your particular case:

- **limited dot lifetimes:** as your dots drift in one direction they go off the edge of the stimulus and are replaced randomly in the stimulus field. This could lead to a higher density of dots in the direction of motion providing subjects with an alternative cue to direction. Keeping dot lives relatively short prevents this.
- **noiseDots='direction':** some groups have used noise dots that appear in a random location on each frame (noiseDots='location'). This has the disadvantage that the noise dots not only have a random direction but also a random speed (whereas signal dots have a constant speed and constant direction)
- **signalDots='same':** on each frame the dots constituting the signal could be the same as on the previous frame or different. If 'different', participants could follow a single dot for a long time and calculate its average direction of motion to get the 'global' direction, because the dots would sometimes take a random direction and sometimes take the signal direction.

As a result of these, the defaults for are to have signalDots that are from a 'different' population, noise dots that have random 'direction' and a dot life of 3 frames.

Categories:
Stimuli

Works in:
PsychoPy

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name
Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start
When the Dots Component should start, see *Defining the onset/duration of components*.

Expected start (s)
If you are using frames to control timing of your stimuli, you can add an expected start time to display the component timeline in the routine.

Start type
How do you want to define your start point?

Options:

- time (s)
- frame N
- condition

Stop

When the Dots Component should stop, see *Defining the onset/duration of components*.

Expected duration (s)

If you are using frames to control timing of your stimuli, you can add an expected duration to display the component timeline in the routine.

Stop type

How do you want to define your end point?

Options:

- duration (s)
- duration (frames)
- time (s)
- frame N
- condition

Layout

How should the stimulus be laid out on screen? Padding, margins, size, position, etc.

Dot size

Size of the dots in pixel units.

Field size

A single value, specifying the diameter of the field (in the specified Spatial Units). Sizes can be negative and can extend beyond the window.

Field position

Where is the field centred (in the specified units)?

Spatial units

Spatial units for this stimulus (e.g. for its position and size), see *Units for the window and stimuli* for more info.

Options:

- from exp settings
- deg
- cm
- pix
- norm
- height
- degFlatPos
- degFlat

Field anchor

Which point in this field should be anchored to the point specified by dotscomponent-pos?

Options:

- center
- top-center
- bottom-center

- center-left
- center-right
- top-left
- top-right
- bottom-left
- bottom-right

Field shape
Defines the shape of the field in which the dots appear.

Options:

- circle
- square

Appearance

How should the stimulus look? Colors, borders, styles, etc.

Dot color
Color of the dots.

Dot color space
In what format (color space) have you specified the colors? See *Color spaces* for more info.

Options:

- rgb
- dkl
- lms
- hsv

Opacity
Vary the transparency, from 0.0 (invisible) to 1.0 (opaque)

Contrast
Contrast of the stimulus (1.0=unchanged contrast, 0.5=decrease contrast, 0.0=uniform/no contrast, -0.5=slightly inverted, -1.0=totally inverted)

Dots

Number of dots
Number of dots in the field (for circular fields this will be average number of dots)

Direction
Direction of motion for the signal dots (degrees)

Speed
Speed of the dots (displacement per frame in the specified units)

Coherence
Coherence of the dots (fraction moving in the signal direction on any one frame)

Dot life-time
Number of frames before each dot is killed and randomly assigned a new position

Signal dots

If 'same' then the signal and noise dots are constant. If different then the choice of which is signal and which is noise gets randomised on each frame. This corresponds to Scase et al's (1996) categories of RDK.

Options:

- same
- different

Dot refresh rule

When should the whole sample of dots be refreshed

Options:

- none
- repeat

Noise dots

Determines the behaviour of the noise dots, taken directly from Scase et al's (1996) categories. For 'position', noise dots take a random position every frame. For 'direction' noise dots follow a random, but constant direction. For 'walk' noise dots vary their direction every frame, but keep a constant speed.

Options:

- direction
- position
- walk

Data

What information about this Component should be saved?

Save onset/offset times

Store the onset/offset times in the data file (as well as in the log file).

Sync timing with screen refresh

Synchronize times with screen refresh (good for visual stimuli and responses based on them)

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Component

Disable this Component

Validate with…

Name of the Validator Routine to use to check the timing of this stimulus. Options are generated live, so will vary according to your setup.

See also

API reference for *DotStim*

Eyetracker Calibration Routine

Calibration routine for eyetrackers

Categories:
Eyetracking

Works in:
PsychoPy

Note: Since this is still in beta, keep an eye out for bug fixes.

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name
Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Target layout
Pre-defined target layouts

Options:

- THREE_POINTS
- FIVE_POINTS
- NINE_POINTS
- THIRTEEN_POINTS

Randomise target positions
Should the order of target positions be randomised?

Text color
Text foreground color

Target

Parameters of the calibration target.

Use custom?
Check this box to use a custom stimulus as a calibration target, rather than creating one from params.

Custom target (*if :ref:`_eyetrackercalibrationroutine-useCustom` is not checked*)
Give the name of any visual Component to use it as a calibration target.

Outer fill color (*if :ref:`_eyetrackercalibrationroutine-useCustom` is not checked*)
Fill color of the outer part of the target

Outer border color (*if :ref:`_eyetrackercalibrationroutine-useCustom` is not checked*)
Border color of the outer part of the target

Inner fill color (*if :ref:`_eyetrackercalibrationroutine-useCustom` is not checked*)
Fill color of the inner part of the target

Inner border color (*if :ref:`_eyetrackercalibrationroutine-useCustom` is not checked*)
Border color of the inner part of the target

Color space (*if :ref:`_eyetrackercalibrationroutine-useCustom` is not checked*)
In what format (color space) have you specified the colors? See *Color spaces* for more info.

Options:

- rgb
- dkl
- lms
- hsv

Outer border width (*if :ref:`_eyetrackercalibrationroutine-useCustom` is not checked*)
Width of the line around the outer part of the target

Inner border width (*if :ref:`_eyetrackercalibrationroutine-useCustom` is not checked*)
Width of the line around the inner part of the target

Outer radius (*if :ref:`_eyetrackercalibrationroutine-useCustom` is not checked*)
Size (radius) of the outer part of the target

Inner radius (*if :ref:`_eyetrackercalibrationroutine-useCustom` is not checked*)
Size (radius) of the inner part of the target

Spatial units (*if :ref:`_eyetrackercalibrationroutine-useCustom` is not checked*)
Spatial units for the target (e.g. for its position and size), see *Units for the window and stimuli* for more info.

Options:

- from exp settings

Animation

Control animations within the calibration routine.

Progress mode
Should the target move to the next position after a keypress or after an amount of time?

Options:

- space key
- time

Target duration
Time limit (s) after which progress to next position

Expand / contract duration
Duration of the target expand/contract animation

Expand scale
How many times bigger than its size the target grows

Animate position changes
Enable / disable animations as target stim changes position

Movement duration
Duration of the animation during position changes.

Target delay
Duration of the delay between positions.

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Routine
Disable this Routine

> **See also**
>
> API reference for `EyetrackerCalibration`

Eyetracker Record Component

The eye-tracker record component provides a way to record eye movement data within an experiment. To do so, specify the starting time relative to the start of the routine (see *start* below) and a stop time (= duration in seconds). Before using the eye-tracking record component, you must specify your eye tracking device under *experiment settings* > *Eyetracking*. Here the available options are:

- GazePoint
- MouseGaze
- SR Research Ltd (aka EyeLink)
- Tobii Technology

If you are developing your eye-tracking paradigm out-of-lab we recommend using *MouseGaze* which will simulate eye movement responses through monitoring your mouse cursor and buttons to simulate movements and blinks.

The resulting eye-movement coordinates are stored and accessible through calling *etRecord.pos* where *etRecord* corresponds to the name of the eye-tracking record component, you can set something (e.g. a polygon) to be in the same location as the current "look" by setting the position field to `etRecord.pos` and setting the field to update on **every frame** When running an eye tracking study, you can optionally save the data in hdf5 format through selecting this option in the experiment settings > data tab.

Categories:
Eyetracking

Works in:
PsychoPy

Note: Since this is still in beta, keep an eye out for bug fixes.

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name
Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Record actions
Should this Component start and / or stop eye tracker recording?

Options:

- Start and Stop
- Start Only

- Stop Only

Start

When the Eyetracker Record Component should start, see *Defining the onset/duration of components*.

Expected start (s)

If you are using frames to control timing of your stimuli, you can add an expected start time to display the component timeline in the routine.

Start type

How do you want to define your start point?

Options:

- time (s)
- frame N
- condition

Stop

When the Eyetracker Record Component should stop, see *Defining the onset/duration of components*.

Expected duration (s)

If you are using frames to control timing of your stimuli, you can add an expected duration to display the component timeline in the routine.

Stop type

How do you want to define your end point?

Options:

- duration (s)
- duration (frames)
- time (s)
- frame N
- condition

Stop with Routine?

Should eyetracking stop when the Routine ends? Tick to force stopping after the Routine has finished.

Data

What information about this Component should be saved?

Save onset/offset times

Store the onset/offset times in the data file (as well as in the log file).

Sync timing with screen refresh

Synchronize times with screen refresh (good for visual stimuli and responses based on them)

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Component

Disable this Component

See also

API reference for *EyeTracker*

Eyetracker Validation Routine

Validation routine for eyetrackers

Categories:
Eyetracking

Works in:
PsychoPy

Note: Since this is still in beta, keep an eye out for bug fixes.

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name
Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Target layout
How many targets do you want to be presented for calibration? Points will be displayed in a grid.

Options:

- THREE_POINTS
- FIVE_POINTS
- NINE_POINTS
- THIRTEEN_POINTS
- SEVENTEEN_POINTS
- CUSTOM…

Target positions
List of positions (x, y) at which the target can appear

Randomise target positions
Should the order of target positions be randomised?

Gaze cursor color
Fill color of the gaze cursor

Text color
Color of text used in validation procedure.

Target

Parameters of the validation target.

Outer fill color
Fill color of the outer part of the target

Outer border color
Border color of the outer part of the target

Inner fill color
Fill color of the inner part of the target

Inner border color
Border color of the inner part of the target

Color space
In what format (color space) have you specified the colors? See *Color spaces* for more info.

Options:

- rgb
- dkl
- lms
- hsv

Outer border width
Width of the line around the outer part of the target

Inner border width
Width of the line around the inner part of the target

Outer radius
Size (radius) of the outer part of the target

Inner radius
Size (radius) of the inner part of the target

Spatial units
Spatial units for the target (e.g. for its position and size), see *Units for the window and stimuli* for more info.

Options:

- from exp settings

Animation

Progress mode
Should the target move to the next position after a keypress or after an amount of time?

Options:

- space key
- time

Target duration
Time limit (s) after which progress to next position

Expand / contract duration
Duration of the target expand/contract animation

Expand scale
How many times bigger than its size the target grows

Animate position changes
Enable / disable animations as target stim changes position

Movement duration
: Duration of the animation during position changes.

Target delay
: Duration of the delay between positions.

Data

What information about this Component should be saved?

Save as image
: Save results as an image

Show results screen
: Show a screen with results after completion?

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Routine
: Disable this Routine

> **See also**
>
> API reference for `EyetrackerCalibration`

Form Component

The Form component enables Psychopy to be used as a questionnaire tool, where participants can be presented with a series of questions requiring responses. Form items, defined as questions and response pairs, are presented simultaneously onscreen with a scrollable viewing window.

Note: We have now introduced Pavlovia Surveys which allow you to create online questionnaires. You can either use them by themselves or in conjunction with your experiments. Click here to watch our Pavlovia Surveys Launch Webinar to find out more.

Categories:
: Responses

Works in:
: PsychoPy, PsychoJS

Note: Since this is still in beta, keep an eye out for bug fixes.

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name
: Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start
: When the Form Component should start, see *Defining the onset/duration of components*.

Expected start (s)

If you are using frames to control timing of your stimuli, you can add an expected start time to display the component timeline in the routine.

Start type

How do you want to define your start point?

Options:

- time (s)
- frame N
- condition

Stop

When the Form Component should stop, see *Defining the onset/duration of components*.

Expected duration (s)

If you are using frames to control timing of your stimuli, you can add an expected duration to display the component timeline in the routine.

Stop type

How do you want to define your end point?

Options:

- duration (s)
- duration (frames)
- time (s)
- frame N
- condition

Items

A csv / xlsx file **To get started, we recommend selecting the "Open/Create Icon" which will open up a template forms spreadsheet**.

A csv/xlsx file should have the following key, value pairs / column headers:

index

The item index as a number

itemText

The item question string

itemWidth

The question width between 0 : 1

type

The type of rating e.g., 'choice', 'rating', 'slider', 'free text'

responseWidth

The question width between 0 : 1

options

A sequence of tick labels for options e.g., yes, no

layout

Response object layout e.g., 'horiz' or 'vert'

itemColor

The question text font color

responseColor
The response object color

granularity
If you are using a slider, what do you want the granularity of the slider to be?

Missing column headers will be replaced by default entries, with the exception of *itemText* and *type*, which are required. The default entries are:

index
0 (increments for each item)

itemWidth
0.7

responseWidth
0.3

options
Yes, No

layout
horiz

itemColor
from style

responseColor
from style

Randomize
Do you want to randomize the order of your questions?

Data format
Store item data by columns, or rows

Options:

- columns
- rows

Layout

How should the stimulus be laid out on screen? Padding, margins, size, position, etc.

Size [w,h]
Size of this stimulus (either a single value or x,y pair, e.g. 2.5, [1,2]

Position [x,y]
Position of this stimulus (e.g. [1,2])

Item padding
The padding or space between items.

Appearance

How should the stimulus look? Colors, borders, styles, etc.

Fill color (*if :ref:`formcomponent-Style` is "Custom…"*)
Color of the form's background

Border color (*if :ref:`formcomponent-Style` is “Custom…”*)

Color of the outline around the form

Item color (*if :ref:`formcomponent-Style` is “Custom…”*)

Base text color for questions

Response color (*if :ref:`formcomponent-Style` is “Custom…”*)

Base text color for responses, also sets color of lines in sliders and borders of textboxes

Marker color (*if :ref:`formcomponent-Style` is “Custom…”*)

Color of markers and the scrollbar

Styles

Styles determine the appearance of the form

Options:

- light
- dark
- custom…

Color space

In what format (color space) have you specified the colors? See *Color spaces* for more info.

Options:

- rgb
- dkl
- lms
- hsv

Opacity

Vary the transparency, from 0.0 (invisible) to 1.0 (opaque)

Contrast

Contrast of the stimulus (1.0=unchanged contrast, 0.5=decrease contrast, 0.0=uniform/no contrast, -0.5=slightly inverted, -1.0=totally inverted)

Formatting

How should this stimulus handle text? Font, spacing, orientation, etc.

Text height

The size of the item text for Form

Font

What font should the text be displayed in? Locally, can be a font installed on your computer, saved to the “fonts” folder in your user folder, or the name of a Google Font. Online, can be any web safe font or a font file added to your resources list in *Experiment settings*.

Data

What information about this Component should be saved?

Save onset/offset times

Store the onset/offset times in the data file (as well as in the log file).

Sync timing with screen refresh

Synchronize times with screen refresh (good for visual stimuli and responses based on them)

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Component
Disable this Component

Validate with…
Name of the Validator Routine to use to check the timing of this stimulus. Options are generated live, so will vary according to your setup.

Note

Top tip: Form has an attribute to check if all questions have been answered `form.complete`. You could use this to make a "submit" button appear only when the form is completed!

See also

API reference for `Form`

Grating Component

The Grating stimulus allows a texture to be wrapped/cycled in 2 dimensions, optionally in conjunction with a mask (e.g. Gaussian window). The texture can be a bitmap image from a variety of standard file formats, or a synthetic texture such as a sinusoidal grating. The mask can also be derived from either an image, or mathematical form such as a Gaussian.

When using gratings, if you want to use the *spatial frequency* setting then create just a single cycle of your texture and allow to handle the repetition of that texture (do not create the cycles you're expecting within the texture).

Gratings can have their position, orientation, size and other settings manipulated on a frame-by-frame basis. There is a performance advantage (in terms of milliseconds) to using images which are square and powers of two (32, 64, 128, etc.), however this is slight and would not be noticed in the majority of experiments.

Categories:
Stimuli

Works in:
PsychoPy, PsychoJS

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name
Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start
When the Grating Component should start, see *Defining the onset/duration of components*.

Expected start (s)
If you are using frames to control timing of your stimuli, you can add an expected start time to display the component timeline in the routine.

Start type

How do you want to define your start point?

Options:

- time (s)
- frame N
- condition

Stop

When the Grating Component should stop, see *Defining the onset/duration of components*.

Expected duration (s)

If you are using frames to control timing of your stimuli, you can add an expected duration to display the component timeline in the routine.

Stop type

How do you want to define your end point?

Options:

- duration (s)
- duration (frames)
- time (s)
- frame N
- condition

Layout

How should the stimulus be laid out on screen? Padding, margins, size, position, etc.

Size [w,h]

Size of this stimulus (either a single value or x,y pair, e.g. 2.5, [1,2]). If the mask is a Gaussian then the size refers to width at 3 standard deviations on either side of the mean (i.e. sd=size/6)

Position [x,y]

Position of this stimulus (e.g. [1,2])

Spatial units

Spatial units for this stimulus (e.g. for its *position* and *size*), see *Units for the window and stimuli* for more info.

Options:

- from exp settings
- deg
- cm
- pix
- norm
- height
- degFlatPos
- degFlat

Anchor

Which point in this stimulus should be anchored to the point specified by *Position [x,y]*?

Options:

- center
- top-center
- bottom-center
- center-left
- center-right
- top-left
- top-right
- bottom-left
- bottom-right

Orientation

The orientation of the entire patch (texture and mask) in degrees.

Draggable?

Should this stimulus be moveble by clicking and dragging?

Appearance

How should the stimulus look? Colors, borders, styles, etc.

Foreground color

Foreground color of this stimulus (e.g. $[1,1,0], red)

Color space

In what format (color space) have you specified the colors? See *Color spaces* for more info.

Options:

- rgb
- dkl
- lms
- hsv

Opacity

Vary the transparency, from 0.0 (invisible) to 1.0 (opaque)

Contrast

Contrast of the stimulus (1.0=unchanged contrast, 0.5=decrease contrast, 0.0=uniform/no contrast, -0.5=slightly inverted, -1.0=totally inverted)

OpenGL blend mode

OpenGL Blendmode: avg gives traditional transparency, add is important to combine gratings)]

Options:

- avg
- add

Texture

Texture

A filename, a standard name (sin, sqr) or a variable giving a numpy array specifying the image that will be used as the *texture* for the visual patch. The image can be repeated on the patch (in either x or y or both) by setting the spatial frequency to be high (or can be stretched so that only a subset of the image appears by setting the spatial frequency to be low). Filenames can be relative or absolute paths and can refer to most image formats (e.g. tif, jpg, bmp, png, etc.). If this is set to none, the patch will be a flat colour.

Mask

The mask can define the shape (e.g. circle will make the patch circular) or something which overlays the patch e.g. noise.

Options:

- gauss
- circle

Phase (in cycles)

The position of the texture within the mask, in both X and Y. If a single value is given it will be applied to both dimensions. The phase has units of cycles (rather than degrees or radians), wrapping at 1. As a result, setting the phase to 0,1,2… is equivalent, causing the texture to be centered on the mask. A phase of 0.25 will cause the image to shift by half a cycle (equivalent to pi radians). The advantage of this is that is if you set the phase according to time it is automatically in Hz.

Spatial frequency

The spatial frequency of the texture on the patch. The units are dependent on the specified units for the stimulus/window; if the units are *deg* then the SF units will be *cycles/deg*, if units are *norm* then the SF units will be cycles per stimulus. If this is set to none then only one cycle will be displayed.

Texture resolution

Defines the size of the resolution of the texture for standard textures such as *sin*, *sqr* etc. For most cases a value of 256 pixels will suffice, but if stimuli are going to be very small then a lower resolution will use less memory.

Options:

- 32
- 64
- 128
- 256
- 512

Interpolate

If *linear* is selected then linear interpolation will be applied when the image is rescaled to the appropriate size for the screen. *Nearest* will use a nearest-neighbour rule.

Options:

- linear
- nearest

Data

What information about this Component should be saved?

Save onset/offset times

Store the onset/offset times in the data file (as well as in the log file).

Sync timing with screen refresh
Synchronize times with screen refresh (good for visual stimuli and responses based on them)

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Component
Disable this Component

Validate with…
Name of the Validator Routine to use to check the timing of this stimulus. Options are generated live, so will vary according to your setup.

See also

API reference for *GratingStim*

Image Component

The Image stimulus allows an image to be presented, which can be a bitmap image from a variety of standard file formats, with an optional transparency mask that can effectively control the shape of the image. The mask can also be derived from an image file, or mathematical form such as a Gaussian.

It is a really good idea to get your image in roughly the size (in pixels) that it will appear on screen to save memory. If you leave the resolution at 12 megapixel camera, as taken from your camera, but then present it on a standard screen at 1680x1050 (=1.6 megapixels) then |PsychoPy| and your graphics card have to do an awful lot of unnecessary work. There is a performance advantage (in terms of milliseconds) to using images which are square and powers of two (32, 64, 128, etc.), but this is slight and would not be noticed in the majority of experiments.

Images can have their position, orientation, size and other settings manipulated on a frame-by-frame basis.

Categories:
Stimuli

Works in:
PsychoPy, PsychoJS

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name
Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start
When the Image Component should start, see *Defining the onset/duration of components*.

Expected start (s)
If you are using frames to control timing of your stimuli, you can add an expected start time to display the component timeline in the routine.

Start type
How do you want to define your start point?

Options:

- time (s)
- frame N
- condition

Stop

When the Image Component should stop, see *Defining the onset/duration of components*.

Expected duration (s)

If you are using frames to control timing of your stimuli, you can add an expected duration to display the component timeline in the routine.

Stop type

How do you want to define your end point?

Options:

- duration (s)
- duration (frames)
- time (s)
- frame N
- condition

Image

A filename or a standard name (sin, sqr). Filenames can be relative or absolute paths and can refer to most image formats (e.g. tif, jpg, bmp, png, etc.). If this is set to none, the patch will be a flat colour.

Layout

How should the stimulus be laid out on screen? Padding, margins, size, position, etc.

Size [w,h]

Size of this stimulus (either a single value or x,y pair, e.g. 2.5, [1,2]). If the mask is a Gaussian then the size refers to width at 3 standard deviations on either side of the mean (i.e. sd=size/6) Set this to be blank to get the image in its native size.

Position [x,y]

Position of this stimulus (e.g. [1,2])

Spatial units

Spatial units for this stimulus (e.g. for its *position* and *size*), see *Units for the window and stimuli* for more info.

Options:

- from exp settings
- deg
- cm
- pix
- norm
- height
- degFlatPos
- degFlat

Anchor
Which point in this stimulus should be anchored to the point specified by *Position [x,y]*?

Options:

- center
- top-center
- bottom-center
- center-left
- center-right
- top-left
- top-right
- bottom-left
- bottom-right

Orientation
Orientation of this stimulus (in deg)

Options:

- -360
- 360

Flip vertically
Should the image be flipped vertically (top to bottom)?

Flip horizontally
Should the image be flipped horizontally (left to right)?

Draggable?
Should this stimulus be moveble by clicking and dragging?

Appearance

How should the stimulus look? Colors, borders, styles, etc.

Foreground color
Foreground color of this stimulus (e.g. $[1,1,0], red)

Color space
In what format (color space) have you specified the colors? See *Color spaces* for more info.

Options:

- rgb
- dkl
- lms
- hsv

Opacity
Vary the transparency, from 0.0 (invisible) to 1.0 (opaque)

Contrast
Contrast of the stimulus (1.0=unchanged contrast, 0.5=decrease contrast, 0.0=uniform/no contrast, -0.5=slightly inverted, -1.0=totally inverted)

Texture

Mask
A filename, a standard name (gauss, circle, raisedCos) or a numpy array of dimensions NxNx1. The mask can define the shape (e.g. circle will make the patch circular) or something which overlays the patch e.g. noise.

Texture resolution
This is only needed if you use a synthetic texture (e.g. sinusoidal grating) as the image.

Interpolate
If *linear* is selected then linear interpolation will be applied when the image is rescaled to the appropriate size for the screen. *Nearest* will use a nearest-neighbour rule.

Options:

- linear
- nearest

Data

What information about this Component should be saved?

Save onset/offset times
Store the onset/offset times in the data file (as well as in the log file).

Sync timing with screen refresh
Synchronize times with screen refresh (good for visual stimuli and responses based on them)

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Component
Disable this Component

Validate with…
Name of the Validator Routine to use to check the timing of this stimulus. Options are generated live, so will vary according to your setup.

See also

API reference for `ImageStim`

Joy Buttons Component

The JoyButtons component can be used to collect gamepad/joystick button responses from a participant.

By not storing the button number pressed and checking the *forceEndTrial* box it can be used simply to end a *Routine* If no gamepad/joystic is installed the keyboard can be used to simulate button presses by pressing ‘ctrl’ + ‘alt’ + digit(0-9).

Categories:
Responses

Works in:
PsychoPy

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name

Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start

When the Joy Buttons Component should start, see *Defining the onset/duration of components*.

Expected start (s)

If you are using frames to control timing of your stimuli, you can add an expected start time to display the component timeline in the routine.

Start type

How do you want to define your start point?

Options:

- time (s)
- frame N
- condition

Stop

When the Joy Buttons Component should stop, see *Defining the onset/duration of components*.

Expected duration (s)

If you are using frames to control timing of your stimuli, you can add an expected duration to display the component timeline in the routine.

Stop type

How do you want to define your end point?

Options:

- duration (s)
- duration (frames)
- time (s)
- frame N
- condition

Force end of Routine

Should an *allowed* response force the end of the Routine (e.g end the trial)?

Device

Information about the device associated with this Component. Keyboards, speakers, microphones, etc.

Device number

Device number, if you have multiple devices which one do you want (0, 1, 2…)

Data

What information about this Component should be saved?

Allowed buttons

A list of allowed buttons can be specified here, e.g. [0,1,2,3], or the name of a variable holding such a list. If this box is left blank then any button that is pressed will be read. Only *allowed buttons* count as having been pressed; any other button will not be stored and will not force the end of the Routine. Note that button numbers (0, 1, 2, 3, . . .), should be separated by commas.

Store

Choose which (if any) responses to store at the end of a trial. If the button press is to force the end of the trial then this setting is unlikely to be necessary, unless two buttons happen to be pressed in the same video frame. The response time will also be stored if a button press is recorded. This time will be taken from the start of joyButtons checking (e.g. if the joyButtons was initiated 2 seconds into the trial and a button was pressed 3.2s into the trials the response time will be recorded as 1.2s).

Options:

- last key
- first key
- all keys
- nothing

Store correct

Check this box if you wish to store whether or not this button press was correct. If so then fill in the next box that defines what would constitute a correct answer e.g. 1 or *$corrAns* (note this should not be in inverted commas). This is given as Python code that should return True (1) or False (0). Often this correct answer will be defined in the settings of the *Loops*.

Correct answer

What is the 'correct' key? Might be helpful to add a correctAns column and use $correctAns to compare to the key press.

Save onset/offset times

Store the onset/offset times in the data file (as well as in the log file).

Sync RT with screen

A reaction time to a visual stimulus should be based on when the screen flipped

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Component

Disable this Component

Joystick Component

The Joystick component can be used to collect responses from a participant. The coordinates of the joystick location are given in the same coordinates as the Window, with (0,0) in the centre. Coordinates are correctly scaled for 'norm' and 'height' units. User defined scaling can be set by updating joystick.xFactor and joystick.yFactor to the desired values. Joystick.device.getX() and joystick.device.getY() always return 'norm' units. Joystick.getX() and joystick.getY() are scaled by xFactor or yFactor

No cursor is drawn to represent the joystick current position, but this is easily provided by updating the position of a partially transparent '.png' immage on each screen frame using the joystick coordinates: joystick.getX() and joystick.getY(). To ensure that the cursor image is drawon on top of other images it should be the last image in the trial.

Joystick Emulation

If no joystick device is found, the mouse and keyboard are used to emulate a joystick device. Joystick position corresponds to mouse position and mouse buttons correspond to joystick buttons (0,1,2). Other buttons can be simulated with key chords: 'ctrl' + 'alt' + digit(0..9).

Categories:

Responses

Works in:

PsychoPy

Scenarios

This can be used in various ways. Here are some scenarios (email the list if you have other uses for your joystick):

Use the joystick to record the location of a button press

Use the joystick to control stimulus parameters

Imagine you want to use your joystick to make your 'patch'_ bigger or smaller and save the final size. Call your *joystickComponent* 'joystick', set it to save its state at the end of the trial and set the button press to end the Routine. Then for the size setting of your Patch stimulus insert *$joystick.getX()* to use the x position of the joystick to control the size or *$joystick.getY()* to use the y position.

Tracking the entire path of the joystick during a period

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name

Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start

When the Joystick Component should start, see *Defining the onset/duration of components*.

Expected start (s)

If you are using frames to control timing of your stimuli, you can add an expected start time to display the component timeline in the routine.

Start type

How do you want to define your start point?

Options:

- time (s)
- frame N
- condition

Stop

When the Joystick Component should stop, see *Defining the onset/duration of components*.

Expected duration (s)

If you are using frames to control timing of your stimuli, you can add an expected duration to display the component timeline in the routine.

Stop type

How do you want to define your end point?

Options:

- duration (s)
- duration (frames)
- time (s)
- frame N
- condition

End Routine on press

Should a button press force the end of the Routine (e.g end the trial)?

Options:

- never
- any click
- valid click

Device

Information about the device associated with this Component. Keyboards, speakers, microphones, etc.

Device number

Device number, if you have multiple devices which one do you want (0, 1, 2…)

Data

What information about this Component should be saved?

Save joystick state

How often should the joystick state (x,y,buttons) be stored? On every video frame, every click or just at the end of the Routine?

Options:

- final
- on click
- every frame
- never

Time relative to

What should the values of joystick.time be relative to?

Options:

- joystick onset
- experiment
- routine

Clickable stimuli

A comma-separated list of your stimulus names that can be "clicked" by the participant. e.g. target, foil

Store params for clicked

The params (e.g. name, text), for which you want to store the current value, for the stimulus that was"clicked" by the joystick. Make sure that all the clickable objects have all these params.

Allowed buttons
Buttons to be read (blank for any) numbers separated by commas

Save onset/offset times
Store the onset/offset times in the data file (as well as in the log file).

Sync timing with screen refresh
Synchronize times with screen refresh (good for visual stimuli and responses based on them)

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Component
Disable this Component

See also

API reference for `Joystick`

Keyboard Component

The Keyboard component can be used to collect responses from a participant.

By not storing the key press and checking the *forceEndTrial* box it can be used simply to end a *Routine*

Categories:
Responses

Works in:
PsychoPy, PsychoJS

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name
Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start
When the Keyboard Component should start, see *Defining the onset/duration of components*.

Expected start (s)
If you are using frames to control timing of your stimuli, you can add an expected start time to display the component timeline in the routine.

Start type
How do you want to define your start point?

Options:

- time (s)
- frame N
- condition

Stop

When the Keyboard Component should stop, see *Defining the onset/duration of components*.

Expected duration (s)

If you are using frames to control timing of your stimuli, you can add an expected duration to display the component timeline in the routine.

Stop type

How do you want to define your end point?

Options:

- duration (s)
- duration (frames)
- time (s)
- frame N
- condition

Force end of Routine

Should a response force the end of the Routine (e.g end the trial)?

Register keypress on…

When should the keypress be registered? As soon as pressed, or when released?

Options:

- press
- release

Allowed keys

A list of allowed keys can be specified here, e.g. [‘m’,’z’,’1’,’2’], or the name of a variable holding such a list. If this box is left blank then any key that is pressed will be read. Only *allowed keys* count as having been pressed; any other key will not be stored and will not force the end of the Routine. Note that key names (even for number keys) should be given in single quotes, separated by commas. Cursor control keys can be accessed with ‘up’, ‘down’, and so on; the space bar is ‘space’. To find other special keys, run the Coder Input demo, “what_key.py”, press the key, and check the Coder output window.

Device

Information about the device associated with this Component. Keyboards, speakers, microphones, etc.

Device label

A label to refer to this Component’s associated hardware device by. If using the same device for multiple components, be sure to use the same label here.

Data

What information about this Component should be saved?

Store

Which key press, if any, should be stored; the first to be pressed, the last to be pressed or all that have been pressed. If the key press is to force the end of the trial then this setting is unlikely to be necessary, unless two keys happen to be pressed in the same video frame. The response time will also be stored if a keypress is recorded. This time will be taken from the start of keyboard checking (e.g. if the keyboard was initiated 2 seconds into the trial and a key was pressed 3.2s into the trials the response time will be recorded as 1.2s).

Options:

- last key
- first key
- all keys
- nothing

Store correct
Check this box if you wish to store whether or not this key press was correct. If so then fill in the next box that defines what would constitute a correct answer e.g. left, 1 or *$corrAns* (note this should not be in inverted commas). This is given as Python code that should return True (1) or False (0). Often this correct answer will be defined in the settings of the *Loops*.

Correct answer
What is the 'correct' key? Might be helpful to add a correctAns column and use $correctAns to compare to the key press.

Save onset/offset times
Store the onset/offset times in the data file (as well as in the log file).

Sync timing with screen
A reaction time to a visual stimulus should be based on when the screen flipped

Discard previous
Do you want to discard all responses occurring before the onset of this Component?

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Component
Disable this Component

See also

API reference for psychopy.hardware.keyboard>

Microphone Component

The microphone component provides a way to record sound during an experiment. You can even transcribe the recording to text! Take a look at the documentation on *Creating a Google Cloud Speech API key* to get started with that.

When using a mic recording, specify the starting time relative to the start of the routine (see *start* below) and a stop time (= duration in seconds). A blank duration evaluates to recording for 0.000s.

The resulting sound files are saved in .wav format (at the specified sampling frequency), one file per recording. The files appear in a new folder within the data directory (the subdirectory name ends in *_wav*). The file names include the unix (epoch) time of the onset of the recording with milliseconds, e.g., *mic-1346437545.759.wav*.

It is possible to stop a recording that is in progress by using a code component. Every frame, check for a condition (such as key 'q', or a mouse click), and call the *mic.stop()* method of the microphone component. The recording will end at that point and be saved.

Categories:
Responses

Works in:
PsychoPy, PsychoJS

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name
Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start
When the Microphone Component should start, see *Defining the onset/duration of components*.

Expected start (s)
If you are using frames to control timing of your stimuli, you can add an expected start time to display the component timeline in the routine.

Start type
How do you want to define your start point?

Options:

- time (s)
- frame N
- condition

Stop
When the Microphone Component should stop, see *Defining the onset/duration of components*.

Expected duration (s)
If you are using frames to control timing of your stimuli, you can add an expected duration to display the component timeline in the routine.

Stop type
The duration of the recording in seconds; blank = 0 sec

Options:

- duration (s)

Device

Information about the device associated with this Component. Keyboards, speakers, microphones, etc.

Device label
A label to refer to this Component's associated hardware device by. If using the same device for multiple components, be sure to use the same label here.

Device
What microphone device would you like the use to record? This will only affect local experiments - online experiments ask the participant which mic to use.

Channels
Record two channels (stereo) or one (mono, smaller file). Select 'auto' to use as many channels as the selected device allows.

Options:

- Auto
- Mono

- Stereo

Sample rate (hz)
How many samples per second (Hz) to record at

Exclusive control
Take exclusive control of the microphone, so other apps can't use it during your experiment.

Max recording size (kb)
To avoid excessively large output files, what is the biggest file size you are likely to expect?

Transcription

Transcribe audio
Whether to transcribe the audio recording and store the transcription

Transcription backend
What transcription service to use to transcribe audio?

Options:

- Google: Uses Google's cloud based speech-to-text engine, requiring a key from Google to use. We *highly* recommend taking a look at the documentation on *Creating a Google Cloud Speech API key* to get started.
- Whisper (OpenAI): Uses an open-source speech recognition AI. Requires the psychopy-whisper plugin to be installed, and will work better with a dedicated graphics card (as the model uses GPU to speed up processing)

Transcription language
What language you expect the recording to be spoken in, e.g. en-US for English

Expected words
Set list of words to listen for - if blank will listen for all words in chosen language.

If using the built-in transcriber, you can set a minimum % confidence level using a colon after the word, e.g. 'red:100', 'green:80'. Otherwise, default confidence level is 80%.

Speaking start / stop times
Tick this to save times when the participant starts and stops speaking

Whisper model (*if :ref:`microphonecomponent-transcribeBackend` is "Whisper"*)
Which model of Whisper AI should be used for transcription? Details of each model are available here

Options:

- tiny
- base
- small
- medium
- large
- tiny.en
- base.en
- small.en
- medium.en

Whisper device (*if :ref:`microphonecomponent-transcribeBackend` is "Whisper"*)
Which device to use for transcription?

Options:

- auto
- gpu
- cpu

Data

What information about this Component should be saved?

Save onset/offset times
Store the onset/offset times in the data file (as well as in the log file).

Sync timing with screen refresh
Synchronize times with screen refresh (good for visual stimuli and responses based on them)

Output file type
What file type should output audio files be saved as?

Options:

- default
- aiff
- au
- avr
- caf
- flac
- htk
- svx
- mat4
- mat5
- mpc2k
- mp3
- ogg
- paf
- pvf
- raw
- rf64
- sd2
- sds
- ircam
- voc
- w64
- wav

- nist
- wavex
- wve
- xi

Full buffer policy

What to do when we reach the max amount of audio data which can be safely stored in memory?

Options:

- Discard incoming data
- Clear oldest data
- Raise error

Trim silent

Trim periods of silence from the output file

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Component

Disable this Component

Mouse Component

The Mouse component can be used to collect responses from a participant. The coordinates of the mouse location are given in the same coordinates as the Window, with (0,0) in the centre.

Categories:

Responses

Works in:

PsychoPy, PsychoJS

Scenarios

This can be used in various ways. Here are some scenarios (email the list if you have other uses for your mouse):

Use the mouse to record the location of a button press

Use the mouse to control stimulus parameters

Imagine you want to use your mouse to make your 'patch'_ bigger or smaller and save the final size. Call your *Mouse Component* 'mouse', set it to save its state at the end of the trial and set the button press to end the Routine. Then for the size setting of your Patch stimulus insert *$mouse.getPos()[0]* to use the x position of the mouse to control the size or *$mouse.getPos()[1]* to use the y position.

Tracking the entire path of the mouse during a period

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name

Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start

When the Mouse Component should start, see *Defining the onset/duration of components*.

Expected start (s)

If you are using frames to control timing of your stimuli, you can add an expected start time to display the component timeline in the routine.

Start type

How do you want to define your start point?

Options:

- time (s)
- frame N
- condition

Stop

When the Mouse Component should stop, see *Defining the onset/duration of components*.

Expected duration (s)

If you are using frames to control timing of your stimuli, you can add an expected duration to display the component timeline in the routine.

Stop type

How do you want to define your end point?

Options:

- duration (s)
- duration (frames)
- time (s)
- frame N
- condition

End Routine on press

Should a button press force the end of the Routine (e.g end the trial)?

Options:

- never
- any click
- valid click
- correct click

New clicks only

If the mouse button is already down when we start checking then wait for it to be released before recording as a new click.

Clickable stimuli

A comma-separated list of your stimulus names that can be "clicked" by the participant. e.g. target, foil

Data

What information about this Component should be saved?

Save mouse state

How often should the mouse state (x,y,buttons) be stored? On every video frame, every click or just at the end of the Routine?

Options:

- final
- on click
- on valid click
- every frame
- never

Time relative to

What should the values of mouse.time should be relative to?

Options:

- mouse onset
- experiment
- routine

Store params for clicked

The params (e.g. name, text), for which you want to store the current value, for the stimulus that was”clicked” by the mouse. Make sure that all the clickable objects have all these params.

Save onset/offset times

Store the onset/offset times in the data file (as well as in the log file).

Sync timing with screen refresh

Synchronize times with screen refresh (good for visual stimuli and responses based on them)

Store correct

Do you want to save the response as correct/incorrect?

Correct answer

What is the ‘correct’ object? To specify an area, remember that you can create a shape Component with 0 opacity.

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Component

Disable this Component

Movie Component

The Movie component allows movie files to be played from a variety of formats (e.g. mpeg, avi, mov).

The movie can be positioned, rotated, flipped and stretched to any size on the screen (using the *Units for the window and stimuli* given).

Categories:

Stimuli

Works in:

PsychoPy, PsychoJS

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name
Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start
When the Movie Component should start, see *Defining the onset/duration of components*.

Expected start (s)
If you are using frames to control timing of your stimuli, you can add an expected start time to display the component timeline in the routine.

Start type
How do you want to define your start point?

Options:

- time (s)
- frame N
- condition

Stop
When the Movie Component should stop, see *Defining the onset/duration of components*.

Expected duration (s)
If you are using frames to control timing of your stimuli, you can add an expected duration to display the component timeline in the routine.

Stop type
How do you want to define your end point?

Options:

- duration (s)
- duration (frames)
- time (s)
- frame N
- condition

Movie file
A filename for the movie (including path)

Force end of Routine
Should the end of the movie cause the end of the Routine (e.g. trial)?

Layout

How should the stimulus be laid out on screen? Padding, margins, size, position, etc.

Size [w,h]
Size of this stimulus (either a single value or x,y pair, e.g. 2.5, [1,2]

Position [x,y]
Position of this stimulus (e.g. [1,2])

Spatial units

Spatial units for this stimulus (e.g. for its *position* and *size*), see *Units for the window and stimuli* for more info.

Options:

- from exp settings
- deg
- cm
- pix
- norm
- height
- degFlatPos
- degFlat

Anchor

Which point in this stimulus should be anchored to the point specified by *Position [x,y]*?

Options:

- center
- top-center
- bottom-center
- center-left
- center-right
- top-left
- top-right
- bottom-left
- bottom-right

Orientation

Orientation of this stimulus (in deg)

Options:

- -360
- 360

Appearance

How should the stimulus look? Colors, borders, styles, etc.

Opacity

Vary the transparency, from 0.0 (invisible) to 1.0 (opaque)

Contrast

Contrast of the stimulus (1.0=unchanged contrast, 0.5=decrease contrast, 0.0=uniform/no contrast, -0.5=slightly inverted, -1.0=totally inverted)

Playback

How should stimulus play? Speed, volume, etc.

Loop playback
Whether the movie should loop back to the beginning on completion.

No audio
Prevent the audio stream from being loaded/processed (moviepy and opencv only)

Backend (*if running locally*)
What underlying Python library to use for loading movies

Options:

- ffpyplayer
- moviepy
- opencv
- vlc

Volume
How loud should audio be played?

Stop with Routine?
Should playback cease when the Routine ends? Untick to continue playing after the Routine has finished.

Data

What information about this Component should be saved?

Save onset/offset times
Store the onset/offset times in the data file (as well as in the log file).

Sync timing with screen refresh
Synchronize times with screen refresh (good for visual stimuli and responses based on them)

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Component
Disable this Component

Validate with…
Name of the Validator Routine to use to check the timing of this stimulus. Options are generated live, so will vary according to your setup.

See also

API reference for `MovieStim`

Panorama Component

Panorama: Present a panoramic image (such as from a phone camera in Panorama mode) on screen.

Categories:
Stimuli

Works in:
PsychoPy

Note: Since this is still in beta, keep an eye out for bug fixes.

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name
Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start
When the Panorama Component should start, see *Defining the onset/duration of components*.

Expected start (s)
If you are using frames to control timing of your stimuli, you can add an expected start time to display the component timeline in the routine.

Start type
How do you want to define your start point?

Options:

- time (s)
- frame N
- condition

Stop
When the Panorama Component should stop, see *Defining the onset/duration of components*.

Expected duration (s)
If you are using frames to control timing of your stimuli, you can add an expected duration to display the component timeline in the routine.

Stop type
How do you want to define your end point?

Options:

- duration (s)
- duration (frames)
- time (s)
- frame N
- condition

Image
The image to be displayed - a filename, including path. *The image used cannot be more than 18,000,000 pixels - this is roughly the resolution of a 6k screen*

Position control
How to control looking around the panorama scene

Options:

- Mouse

- Drag
- Keyboard (Arrow Keys)
- Keyboard (WASD)
- Keyboard (Custom keys)
- Custom

Azimuth (*if :ref:`panoramacomponent-posCtrl` is "Custom"*)
Horizontal look position, ranging from -1 (fully left) to 1 (fully right)

Elevation (*if :ref:`panoramacomponent-posCtrl` is "Custom"*)
Vertical look position, ranging from -1 (fully down) to 1 (fully up)

Up / Down / Left / Right / Stop (*if :ref:`panoramacomponent-posCtrl` is "Keyboard (Custom keys)"*)
What key corresponds to each view action?

Movement sensitivity (*if :ref:`panoramacomponent-posCtrl` is not "Custom"*)
Multiplier to apply to view changes. 1 means that moving the mouse from the center of the screen to the edge or holding down a key for 2s will rotate 180°.

Note: The bigger the multiplier, the quicker the movement

Smooth? (*if :ref:`panoramacomponent-posCtrl` is not "Custom" or "Mouse"*)
Should movement be smoothed, so the view keeps moving a little after a change?

Zoom control
How to control zooming in and out of the panorama scene

Options:

- Mouse Wheel
- Mouse Wheel (Inverted)
- Keyboard (Arrow Keys)
- Keyboard (+-)
- Keyboard (Custom keys)
- Custom

Zoom (*if :ref:`panoramacomponent-zoomCtrl` is "Custom"*)
How zoomed in the scene is, with 1 being no adjustment.

Zoom in / Zoom out (*if :ref:`panoramacomponent-zoomCtrl` is "Keyboard (Custom keys)"*)
What keys correspond to zooming in and out?

Zoom sensitivity (*if :ref:`panoramacomponent-zoomCtrl` is not "Custom"*)
Multiplier to apply to zoom changes. 1 means that pressing the zoom in key for 1s or scrolling the mouse wheel 100% zooms in 100%.

Interpolate
How should the image be interpolated if/when rescaled

Options:

- linear
- nearest

Data

What information about this Component should be saved?

Save onset/offset times
Store the onset/offset times in the data file (as well as in the log file).

Sync timing with screen refresh
Synchronize times with screen refresh (good for visual stimuli and responses based on them)

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Component
Disable this Component

Validate with…
Name of the Validator Routine to use to check the timing of this stimulus. Options are generated live, so will vary according to your setup.

See also

API reference for `Panorama`

Parallel Out Component

This component allows you to send triggers to a parallel port, USB2TTL8, or LabJack U3 device.

An example usage would be in EEG experiments to set the port to 0 when no stimuli are present and then set it to an identifier value for each stimulus synchronised to the start/stop of that stimulus. In that case you might set the *Start data* to be *$ID* (with ID being a column in your conditions file) and set the *Stop Data* to be 0

Categories:
I/O, EEG

Works in:
PsychoPy

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name
Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start
When the Parallel Out Component should start, see *Defining the onset/duration of components*.

Expected start (s)
If you are using frames to control timing of your stimuli, you can add an expected start time to display the component timeline in the routine.

Start type
How do you want to define your start point?

Options:

- time (s)
- frame N
- condition

Stop

When the Parallel Out Component should stop, see *Defining the onset/duration of components*.

Expected duration (s)

If you are using frames to control timing of your stimuli, you can add an expected duration to display the component timeline in the routine.

Stop type

How do you want to define your end point?

Options:

- duration (s)
- duration (frames)
- time (s)
- frame N
- condition

Device

Information about the device associated with this Component. Keyboards, speakers, microphones, etc.

Port address

You need to know the address of the parallel port you wish to write to. The options that appear in this drop-down list are determined by the application preferences. You can add your particular port there if you prefer.

Options:

- 0x0378
- 0x03BC
- LabJack U3
- USB2TTL8

U3 register (*if :ref:`paralleloutcomponent-address` ==’LabJack U3’*)

When using a LabJack U3, you can select which register is used to write a data byte to. Register EIO is the default.

Options:

- EIO
- FIO

Data

What information about this Component should be saved?

Start data

Data to be sent at ‘start’. The value is given as a byte (a value from 0-255) controlling the 8 data pins of the parallel port.

Stop data

Data to be sent at 'end'. The value is given as a byte (a value from 0-255) controlling the 8 data pins of the parallel port.

Save onset/offset times

Store the onset/offset times in the data file (as well as in the log file).

Sync timing with screen refresh

If true then the parallel port will be sent synchronised to the next screen refresh, which is ideal if it should indicate the onset of a visual stimulus. If set to False then the data will be set on the parallel port immediately.

Sync to screen

If the parallel port data relates to visual stimuli then sync its pulse to the screen refresh

Options:

- True
- False

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Component

Disable this Component

Pavlovia Survey Routine

This component is only for use with online experiments.

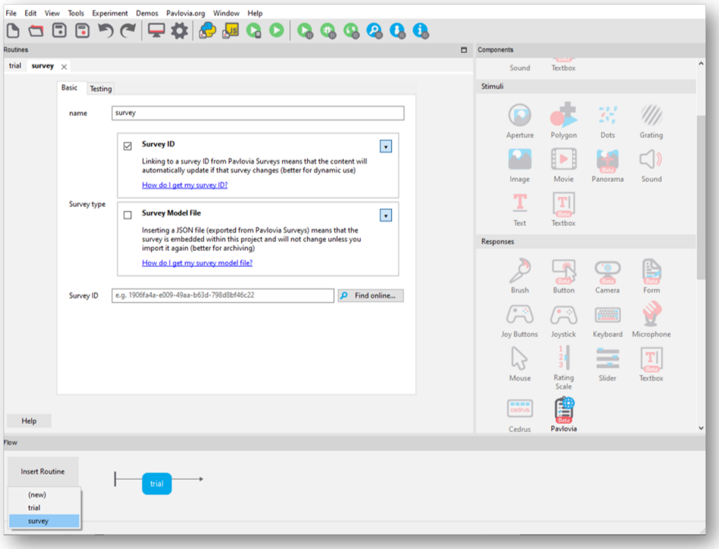
You can use Pavlovia.org to create feature rich surveys, with a range of response options, which display nicely across a range of devices (i.e. laptops, smart phones, tablets). To create and launch a Pavlovia Survey, you technically do not need the PsychoPy app at all.

Useful links for creating surveys:

- Pavlovia surveys launch video.
- Pavlovia Surveys information.
- YouTube tutorial on how to use the Pavlovia Surveys component.

The Pavlovia Survey Routine is used to integrate a Pavlovia Survey into a behavioural task you have created in PsychoPy.

The Pavlovia Survey Routine is a "Standalone Routine", which means rather than adding a Component to an existing Routine, it will create a whole new Routine, which you can then add to your flow. Once you have selected the Routine, select Insert Routine and add it to your flow.



name: survey

Survey type

☑ **Survey ID**

Linking to a survey ID from Pavlovia Surveys means that the content will automatically update if that survey changes (better for dynamic use)

How do I get my survey ID?

☐ **Survey Model File**

Inserting a JSON file (exported from Pavlovia Surveys) means that the survey is embedded within this project and will not change unless you import it again (better for archiving)

How do I get my survey model file?

Survey ID: e.g. 1906fa4a-e009-49aa-b63d-798d8bf46c22 Find online...

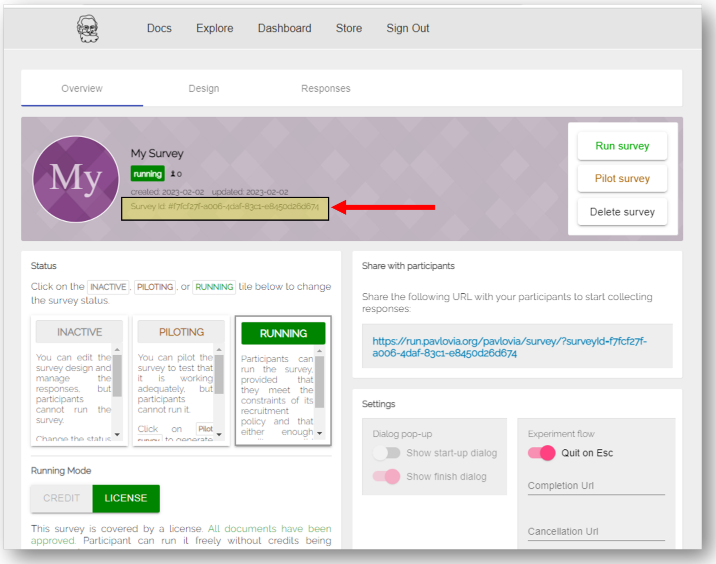
To specify a survey you can either use “Survey ID” or “Survey Model File”.

Categories:
Responses

Works in:
PsychoJS

Get ID

You can make a Pavlovia Survey in Pavlovia by selecting “Dashboard” and Surveys (for details see this guide). Once you have created a Survey, the survey ID will be visible in the “Overview” tab of that survey as shown below. Alternatively, you can find the Survey directly from PsychoPy by selecting “Find online…”



My Survey

running 0

created: 2023-02-02 updated: 2023-02-02

Survey Id: #f7fcf27f-a006-4daf-83c1-e8450d26d674

Status

Click on the INACTIVE, PILOTING, or RUNNING tile below to change the survey status.

INACTIVE: You can edit the survey design and manage the responses, but participants cannot run the survey. Change the status

PILOTING: You can pilot the survey to test that it is working adequately, but participants cannot run it. Click on Pilot survey to generate

RUNNING: Participants can run the survey, provided that they meet the constraints of its recruitment policy and that either enough

Running Mode

This survey is covered by a license. All documents have been approved. Participant can run it freely without credits being

Share with participants

Share the following URL with your participants to start collecting responses:

https://run.pavlovia.org/pavlovia/survey/?surveyId=f7fcf27f-a006-4daf-83c1-e8450d26d674

Settings

Dialog pop-up

Show start-up dialog

Show finish dialog

Experiment flow

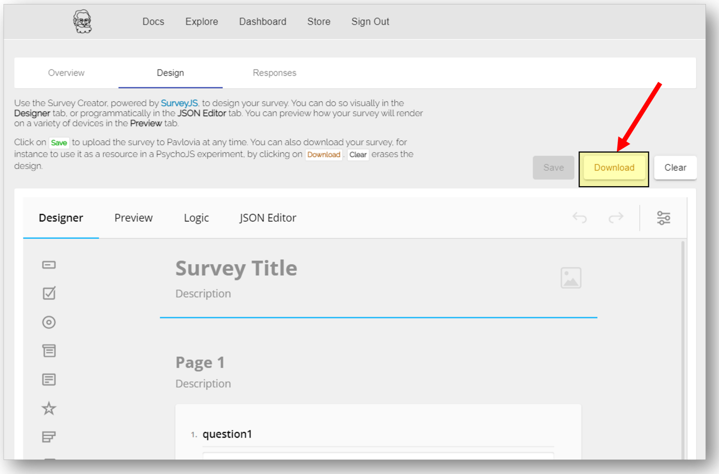
Quit on Esc

Completion Url

Cancellation Url

Get JSON

Another way you can add a Pavlovia Survey to your experiment is by directly adding the “Survey Model File”. When creating a Survey in Pavlovia you can select “Download” to download the json file used to create that Survey (you could actually share this with others and they could “Import” your json to re-use your Survey!). In PsychoPy, if you select “Survey Model File” - you will need to load the json file you’ve downloaded.



Use the Survey Creator, powered by SurveyJS, to design your survey. You can do so visually in the **Designer** tab, or programmatically in the **JSON Editor** tab. You can preview how your survey will render on a variety of devices in the **Preview** tab.

Click on Save to upload the survey to Pavlovia at any time. You can also download your survey, for instance to use it as a resource in a PsychoJS experiment, by clicking on Download . Clear erases the design.

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name
Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Survey type
How to specify the survey.

Options:

- Survey id: Linking to a survey ID from Pavlovia Surveys means that the content will automatically update if that survey changes (better for dynamic use)
- Survey Model File: Inserting a JSON file (exported from Pavlovia Surveys) means that the survey is embedded within this project and will not change unless you import it again (better for archiving)

Survey id (*if :ref:`pavloviasurveyroutine-surveytype` is “Survey id”*)
The ID for your survey on Pavlovia. Tip: Right click to open the survey in your browser!

Survey JSON (*if :ref:`pavloviasurveyroutine-surveytype` is “Survey Model File”*)
File path of the JSON file used to construct the survey

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Routine
Disable this Routine

Polygon Component

The Polygon stimulus allows you to present a wide range of regular geometric shapes. The basic control comes from setting the number of vertices:

- 2 vertices give a line
- 3 give a triangle
- 4 give a rectangle etc.
- a large number will approximate a circle/ellipse

The size parameter takes two values. For a line only the first is used (then use ori to specify the orientation). For triangles and rectangles the size specifies the height and width as expected. Note that for pentagons upwards, however, the size determines the width/height of the ellipse on which the vertices will fall, rather than the width/height of the vertices themselves (slightly smaller typically).

Categories:
Stimuli

Works in:
PsychoPy, PsychoJS

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name
Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start
When the Polygon Component should start, see *Defining the onset/duration of components*.

Expected start (s)
If you are using frames to control timing of your stimuli, you can add an expected start time to display the component timeline in the routine.

Start type
How do you want to define your start point?

Options:

- time (s)
- frame N
- condition

Stop
When the Polygon Component should stop, see *Defining the onset/duration of components*.

Expected duration (s)
If you are using frames to control timing of your stimuli, you can add an expected duration to display the component timeline in the routine.

Stop type
How do you want to define your end point?

Options:

- duration (s)

- duration (frames)
- time (s)
- frame N
- condition

Shape
What shape is this?

Options:

- Line
- Triangle
- Rectangle
- Circle
- Cross
- Star
- Arrow
- Regular polygon…
- Custom polygon…

Num. vertices (*if :ref:`polygoncomponent-shape` is "Regular polygon…"*)
How many vertices in your regular polygon?

Vertices (*if :ref:`polygoncomponent-shape` is "custom polygon…"*)
What are the vertices of your polygon? Should be an nx2 array or a list of [x, y] lists

Layout

How should the stimulus be laid out on screen? Padding, margins, size, position, etc.

Size [w,h]
Size of this stimulus [w,h]. Note that for a line only the first value is used, for triangle and rect the [w,h] is as expected, but for higher-order polygons it represents the [w,h] of the ellipse that the polygon sits on!!

Position [x,y]
Position of this stimulus (e.g. [1,2])

Spatial units
Spatial units for this stimulus (e.g. for its *position* and *size*), see *Units for the window and stimuli* for more info.

Options:

- from exp settings
- deg
- cm
- pix
- norm
- height
- degFlatPos
- degFlat

Anchor (*if :ref:`polygoncomponent-shape` isn’t ==’line’*)
Which point in this stimulus should be anchored to the point specified by *Position [x,y]*?

Options:

- center
- top-center
- bottom-center
- center-left
- center-right
- top-left
- top-right
- bottom-left
- bottom-right

Orientation
Orientation of this stimulus (in deg)

Options:

- -360
- 360

Draggable?
Should this stimulus be moveble by clicking and dragging?

Appearance

How should the stimulus look? Colors, borders, styles, etc.

Fill color
Fill color of this stimulus (e.g. $[1,1,0], red)

Border color
Border color of this stimulus (e.g. $[1,1,0], red)

Color space
In what format (color space) have you specified the colors? See *Color spaces* for more info.

Options:

- rgb
- dkl
- lms
- hsv

Opacity
Vary the transparency, from 0.0 (invisible) to 1.0 (opaque)

Contrast
Contrast of the stimulus (1.0=unchanged contrast, 0.5=decrease contrast, 0.0=uniform/no contrast, -0.5=slightly inverted, -1.0=totally inverted)

Line width
Width of the shape’s line (always in pixels - this does NOT use ‘units’)

Texture

Control how the stimulus handles textures.

Interpolate
How should the image be interpolated if/when rescaled

Options:

- linear
- nearest

Data

What information about this Component should be saved?

Save onset/offset times
Store the onset/offset times in the data file (as well as in the log file).

Sync timing with screen refresh
Synchronize times with screen refresh (good for visual stimuli and responses based on them)

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Component
Disable this Component

Validate with…
Name of the Validator Routine to use to check the timing of this stimulus. Options are generated live, so will vary according to your setup.

See also

API reference for `Polygon` API reference for `Rect` API reference for `ShapeStim`

Progress Component

Present a progress bar, with values ranging from 0 to 1.

Categories:
Stimuli

Works in:
PsychoPy, PsychoJS

Note: Since this is still in beta, keep an eye out for bug fixes.

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name
Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start
When the Progress Component should start, see *Defining the onset/duration of components*.

Expected start (s)
If you are using frames to control timing of your stimuli, you can add an expected start time to display the component timeline in the routine.

Start type
How do you want to define your start point?

Options:

- time (s)
- frame N
- condition

Stop
When the Progress Component should stop, see *Defining the onset/duration of components*.

Expected duration (s)
If you are using frames to control timing of your stimuli, you can add an expected duration to display the component timeline in the routine.

Stop type
How do you want to define your end point?

Options:

- duration (s)
- duration (frames)
- time (s)
- frame N
- condition

Progress
Value between 0 (not started) and 1 (complete) to set the progress bar to. Use "set every frame" to continuously update progress throughout a Routine!

Layout

How should the stimulus be laid out on screen? Padding, margins, size, position, etc.

Size [w,h]
Size of this stimulus (either a single value or x,y pair, e.g. 2.5, [1,2]

Position [x,y]
Position of this stimulus (e.g. [1,2])

Spatial units
Spatial units for this stimulus (e.g. for its *position* and *size*), see *Units for the window and stimuli* for more info.

Options:

- from exp settings
- deg
- cm
- pix

- norm
- height
- degFlatPos
- degFlat

Anchor

Which point in this stimulus should be anchored to the point specified by *Position [x,y]*? The bar will fill up from the anchor point.

Options:

- center
- top-center
- bottom-center
- center-left
- center-right
- top-left
- top-right
- bottom-left
- bottom-right

Orientation

Orientation of this stimulus (in deg)

Options:

- -360
- 360

Appearance

How should the stimulus look? Colors, borders, styles, etc.

Bar color

Color of the filled part of the progress bar.

Back color

Color of the empty part of the progress bar.

Border color

Color of the line around the progress bar.

Color space

In what format (color space) have you specified the colors? See *Color spaces* for more info.

Options:

- rgb
- dkl
- lms
- hsv

Opacity
Vary the transparency, from 0.0 (invisible) to 1.0 (opaque)

Contrast
Contrast of the stimulus (1.0=unchanged contrast, 0.5=decrease contrast, 0.0=uniform/no contrast, -0.5=slightly inverted, -1.0=totally inverted)

Line width
Width of the shape's line (always in pixels - this does NOT use 'units')

Data

What information about this Component should be saved?

Save onset/offset times
Store the onset/offset times in the data file (as well as in the log file).

Sync timing with screen refresh
Synchronize times with screen refresh (good for visual stimuli and responses based on them)

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Component
Disable this Component

Validate with…
Name of the Validator Routine to use to check the timing of this stimulus. Options are generated live, so will vary according to your setup.

See also

API reference for `Progress`

Region Of Interest Component

Record eye movement events occurring within a defined Region of Interest (ROI). Note that you will still need to add an Eyetracker Record component to this routine to save eye movement data.

Categories:
Eyetracking

Works in:
PsychoPy

Note: Since this is still in beta, keep an *eye* (haha) out for bug fixes.

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name
Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start

When the Region Of Interest Component should start, see *Defining the onset/duration of components*.

Expected start (s)

If you are using frames to control timing of your stimuli, you can add an expected start time to display the component timeline in the routine.

Start type

How do you want to define your start point?

Options:

- time (s)
- frame N
- condition

Stop

When the Region Of Interest Component should stop, see *Defining the onset/duration of components*.

Expected duration (s)

If you are using frames to control timing of your stimuli, you can add an expected duration to display the component timeline in the routine.

Stop type

How do you want to define your end point?

Options:

- duration (s)
- duration (frames)
- time (s)
- frame N
- condition

Shape

A shape to outline the Region of Interest, see *Polygon Component*

Options:

- Line
- Triangle
- Rectangle
- Circle
- Cross
- Star
- Arrow
- Regular polygon…
- Custom polygon…

Num. vertices (*if :ref:`regionofinterestcomponent-shape` is “Regular polygon…”*)

How many vertices in your regular polygon?

Vertices (*if :ref:`regionofinterestcomponent-shape` is “Custom polygon…”*)

What are the vertices of your polygon? Should be an nx2 array or a list of [x, y] lists

End Routine on…

Under what condition should this ROI end the Routine?

Options:

- Look at: End the Routine when this ROI is looked at, for more than the *Min. look time (if :ref:`regionofinterestcomponent-endroutineon` isn't None)*
- Look away: End the Routine when this ROI is *not* looked at, for more than the *Min. look time (if :ref:`regionofinterestcomponent-endroutineon` isn't None)*
- None: This ROI will not end the Routine

Min. look time (*if :ref:`regionofinterestcomponent-endroutineon` isn't None*)

Minimum dwell time within roi (look at) or outside roi (look away).

Layout

How should the stimulus be laid out on screen? Padding, margins, size, position, etc.

Size [w,h]

Size of this stimulus [w,h]. Note that for a line only the first value is used, for triangle and rect the [w,h] is as expected, but for higher-order polygons it represents the [w,h] of the ellipse that the polygon sits on!!

Position [x,y]

Position of this stimulus (e.g. [1,2])

Spatial units

Spatial units for the ROI is fixed to the same units as the window.

Anchor (*if :ref:`regionofinterestcomponent-shape` isn't "Line"*)

Which point in this stimulus should be anchored to the point specified by *Position [x,y]*?

Options:

- center
- top-center
- bottom-center
- center-left
- center-right
- top-left
- top-right
- bottom-left
- bottom-right

Orientation

Orientation of this stimulus (in deg)

Options:

- -360
- 360

Draggable?

Should this stimulus be moveble by clicking and dragging?

Data

What information about this Component should be saved?

Save onset/offset times
Store the onset/offset times in the data file (as well as in the log file).

Sync timing with screen refresh
Synchronize times with screen refresh (good for visual stimuli and responses based on them)

Save…
What looks on this ROI should be saved to the data output?

Options:

- first look
- last look
- every look
- none

Time relative to…
What should the values of roi.time should be relative to?

Options:

- roi onset
- experiment
- routine

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Component
Disable this Component

Validate with…
Name of the Validator Routine to use to check the timing of this stimulus. Options are generated live, so will vary according to your setup.

Debug mode
In debug mode, the ROI is drawn in red. Use this to see what area of the screen is in the ROI.

Resource Manager Component

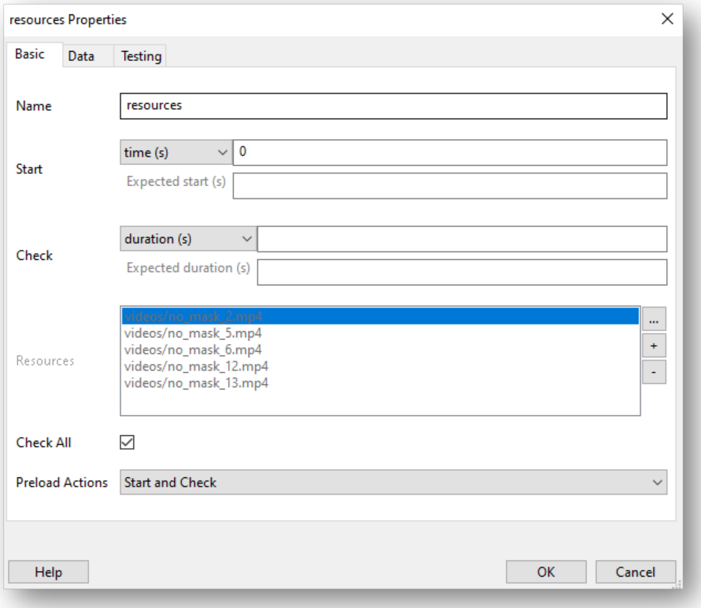
Pre-load resources into memory so that components using them can start without having to load first.

This component is only relevant to online studies. If you use this component it will override loading of resources at the beginning of the experiment. This means that any resources specified in the Experiment Settings > Online > Additional Resources will not be used, make sure to load all required resources within the experiment.

Most experiments need “resources” in order to run. Be it images, sounds, spreadsheets or movies. When running a study online through pavlovia.org, these resources are loaded by default at the beginning of the experiment, and you will usually see a loading bar.

However, sometimes this loading can take a pretty long time. This happens either because you have a very large number of resources or because individual files are large (e.g. long movies) . In cases like this, it may be preferred to load these within your experiment, for example whilst your participants are reading through the instructions, in an inter-trial interval or during a break between blocks. This is where the *Resource Manager* component and/or the *Static Component* come in.

You can find the Resource Manager under “Custom” in the Component Panel. The component has many properties similar to any other component, a start time, a duration etc. The most important fields in the component are **Resources**, indicating the list of resources to load, and **Preload Actions**, indicating if we are initiating loading (Start), checking previously initiated loading has completed (Check), or both (Start and Check). For experiments where we might have several resource manager components, we can also check if the resources from *all* components currently exist in memory by selecting “Check All”.



Name: resources

Start: time (s) 0
Expected start (s):

Check: duration (s)
Expected duration (s):

Resources:
videos/no_mask_2.mp4
videos/no_mask_5.mp4
videos/no_mask_6.mp4
videos/no_mask_12.mp4
videos/no_mask_13.mp4

Check All ☑

Preload Actions: Start and Check

Categories:
Custom

Works in:
PsychoJS

Note: Since this is still in beta, keep an eye out for bug fixes.

Examples

Loading resources in the background of instructions

A common use case for resource manager might be to load resources in the background of instructions (or any routine!), and only let your participants move forward when the resources are loaded. To do this:

1. Add a resource manager component.
2. Populate the resources field with the resources to be loaded.
3. Set *Preload Actions* to *Start and Check*.
4. Add a code component and use this in the “Each Frame” tab (where “resources” refers to the name of your resource manager component):

```
if resources.status == FINISHED:
    continueRoutine = False
```

5. Alternatively to step 4, you might want to have an image or text that is clickable, but have *Start* set to `resources.status == FINISHED`. This will make the button "pop-up" when the resources have finished loading!

> **Note**
>
> The resource manager has an attribute "status" and we can check if it has finished using *resources.status* == *FINISHED* (where *resources* corresponds to the name of your resource manager component).

Loading resources for blocked or branched designs, or loading trial-by-trial

Sometimes we might have a design where participants only need to be presented with a subset of resources. We might have 100 movies, but group 1 sees 50 movies and group 2 sees the other 50. In cases like this you might ask "How to I make the resources in my resource manager conditional?". Well, for designs like this we actually recommend you use something a little different, the *Static Component* - so check it out!.

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name
Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start
When the Resource Manager Component should start, see *Defining the onset/duration of components*.

Expected start (s)
If you are using frames to control timing of your stimuli, you can add an expected start time to display the component timeline in the routine.

Start type
How do you want to define your start point?

Options:

- time (s)
- frame N
- condition

Check
When the Resource Manager Component should stop, see *Defining the onset/duration of components*.

Expected duration (s)
If you are using frames to control timing of your stimuli, you can add an expected duration to display the component timeline in the routine.

Stop type
How do you want to define your end point?

Options:

- duration (s)
- duration (frames)
- time (s)

- frame N
- condition

Resources
Resources to download/check

Check all
When checking these resources, also check for all currently downloading?

Preload actions
Should this Component start an / or check resource preloading?

Options:

- Start and Check
- Start Only
- Check Only

Force end Routine
Should we end the Routine when the resource download is complete?

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Component
Disable this Component

Serial Out Component

This component allows you to send triggers to a serial port. For a full tutorial please see :ref: *this page <serial>*.

An example usage would be in EEG experiments to set the port to 0 when no stimuli are present and then set it to an identifier value for each stimulus synchronised to the start/stop of that stimulus. In that case you might set the *Start data* to be *$ID* (with ID being a column in your conditions file) and set the *Stop Data* to be "0".

Categories:
I/O, EEG

Works in:
PsychoPy

Note: Since this is still in beta, keep an eye out for bug fixes.

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name
Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start
When the Serial Out Component should start, see *Defining the onset/duration of components*.

Expected start (s)
If you are using frames to control timing of your stimuli, you can add an expected start time to display the component timeline in the routine.

Start type

How do you want to define your start point?

Options:

- time (s)
- frame N
- condition

Stop

When the Serial Out Component should stop, see *Defining the onset/duration of components*.

Expected duration (s)

If you are using frames to control timing of your stimuli, you can add an expected duration to display the component timeline in the routine.

Stop type

How do you want to define your end point?

Options:

- duration (s)
- duration (frames)
- time (s)
- frame N
- condition

Port

You need to know the address of the serial port you wish to write to. For more information on how to find out this address please see :ref: *this page <serial>*.

Start data

Data to be sent at start of pulse. Data will be converted to bytes, so to specify anumeric value directly use $chr(…).

Stop data

String data to be sent at end of pulse. Data will be converted to bytes, so to specify anumeric value directly use $chr(…).

Device

Information about the device associated with this Component. Keyboards, speakers, microphones, etc.

Baud rate

The baud rate, or speed, of the connection.

Data bits

Size of bits to be sent.

Stop bits

Size of bits to be sent on stop.

Parity

Parity mode.

Options:

- None

- Even
- Off
- Mark
- Space

Timeout
Time at which to give up listening for a response (leave blank for no limit)

Data

What information about this Component should be saved?

Save onset/offset times
Store the onset/offset times in the data file (as well as in the log file).

Sync timing with screen refresh
If true then the serial port will be sent synchronised to the next screen refresh, which is ideal if it should indicate the onset of a visual stimulus. If set to False then the data will be set on the serial port immediately.

Get response?
After sending a signal, should PsychoPy read and record a response from the port?

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Component
Disable this Component

Slider Component

A Slider uses mouse input to collect ratings, all sliders have the same basic structure (a line, rectangle or series of dots to indicate the range of values, several tick marks and labels, a marker) but their appearance can be varied significantly by changing the *style* parameter. For example, a *radio* style Slider features several dots and a circular marker, while a *scrollbar* style Slider features a translucent rectangle with a long marker like the page scrollbar on a website.

Categories:
Responses

Works in:
PsychoPy, PsychoJS

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name
Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start
When the Slider Component should start, see *Defining the onset/duration of components*.

Expected start (s)
If you are using frames to control timing of your stimuli, you can add an expected start time to display the component timeline in the routine.

Start type

How do you want to define your start point?

Options:

- time (s)
- frame N
- condition

Stop

When the Slider Component should stop, see *Defining the onset/duration of components*.

Expected duration (s)

If you are using frames to control timing of your stimuli, you can add an expected duration to display the component timeline in the routine.

Stop type

How do you want to define your end point?

Options:

- duration (s)
- duration (frames)
- time (s)
- frame N
- condition

Force end of Routine

Should setting a rating (releasing the mouse) cause the end of the Routine (e.g. trial)?

Styles

Discrete styles to control the overall appearance of the slider.

Options:

- slider
- rating
- radio
- scrollbar
- choice

Ticks (*if :ref:`slidercomponent-styles` isn’t ==’radio’*)

The ticks that will be place on the slider scale. The first and last ticks will be placed on the ends of the slider, and the remaining are spaced between the endpoints corresponding to their values. For example, (1, 2, 3, 4, 5) will create 5 evenly spaced ticks. (1, 3, 5) will create three ticks, one at each end and one in the middle.

Labels

The text to go with each tick (or spaced evenly across the ticks). If you give 3 labels but 5 tick locations then the end and middle ticks will be given labels. If the labels can’t be distributed across the ticks then an error will be raised. If you want an uneven distribution you should include a list matching the length of ticks but with some values set to None.

Granularity (*if :ref:`slidercomponent-styles` isn’t ==’radio’*)

Specifies the minimum step size (0 for a continuous scale, 1 for integer rating scale)

Starting value
Value of the slider befre any response, leave blank to hide the marker until clicked on

Layout

How should the stimulus be laid out on screen? Padding, margins, size, position, etc.

Size [w,h]
Size of this stimulus (either a single value or x,y pair, e.g. 2.5, [1,2]). The slider is oriented horizontally when the width is greater than the height, and oriented vertically otherwise. Default is (1.0, 0.1)

Position [x,y]
Position of this stimulus (e.g. [1,2])

Spatial units
Spatial units for this stimulus (e.g. for its *position* and *size*), see *Units for the window and stimuli* for more info.

Options:

- from exp settings
- deg
- cm
- pix
- norm
- height
- degFlatPos
- degFlat

Orientation
Orientation of this stimulus (in deg)

Options:

- -360
- 360

Flip
By default the labels will be on the bottom or left of the scale, but this can be flipped to the other side.

Appearance

How should the stimulus look? Colors, borders, styles, etc.

Label color
Color of all labels on this slider

Marker color
Color of the marker on this slider

Line color
Color of all lines on this slider

Color space
In what format (color space) have you specified the colors? See *Color spaces* for more info.

Options:

- rgb
- dkl
- lms
- hsv

Opacity
Vary the transparency, from 0.0 (invisible) to 1.0 (opaque)

Contrast
Contrast of the stimulus (1.0=unchanged contrast, 0.5=decrease contrast, 0.0=uniform/no contrast, -0.5=slightly inverted, -1.0=totally inverted)

Style tweaks
Tweaks to the style of the slider which can be applied on top of the overall style - multiple tweaks can be selected.

Options:

- labels45: Rotate all labels 45°
- triangleMarker: Replace the marker with a triangle pointing towards the line

Formatting

How should this stimulus handle text? Font, spacing, orientation, etc.

Font
What font should the text be displayed in? Locally, can be a font installed on your computer, saved to the “fonts” folder in your user folder, or the name of a Google Font. Online, can be any web safe font or a font file added to your resources list in *Experiment settings*.

Letter height
Letter height for text in labels

Data

What information about this Component should be saved?

Read only
Should participant be able to change the rating on the Slider?

Save onset/offset times
Store the onset/offset times in the data file (as well as in the log file).

Sync timing with screen refresh
Synchronize times with screen refresh (good for visual stimuli and responses based on them)

Store rating
store the rating

Store rating time
Store the time taken to make the choice (in seconds)

Store history
store the history of (selection, time)

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Component
Disable this Component

Validate with…
Name of the Validator Routine to use to check the timing of this stimulus. Options are generated live, so will vary according to your setup.

> **See also**
>
> API reference for `Slider`

Sound Component

Play recorded files or generated sounds

Categories:
Stimuli

Works in:
PsychoPy, PsychoJS

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name
Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start
When the Sound Component should start, see *Defining the onset/duration of components*.

Expected start (s)
If you are using frames to control timing of your stimuli, you can add an expected start time to display the component timeline in the routine.

Start type
How do you want to define your start point?

Options:

- time (s)
- frame N
- condition

Stop
When the Sound Component should stop, see *Defining the onset/duration of components*.

Expected duration (s)
If you are using frames to control timing of your stimuli, you can add an expected duration to display the component timeline in the routine.

Stop type

How do you want to define your end point?

Options:

- duration (s)
- duration (frames)
- time (s)
- frame N
- condition

Sound

A sound can be a note name (e.g. A or Bf), a number to specify Hz (e.g. 440) or a filename.

Sync start with screen

A reaction time to a sound stimulus should be based on when the screen flipped

Device

Information about the device associated with this Component. Keyboards, speakers, microphones, etc.

Device label

A label to refer to this Component's associated hardware device by. If using the same device for multiple components, be sure to use the same label here.

Speaker

What speaker to play this sound on

Resample

If the sample rate of a clip doesn't match the sample rate of the speaker, should we resample it to match?

Latency/exclusivity mode

How should PsychoPy handle latency? Some options will result in other apps being denied access to the speaker while your experiment is running.

Options:

- Shared: Don't take exclusive control over the speaker, so other apps can still use it. Send sounds via the system mixer so that sample rates are all handled, even though this introduces latency.
- Shared low-latency: Don't take exclusive control over the speaker, so other apps can still use it. Send sounds directly to reduce latency, so sounds will need to match the sample rate of the speaker. **Recommended in most cases - if :ref:`soundcomponent-resample` is checked then sample rates are already handled on load!**
- Exclusive low-latency: Take exclusive control over the speaker, so other apps can't use it. Send sounds directly to reduce latency, so sounds will need to be the same sample rate as one another, but this can be any sample rate supported by the speaker.
- Exclusive aggressive low-latency (with fallback): Take exclusive control over the speaker, so other apps can't use it. Send sounds directly to reduce latency, so sounds will need to be the same sample rate as one another, but this can be any sample rate supported by the speaker. Force the system to prioritise resources towards playing sounds on this speaker for absolute minimum latency, but fallback to mode 2 if the system rejects this.
- Exclusive aggressive low-latency (no fallback): Take exclusive control over the speaker, so other apps can't use it. Send sounds directly to reduce latency, so sounds will need to be the same sample rate as one another, but this can be any sample rate supported by the speaker. Force the system to prioritise resources towards playing sounds on this speaker for absolute minimum latency, and raise an error if the system rejects this.

Playback

Volume
The volume with which the sound should be played. It's a normalized value between 0 (minimum) and 1 (maximum).

Hamming window
For tones we can apply a hamming window to prevent 'clicks' that are caused by a sudden onset. This delays onset by roughly 1ms.

Stop with Routine?
Should playback cease when the Routine ends? Untick to continue playing after the Routine has finished.

Force end of Routine
Should the end of the sound cause the end of the Routine (e.g. trial)?

Data

What information about this Component should be saved?

Save onset/offset times
Store the onset/offset times in the data file (as well as in the log file).

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Component
Disable this Component

Validate with…
Name of the Validator Routine to use to check the timing of this stimulus. Options are generated live, so will vary according to your setup.

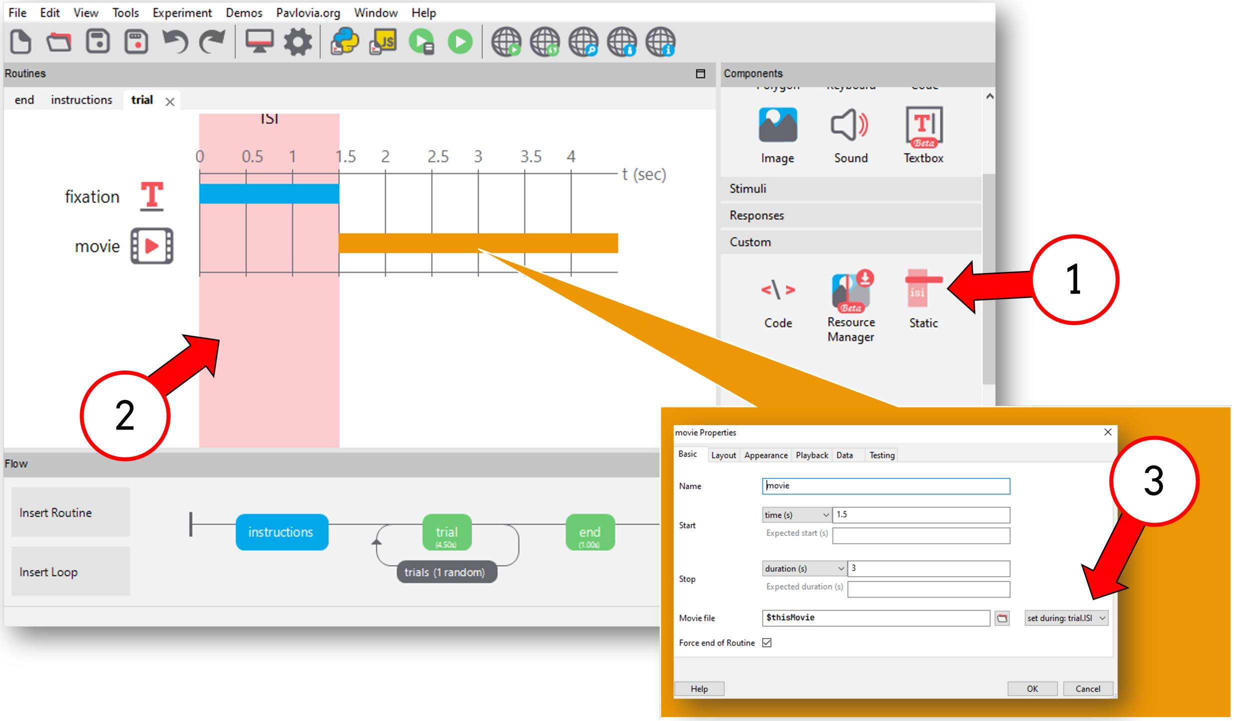
Static Component

The Static Component allows you to have a period where you can preload images or perform other time-consuming operations that not be possible while the screen is being updated. Static periods are also particularly useful for *online* studies to decrease the time taken to load resources at the start (see also *Resource Manager Component*).

> **Note**
>
> For online studies, if you use a static component this will override the resources loaded at the beginning via Experiment settings > Online > Additional resources. You might therefore want to combine a static period with a *Resource Manager Component* to make sure that all resources your study needs will be loaded and available for the experiment.

Typically a static period would be something like an inter-trial or inter-stimulus interval (ITI/ISI). During this period you should not have any other objects being presented that are being updated (this isn't checked for you - you have to make that check yourself), but you can have components being presented that are themselves static. For instance a fixation point never changes and so it can be presented during the static period (it will be presented and left on-screen while the other updates are being made).

Any stimulus updates can be made to occur during any static period defined in the experiment (it does not have to be in the same Routine). This is done in the updates selection box- once a static period exists it will show up here as well as the standard options of *constant* and *every repeat* etc. Many parameter updates (e.g. orientation are made so quickly



Name: movie

Start: time (s) 1.5

Expected start (s)

Stop: duration (s) 3

Expected duration (s)

Movie file: $thisMovie — set during: trial.ISI

Force end of Routine ☑

Fig. 5.2: How to use a static component. 1) To use a static component first select it from the component panel. 2) highlights in red the time window you are treating as "static". If you click on the red highlighted window you can edit the static component. 3) To use the static window to load a resource, select the component where the resource will be load, and in the dropdown window choose "set during:trial.ISI" - here "trial" refers to the routine where the static component is and "ISI" refers to the name of the static component.

that using the static period is of no benefit but others, most notably the loading of images from disk, can take substantial periods of time and these should always be performed during a static period to ensure good timing.

If the updates that have been requested were not completed by the end of the static period (i.e. there was a timing overshoot) then you will receive a warning to that effect. In this case you either need a longer static period to perform the actions or you need to reduce the time required for the action (e.g. use an image with fewer pixels).

Categories:
Custom

Works in:
PsychoPy, PsychoJS

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name
Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start
When the Static Component should start, see *Defining the onset/duration of components*.

Expected start (s)
If you are using frames to control timing of your stimuli, you can add an expected start time to display the component timeline in the routine.

Start type
How do you want to define your start point?

Options:

- time (s)
- frame N
- condition

Stop
When the Static Component should stop, see *Defining the onset/duration of components*.

Expected duration (s)
If you are using frames to control timing of your stimuli, you can add an expected duration to display the component timeline in the routine.

Stop type
How do you want to define your end point?

Options:

- duration (s)
- duration (frames)
- time (s)
- frame N
- condition

Data

What information about this Component should be saved?

Save onset/offset times
Store the onset/offset times in the data file (as well as in the log file).

Sync timing with screen refresh
Synchronize times with screen refresh (good for visual stimuli and responses based on them)

Custom

Parameters for injecting custom code

Custom code
After running the component updates (which are defined in each component, not here) any code inserted here will also be run

Save data during
While the frame loop is paused, should we take the opportunity to save data now? This is only relevant locally, online data saving is either periodic or on close.

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Component
Disable this Component

> **See also**
>
> API reference for `StaticPeriod`

Text Component

This component can be used to present text to the participant, either instructions or stimuli. It differs from *Textbox Component* in that changing the size of the stimulus changes the size of the text itself, not the box containing the text.

Categories:
Stimuli

Works in:
PsychoPy, PsychoJS

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name
Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start
When the Text Component should start, see *Defining the onset/duration of components*.

Expected start (s)

If you are using frames to control timing of your stimuli, you can add an expected start time to display the component timeline in the routine.

Start type

How do you want to define your start point?

Options:

- time (s)
- frame N
- condition

Stop

When the Text Component should stop, see *Defining the onset/duration of components*.

Expected duration (s)

If you are using frames to control timing of your stimuli, you can add an expected duration to display the component timeline in the routine.

Stop type

How do you want to define your end point?

Options:

- duration (s)
- duration (frames)
- time (s)
- frame N
- condition

Text

The text to be displayed

Layout

How should the stimulus be laid out on screen? Padding, margins, size, position, etc.

Position [x,y]

Position of this stimulus (e.g. [1,2])

Spatial units

Spatial units for this stimulus (e.g. for its *position* and size), see *Units for the window and stimuli* for more info.

Options:

- from exp settings
- deg
- cm
- pix
- norm
- height
- degFlatPos
- degFlat

Orientation
Orientation of this stimulus (in deg)

Options:

- -360
- 360

Draggable?
Should this stimulus be moveble by clicking and dragging?

Wrap width
How wide should the text get when it wraps? (in the specified units)

Flip (mirror)
Whether to mirror-reverse the text: 'horiz' for left-right mirroring, 'vert' for up-down mirroring. The flip can be set dynamically on a per-frame basis by using a variable, e.g., $mirror, as defined in a code component or conditions file and set to either 'horiz' or 'vert'.

Options:

- horiz
- vert
- None

Appearance

How should the stimulus look? Colors, borders, styles, etc.

Text color
Color of the text (e.g. $[1,1,0], red)

Color space
In what format (color space) have you specified the colors? See *Color spaces* for more info.

Options:

- rgb
- dkl
- lms
- hsv

Opacity
Vary the transparency, from 0.0 (invisible) to 1.0 (opaque)

Contrast
Contrast of the stimulus (1.0=unchanged contrast, 0.5=decrease contrast, 0.0=uniform/no contrast, -0.5=slightly inverted, -1.0=totally inverted)

Formatting

How should this stimulus handle text? Font, spacing, orientation, etc.

Font
What font should the text be displayed in? Locally, can be a font installed on your computer, saved to the "fonts" folder in your user folder, or the name of a Google Font. Online, can be any web safe font or a font file added to your resources list in *Experiment settings*.

Letter height

The height of the characters in the given units of the stimulus/window. Note that nearly all actual letters will occupy a smaller space than this, depending on font, character, presence of accents etc. The width of the letters is determined by the aspect ratio of the font.

Language style

Handle right-to-left (RTL) languages and Arabic reshaping

Options:

- LTR
- RTL
- Arabic

Data

What information about this Component should be saved?

Save onset/offset times

Store the onset/offset times in the data file (as well as in the log file).

Sync timing with screen refresh

Synchronize times with screen refresh (good for visual stimuli and responses based on them)

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Component

Disable this Component

Validate with…

Name of the Validator Routine to use to check the timing of this stimulus. Options are generated live, so will vary according to your setup.

See also

API reference for *`TextStim`*

Textbox Component

This component can be used either to present text to the participant, or to allow free-text answers via the keyboard. It differs from *Text Component* in that text is wrapped within a container.

Categories:

Stimuli, Responses

Works in:

PsychoPy, PsychoJS

Note: Since this is still in beta, keep an eye out for bug fixes.

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name

Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start

When the Textbox Component should start, see *Defining the onset/duration of components*.

Expected start (s)

If you are using frames to control timing of your stimuli, you can add an expected start time to display the component timeline in the routine.

Start type

How do you want to define your start point?

Options:

- time (s)
- frame N
- condition

Stop

When the Textbox Component should stop, see *Defining the onset/duration of components*.

Expected duration (s)

If you are using frames to control timing of your stimuli, you can add an expected duration to display the component timeline in the routine.

Stop type

How do you want to define your end point?

Options:

- duration (s)
- duration (frames)
- time (s)
- frame N
- condition

Editable?

Should textbox be editable?

Text

The text to be displayed

Placeholder text (*if :ref:`textboxcomponent-editable` is checked*)

Placeholder text to show when there is no text contents.

Layout

How should the stimulus be laid out on screen? Padding, margins, size, position, etc.

Size [w,h]

Size of this stimulus (either a single value or x,y pair, e.g. 2.5, [1,2]).

Note: This is the size of the box, not the text!

Position [x,y]

Position of this stimulus (e.g. [1,2])

Padding
Defines the space between text and the textbox border

Spatial units
Spatial units for this stimulus (e.g. for its *position* and *size*), see *Units for the window and stimuli* for more info.

Options:

- from exp settings
- deg
- cm
- pix
- norm
- height
- degFlatPos
- degFlat

Anchor
Which point in this stimulus should be anchored to the point specified by *Position [x,y]*?

Options:

- center
- top-center
- bottom-center
- center-left
- center-right
- top-left
- top-right
- bottom-left
- bottom-right

Orientation
Orientation of this stimulus (in deg)

Options:

- -360
- 360

Flip horizontal
Whether to mirror-reverse the text horizontally (left-right mirroring)

Flip vertical
Whether to mirror-reverse the text vertically (top-bottom mirroring)

Draggable?
Should this stimulus be moveble by clicking and dragging?

Overflow
If the text is bigger than the textbox, how should it behave?

Options:

- visible: Show the overflowing text as it flows past the bottom
- scroll: Show a scrollbar to view overflowing text
- hidden: Hide overflowing text

Appearance

How should the stimulus look? Colors, borders, styles, etc.

Text color
Color of the text within the box (e.g. $[1,1,0], red)

Fill color
Fill color of this stimulus (e.g. $[1,1,0], red)

Border color
Border color of this stimulus (e.g. $[1,1,0], red)

Color space
In what format (color space) have you specified the colors? See *Color spaces* for more info.

Options:

- rgb
- dkl
- lms
- hsv

Opacity
Vary the transparency, from 0.0 (invisible) to 1.0 (opaque)

Border width
Textbox border width

Contrast
Contrast of the stimulus (1.0=unchanged contrast, 0.5=decrease contrast, 0.0=uniform/no contrast, -0.5=slightly inverted, -1.0=totally inverted)

Speech point [x,y]
If specified, adds a speech bubble tail going to that point on screen.

Formatting

How should this stimulus handle text? Font, spacing, orientation, etc.

Font
What font should the text be displayed in? Locally, can be a font installed on your computer, saved to the "fonts" folder in your user folder, or the name of a Google Font. Online, can be any web safe font or a font file added to your resources list in *Experiment settings*.

Letter height
The height of the characters in the given units of the stimulus/window. Note that nearly all actual letters will occupy a smaller space than this, depending on font, character, presence of accents etc. The width of the letters is determined by the aspect ratio of the font.

Line spacing
Defines the space between lines, proportional to the size of the font

Bold

Should text be bold?

Italic

Should text be italic?

Language style

Handle right-to-left (RTL) languages and Arabic reshaping

Options:

- LTR
- RTL
- Arabic

Alignment

How should text be laid out within the box?

Options:

- center
- top-center
- bottom-center
- center-left
- center-right
- top-left
- top-right
- bottom-left
- bottom-right

Data

What information about this Component should be saved?

Save onset/offset times

Store the onset/offset times in the data file (as well as in the log file).

Sync timing with screen refresh

Synchronize times with screen refresh (good for visual stimuli and responses based on them)

Auto log

Automatically record all changes to this in the log file

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Component

Disable this Component

Validate with…

Name of the Validator Routine to use to check the timing of this stimulus. Options are generated live, so will vary according to your setup.

> **See also**
>
> API reference for `TextBox`

Unknown Component

Unknown Component is the placeholder for any Component which PsychoPy doesn't recognise (and which doesn't declare itself as coming from a plugin, as in *Unknown Plugin Component*). If you see an Unknown Component, there's two most likely explanations:

- **This experiment was built in an older version of PsychoPy, and this Component has since been removed**. Usually, when we remove a Component from PsychoPy, it's either because we've moved it out to a plugin, or there's a newer Component which covers everything it does and does so in a better way. In this case, you should search online for the modern equivalent of the removed Component (try the PsychoPy forum!)
- **This experiment was built in a newer version of PsychoPy, and this Component wasn't added yet in your version**. This one's an easy fix, just update to the latest version! Or, at least, the version in which that Component was added. You can see what Components were added when via PsychoPy's GitHub release notes.

Unknown Plugin Component

Similar to *Unknown Component*, Unknown Plugin Components are Components which PsychoPy doesn't recognise, but which declare themselves as coming from a particular plugin. This means you need to install the plugin for PsychoPy to properly recognise and make use of the Component.

Thankfully, installing the necessary plugin is easy! Just open up the Component and click the link at the bottom.

Variable Component

A variable holds a value which you can refer to by name later in the experiment.

Categories:
Custom

Works in:
PsychoPy

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name
Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces). The name of the Component will be the name of the variable!

Start
The time or condition from when you want your variable to be defined. The default value is None, and so will be defined at the beginning of the experiment, trial or frame., see *Defining the onset/duration of components*.

Expected start (s)
If you are using frames to control timing of your stimuli, you can add an expected start time to display the component timeline in the routine.

Start type
How do you want to define your start point?

Options:

- time (s)
- frame N
- condition

Stop
The duration for which the variable is defined/updated, see *Defining the onset/duration of components*.

Expected duration (s)
If you are using frames to control timing of your stimuli, you can add an expected duration to display the component timeline in the routine.

Stop type
How do you want to define your end point?

Options:

- duration (s)
- duration (frames)
- time (s)
- frame N
- condition

Experiment start value
The start value. A variable can be set to any value.

Routine start value
Set the value for the beginning of each Routine.

Frame start value
Set the value for the beginning of every screen refresh.

Data

What information about this Component should be saved?

Save exp start value
Save the experiment start value in data file.

Save Routine start value
Choose whether or not to save the experiment start value to your data file.

Save frame value
Frame values are contained within a list for each trial, and discarded at the end of each trial. Choose whether or not to take the first, last or average variable values from the frame container, and save to your data file.

Options:

- first
- last
- all
- never

Save Routine end value
Choose whether or not to save the routine end value to your data file.

Save exp end value
Choose whether or not to save the experiment end value to your data file.

Save onset/offset times
Store the onset/offset times in the data file (as well as in the log file).

Sync timing with screen refresh
Synchronize times with screen refresh (good for visual stimuli and responses based on them)

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Component
Disable this Component

Visual Validator Routine

Use a light sensor to confirm that visual stimuli are presented when they should be.

Categories:
Validation

Works in:
PsychoPy

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name
Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Find best threshold?
Run a brief Routine to find the best threshold for the light sensor at experiment start?

Threshold (*if :ref: `visualvalidatorroutine-findthreshold` isn't ==True*)
Light threshold at which the light sensor should register a positive, units go from 0 (least light) to 255 (most light).

Find diode?
Run a brief Routine to find the size and position of the light sensor at experiment start?

Position [x,y] (*if :ref: `visualvalidatorroutine-finddiode` isn't ==True*)
Position of the light sensor on the window.

Size [x,y] (*if :ref: `visualvalidatorroutine-finddiode` isn't ==True*)
Size of the area covered by the light sensor on the window.

Spatial units (*if :ref: `visualvalidatorroutine-finddiode` isn't ==True*)
Spatial units in which the light sensor size and position are specified.

Options:

- from exp settings
- deg
- cm
- pix
- norm

- height
- degFlatPos
- degFlat

Device

Information about the device associated with this Component. Keyboards, speakers, microphones, etc.

Device name
A name to refer to this Component's associated hardware device by. If using the same device for multiple components, be sure to use the same name here.

Light sensor type
Type of light sensor to use.

Light sensor channel
If relevant, a channel number attached to the light sensor, to distinguish it from other light sensors on the same port. Leave blank to use the first light sensor which can detect the Window.

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Routine
Disable this Routine

Voice Key Component

Voice Key: Get input from a microphone as simple true/false values

Categories:
Responses

Works in:
PsychoPy

Note: Since this is still in beta, keep an eye out for bug fixes.

Parameters

Basic

The required attributes of the stimulus, controlling its basic function and behaviour

Name
Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start
When the Voice Key Component should start, see *Defining the onset/duration of components*.

Expected start (s)
If you are using frames to control timing of your stimuli, you can add an expected start time to display the component timeline in the routine.

Start type
How do you want to define your start point?

Options:

- time (s)
- frame N
- condition

Stop
When the Voice Key Component should stop, see *Defining the onset/duration of components*.

Expected duration (s)
If you are using frames to control timing of your stimuli, you can add an expected duration to display the component timeline in the routine.

Stop type
How do you want to define your end point?

Options:

- duration (s)
- duration (frames)
- time (s)
- frame N
- condition

Force end of Routine
Should a response force the end of the Routine (e.g end the trial)?

Device

Information about the device associated with this Component. Keyboards, speakers, microphones, etc.

Device label
A label to refer to this Component's associated hardware device by. If using the same device for multiple components, be sure to use the same label here.

Device backend
What kind of voicekey is it? What package/plugin should be used to talk to it?

Microphone (*if :ref:`voicekeycomponent-devicebackend` is "Microphone"*)
What microphone device to take volume readings from?

Threshold (0-255) (*if :ref:`voicekeycomponent-devicebackend` is "Microphone"*)
Threshold volume (0 for min value in dB range, 255 for max value) above which to register a voicekey response

Decibel range (*if :ref:`voicekeycomponent-devicebackend` is "Microphone"*)
What kind of values (dB) would you expect to receive from this device? In other words, how many dB does a threshold of 0 and of 255 correspond to?

Sampling window (s) (*if :ref:`voicekeycomponent-devicebackend` is "Microphone"*)
How many seconds to average volume readings across? Bigger windows are less precise, but also less subject to random noise.

Data

What information about this Component should be saved?

Register button press on…
When should the response be registered? When the sound starts, or when it stops?

Options:

- Press
- Release

Store
Choose which (if any) responses to store at the end of a trial

Options:

- Last response
- First response
- All responses
- Nothing

Store correct
Do you want to save the response as correct/incorrect?

Correct answer (*if :ref:`voicekeycomponent-storecorrect` is checked*)
What is the 'correct' response (True/False)? Might be helpful to add a correctAns column and use $correctAns to compare to the response.

Save onset/offset times
Store the onset/offset times in the data file (as well as in the log file).

Sync timing with screen refresh
Synchronize times with screen refresh (good for visual stimuli and responses based on them)

Testing

Tools for testing, debugging and checking the performance of this Component.

Disable Component
Disable this Component

About Components

Routines in the Builder contain any number of components, which typically define the parameters of a stimulus or an input/output device.

Entering parameters

Most of the entry boxes for Component parameters simply receive text or numeric values or lists (sequences of values surrounded by square brackets) as input. In addition, the user can insert variables and code into most of these, which will be interpreted either at the beginning of the experiment or at regular intervals within it.

To indicate to that the value represents a variable or python code, rather than literal text, it should be preceded by a *$*. For example, inserting *intensity* into the text field of the Text Component will cause that word literally to be presented, whereas *$intensity* will cause python to search for the variable called intensity in the script.

Variables associated with *Loops* can also be entered in this way (see *"Using loops to update stimuli trial-by-trial"* for further details). But it can also be used to evaluate arbitrary python code.

For example:

- `$random(2)` will generate a pair of random numbers
- `$"yn"[randint(2)]` will randomly choose the first or second character (y or n)
- `$globalClock.getTime()` will insert the current time in secs of the globalClock object
- `$[sin(angle), cos(angle)]` will insert the sin and cos of an angle (e.g. into the x,y coords of a stimulus)

How often to evaluate the variable/code

If you do want the parameters of a stimulus to be evaluated by code in this way you need also to decide how often it should be updated. By default, the parameters of Components are set to be *constant*; the parameter will be set at the beginning of the experiment and will remain that way for the duration. Alternatively, they can be set to change either on *every repeat* in which case the parameter will be set at the beginning of the Routine on each repeat of it. Lastly many parameters can even be set *on every frame*, allowing them to change constantly on every refresh of the screen.

5.1.9 Components which have moved to plugins

The following plugins were once a part of core PsychoPy, but have since moved out to various plugins:

Emotiv Marking Component

The Emotiv Marking Component Component is now located within the psychopy-emotiv plugin.

Emotiv Record Component

The Emotiv Record Component Component is now located within the psychopy-emotiv plugin.

ioLabs Response Box Component

The ioLabs Response Box Component is now located within the psychopy-iolabs plugin.

Patch Component

The Patch Component is now located within the psychopy-legacy plugin.

Pump Component

The Pump Component is now located within the psychopy-qmix plugin.

Rating Scale Component

The Rating Scale Component is now located within the psychopy-legacy plugin.

Reward Component

The Reward Component is now located within the psychopy-labeotech plugin.

5.1.10 Experiment settings

The settings menu can be accessed by clicking the icon at the top of the window. It allows the user to set various aspects of the experiment, such as the size of the window to be used or what information is gathered about the subject and determine what outputs (data files) will be generated.

Settings

Basic

Experiment name
A name that will be stored in the metadata of the data file.

Use PsychoPy version
Which version of was the task created in? if you are using a more recently installed version of this can compile using an archived, older version to run previously created tasks.

Show info dlg
If this box is checked then a dialog will appear at the beginning of the experiment allowing the *Experiment Info* to be changed.

Enable escape
If ticked then the *Esc* key can be used to exit the experiment at any time (even without a keyboard component)

Experiment Info
This information will be presented in a dialog box at the start and will be saved with any data files and so can be used for storing information about the current run of the study. The information stored here can also be used within the experiment. For example, if the *Experiment Info* included a field called *ori* then Builder components could access expInfo['ori'] to retrieve the orientation set here. Obviously this is a useful way to run essentially the same experiment, but with different conditions set at run-time. If you are running a study online, we recommend keeping the field "participant" because this is used to name data output files.

Screen

Monitor
The name of the monitor calibration. Must match one of the monitor names from *Monitor Center*.

Screen:
If multiple screens are available (and if the graphics card is *not* an intel integrated graphics chip) then the user can choose which screen they use (e.g. 1 or 2).

Full-screen window:
If this box is checked then the experiment window will fill the screen (overriding the window size setting and using the size that the screen is currently set to in the operating system settings).

Window size:
The size of the window in pixels, if this is not to be a full-screen window.

Units
The default units of the window (see *Units for the window and stimuli*). These can be overridden by individual components.

Audio

Audio library
Choice of audio library to use to present sound, default uses preferences (see *Preferences*).

Audio latency priority
Latency mode for PsychToolbox audio (see *Preferences*) (because this applies to the PTB sound backend, this only applies for local, not online studies)

Force stereo
Force audio to stereo (2-channel) output

Online

Output path
Where to export the compiled javascript experiment and associated html files. (note that in earlier versions of this was *html* by default, this is not necissary as it will duplicate your resources, associated discourse threads with this suggestion might now be outdated)

Export html
When to export a html file and compile a javascript version of the experiment. This is on sync by default, meaning these files will be generated when a project is pushed/synced to . Alternatively this can be "on save" or "manually" the latter might be used if you are making manual edits to the exported javascript file, though this is not recommended as changes will not be reflected back in your builder file.

Completed URL
The URL to direct participants to upon completion (when they select "OK" in the green thank-you message online)

Incomplete URL
The URL to direct participants to if they exit the task early (e.g. by pressing the escape key).

Additional resources
Resources that your task will require (e.g. image files, excel sheets). Note that will attempt to populate this automatically, though if you encounter an "Unknown resource" error online, it is possible that you need to add resources to this list.

Eyetracking

Eyetracker Device
Specify what kind of eye tracker you are using. If you are creating your paradigm out-of-lab (i.e. with no eye tracker) we suggest using MouseGaze, which will use your mouse to simulate eye movements and blinks. Alternatively, you can select which device you are currently using and set-up those parameters (see *ioHub Common Eye Tracker Interface*)

Data

Data filename:
A *formatted string* to control the base filename and path, often based on variables such as the date and/or the participant. This base filename will be given the various extensions for the different file types as needed. Examples:

```
# all in data folder relative to experiment file: data/JWP_memoryTask_2014_Feb_15_
↪1648
'data/%s_%s_%s' %(expInfo['participant'], expName, expInfo['date'])

# group by participant folder: data/JWP/memoryTask-2014_Feb_15_1648
'data/%s/%s-%s' %(expInfo['participant'], expName, expInfo['date'])

# put into dropbox: ~/dropbox/data/memoryTask/JWP-2014_Feb_15_1648
# os.path.expanduser replaces '~' with the path to your home directory,
# os.path.join joins the path components together correctly, regardless of OS
# os.path.relpath creates a relative path between the specified path and the␣
↪current directory
'$os.path.relpath(os.path.join(os.path.expanduser('~'), 'dropbox', 'data', expName,␣
↪expInfo['participant'] + '-' + expInfo['date']))
```

Data file delimiter
What delimiter should your data file use to separate the columns

Save Excel file
If this box is checked an Excel data file (.xlsx) will be stored.

Save csv file (summaries)
If this box is checked a summary file will be created with one row corresponding to the entire loop. If a keyboard response is used the mean and dtandard deviations of responses across trials will also be stored.

> **Note**
>
> Up to 2024.2.4, the summary file naming convention is `rf"{filename}{loop_name}.csv"`. From 2025.1.0 onwards, the new naming convention is `rf"{filename}_{loop_name}.csv"` to be consistent with uses of suffix

throughout PsychoPy. If you are using analysis scripts based on versions prior to 2025.1.0, please be aware of this change and adjust file paths accordingly.

Save csv file (trial-by-trial)
If this box is checked a comma separated variable (.csv) will be stored. Each trial will be stored as a new row.

Save psydat file
If this box is checked a *data file (.psydat)* will be stored. This is a Python specific format (.pickle files) which contains more information that .xlsx or .csv files that can be used with data analysis and plotting scripts written in Python. Whilst you may not wish to use this format it is recommended that you always save a copy as it contains a complete record of the experiment at the time of data collection.

Save hdf5 file
If this box is checked data will be stored to a hdf5 file, this is mainly applicable if a component is implemented that requires a complex data structure e.g. eyetracking.

Save log file
A log file provides a record of what occurred during the experiment in chronological order, including information about any errors or warnings that may have occurred.

Logging level
How much detail do you want to be output to the log file, if it is being saved. The lowest level is *error*, which only outputs error messages; *warning* outputs warnings and errors; *info* outputs all info, warnings and errors; *debug* outputs all info that can be logged. This system enables the user to get a great deal of information while generating their experiments, but then reducing this easily to just the critical information needed when actually running the study. If your experiment is not behaving as you expect it to, this is an excellent place to begin to work out what the problem is.

5.1.11 Defining the onset/duration of components

As of version 1.70.00, the onset and offset times of stimuli can be defined in several ways.

Start and stop times can be entered in terms of seconds (*time (s)*), by frame number (*frameN*) or in relation to another stimulus (*condition*). *Condition* would be used to make *Components* start or stop depending on the status of something else, for example when a sound has finished. Duration can also be varied using a *Code Component*.

If you need very precise timing (particularly for very brief stimuli for instance) then it is best to control your onset/duration by specifying the number of frames the stimulus will be presented for.

Measuring duration in seconds (or milliseconds) is not very precise because it doesn't take into account the fact that your monitor has a fixed frame rate. For example if the screen has a refresh rate of 60Hz you cannot present your stimulus for 120ms; the frame rate would limit you to 116.7ms (7 frames) or 133.3ms (8 frames). The duration of a frame (in seconds) is simply 1/refresh rate in Hz.

Condition would be used to make *Components* start or stop depending on the status of something else, for example when a movie has finished. Duration can also be varied using a code component.

In cases where cannot determine the start/endpoint of your Component (e.g. because it is a variable) you can enter an 'Expected' start/duration. This simply allows components with variable durations to be drawn in the Routine window. If you do not enter the approximate duration it will not be drawn, but this will not affect experimental performance.

For more details of how to achieve good temporal precision see *Timing Issues and synchronisation*

Examples

- Use *time(s)* or *frameN* and simply enter numeric values into the start and duration boxes.
- Use *time(s)* or *frameN* and enter a numeric value into the start time and set the duration to a variable name *by preceding it with a $*. Then set *expected time* to see an approximation in your *routine*

- Use condition to cause the stimulus to start immediately after a movie component called myMovie, by entering *$myMovie.status==FINISHED* into the *start* time.

5.1.12 Generating outputs (datafiles)

There are 4 main forms of *output* file from :

- Excel 2007 files (.xlsx) see *Excel Data Files* for more details
- text data files (.csv, .tsv, or .txt) see *Delimited Text Files* for more details
- binary data files (.psydat) see *|PsychoPy| Data Files* for more details
- log files (.log) see *Log Files* for more details

5.1.13 Common Mistakes (aka Gotcha's)

General Advice

- Python and therefore is CASE SENSITIVE
- To use a dollar sign ($) for anything other than to indicate a code snippet for example in a text, precede it with a backslash `\$` (the backslash won't be printed)
- Have you entered your the settings for your *monitor*? If you are using degrees as a unit of measurement and have not entered your monitor settings, the size of stimuli will not be accurate.
- If your experiment is not behaving in the way that you expect. Have you looked at the *log file*? This can point you in the right direction. Did you know you can change the type of information that is stored in the log file in preferences by changing the *logging level*.
- Have you tried compiling the script and running it. Does this produce a particular error message that points you at a particular problem area? You can also change things in a more detailed way in the coder view and if you are having problems, reading through the script can highlight problems. Reading a compiled script can also help with the creation of a *Code Component*

My stimulus isn't appearing, there's only the grey background

- Have you checked the size of your stimulus? If it is 0.5x0.5 pixels you won't be able to see it!
- Have you checked the position of your stimulus? Is it positioned off the screen?

The loop isn't using my Excel spreadsheet

- Have you remembered to specify the file you want to use when setting up the loop?
- Have you remembered to add the variables proceeded by the $ symbol to your stimuli?

I just want a plain square, but it's turning into a grating

- If you don't want your stimulus to have a texture, you need Image to be None

The code snippet I've entered doesn't do anything

- Have you remembered to put a $ symbol at the beginning (this isn't necessary, and should be avoided in a *Code Component*)?
- A dollar sign as the first character of a line indicates to that the rest of the line is code. It does not indicate a variable name (unlike in perl or php). This means that if you are, for example, using variables to determine position, enter $[x,y]. The temptation is to use [$x,$y], which will not work.

My stimulus isn't changing as I progress through the loop

- Have you changed the setting for the variable that you want to change to 'change every repeat' (or 'change every frame')?

I'm getting the error message AttributeError: 'unicode object has no attribute 'XXXX'

- This type of error is usually caused by a naming conflict. Whilst we have made every attempt to make sure that these conflicts produce a warning message it is possible that they may still occur.
- The most common source of naming conflicts in an external file which has been imported to be used in a loop i.e. .xlsx, .csv.
- Check to make sure that all of the variable names are unique. There can be no repeated variable names anywhere in your experiment.

The window opens and immediately closes

- Have you checked all of your variable entries are accepted commands e.g. gauss but not Gauss
- If you compile your experiment and run it from the coder window what does the error message say? Does it point you towards a particular variable which may be incorrectly formatted?

If you are having problems getting the application to run please see *Troubleshooting*

5.1.14 Compiling a Script

If you click the *compile script* icon this will display the script for your experiment in the *Coder* window.

This can be used for debugging experiments, entering small amounts of code and learning a bit about writing scripts amongst other things.

The code is fully commented and so this can be an excellent introduction to writing your own code.

5.1.15 Set up your monitor properly

It's a really good idea to tell about the set up of your monitor, especially the size in cm and pixels and its distance, so that can present your stimuli in units that will be consistent in another lab with a different set up (e.g. cm or degrees of visual angle).

You should do this in *Monitor Center* which can be opened from Builder by clicking on the icon that shows two monitors. In *Monitor Center* you can create settings for multiple configurations, e.g. different viewing distances or different physical devices and then select the appropriate one by name in your experiments or scripts.

Having set up your monitor settings you should then tell which of your monitor setups to use for this experiment by going to the *Experiment settings* dialog.

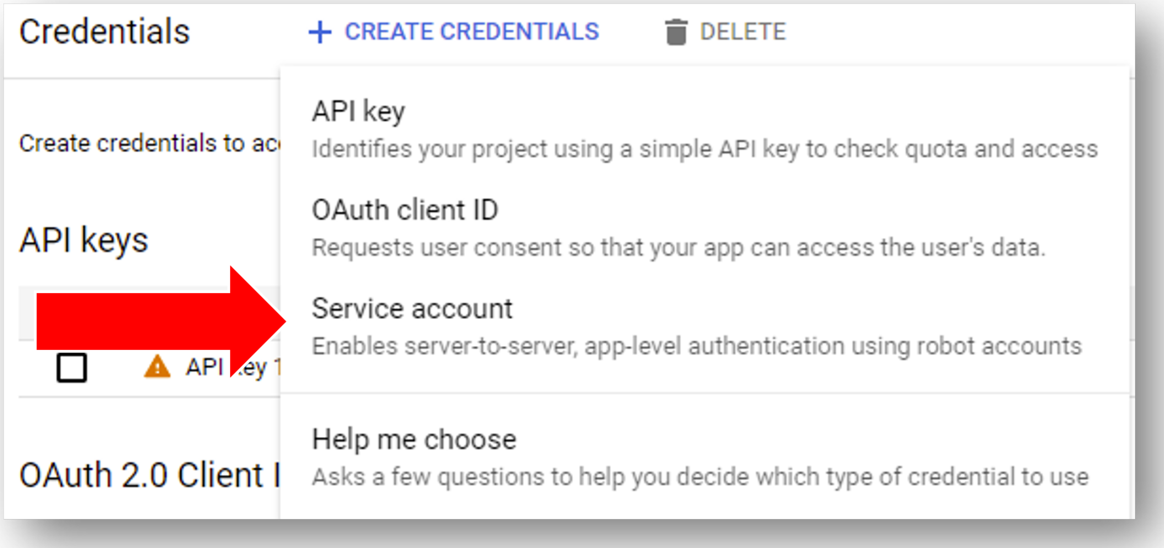
5.1.16 Creating a Google Cloud Speech API key

There are a few steps but they're relatively easy. Pricing is free for the first 60 minutes per month and 1-2cents per minute after that Information here: https://cloud.google.com/speech-to-text

Note that You might be asked to enter card details but you are not charged an auto update unless you manually enter the card details when prompted

Steps

- Create an account on Google Cloud Platform (this is not the same as simply gmail or Google Worksuite)
- Create a project from here: https://console.cloud.google.com/home/dashboard by selecting manage resources > create project The projects could just be for the entire lab, say, or for each experiment, depending on the granularity you need for billing (We believe)
- Enable the Speech API for that project: select the project in the manage resources page, go to https://console.cloud.google.com/apis/library/speech.googleapis.com click "enable".
- Then click on Credentials and create Service Account credentials.



API key
Identifies your project using a simple API key to check quota and access

OAuth client ID
Requests user consent so that your app can access the user's data.

Service account
Enables server-to-server, app-level authentication using robot accounts

Help me choose
Asks a few questions to help you decide which type of credential to use

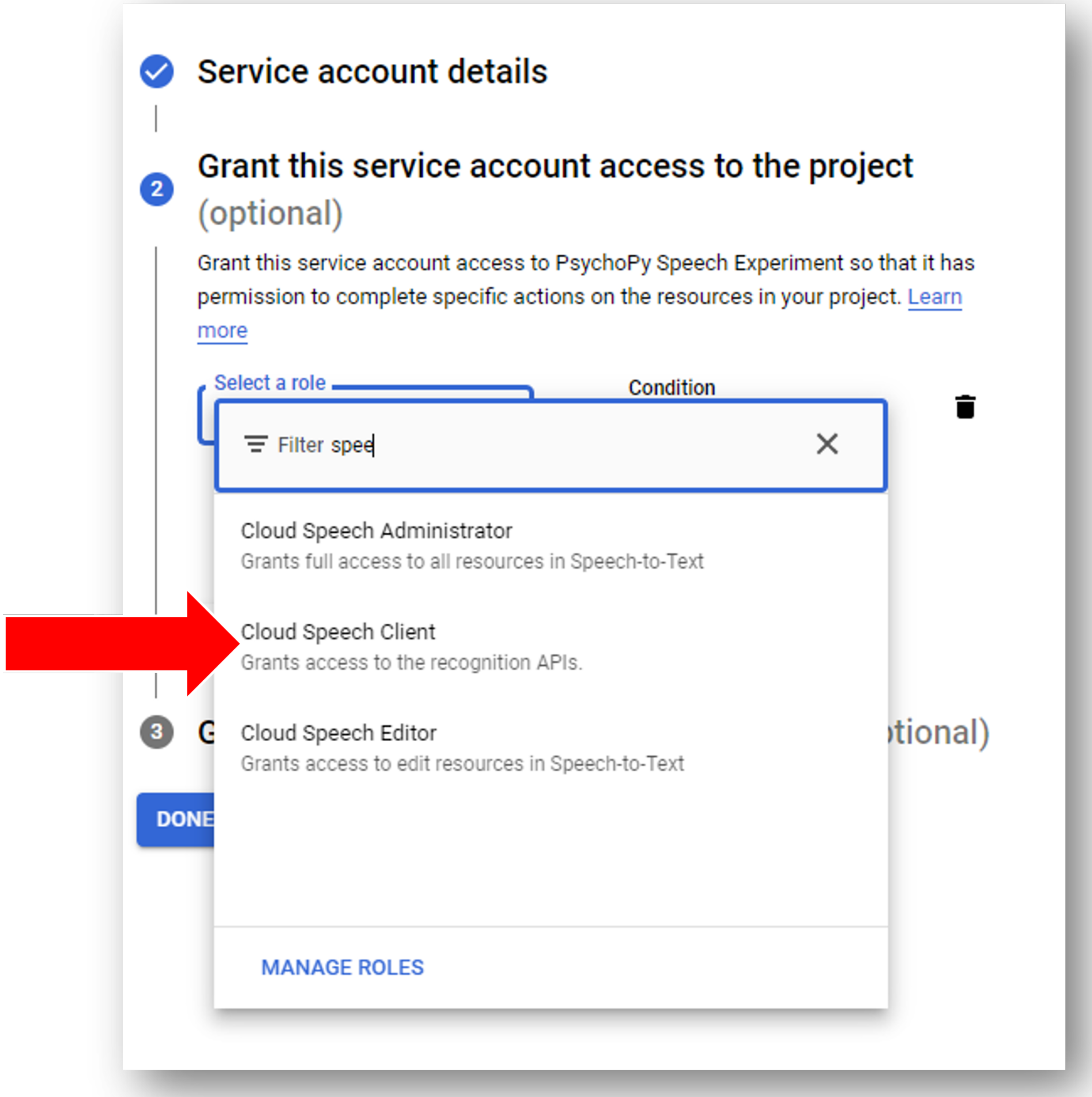
Fig. 5.3: Add credentials to your Google cloud project and select "Service Account".

- Grant the service account access to Google Speech Client.
- Once you have your service account set up you can add a key and make a downloadable JSON file. Store it somewhere (private) on your computer. You don't need to go through these steps for every new project - once you have a key you can use it for all of your projects.

> **Warning**
>
> Be careful not to store the json file in the same location as any experiment folder that might later be shared on - this is a private file - so keep it somewhere safe.

- Finally, in go fo File > Preferences and add the path to the JSON file in General > appKeyGoogleCloud.

> **Warning**
>
> Remember to check that your accounts billing information stays up to date. Even if you haven't done enough recordings to warrant a large payment, if a card on your billing account expires this will invalidate the JSON key and raise a "billing" error in .



Service account details

2 Grant this service account access to the project (optional)

Grant this service account access to PsychoPy Speech Experiment so that it has permission to complete specific actions on the resources in your project. Learn more

Select a role

Condition

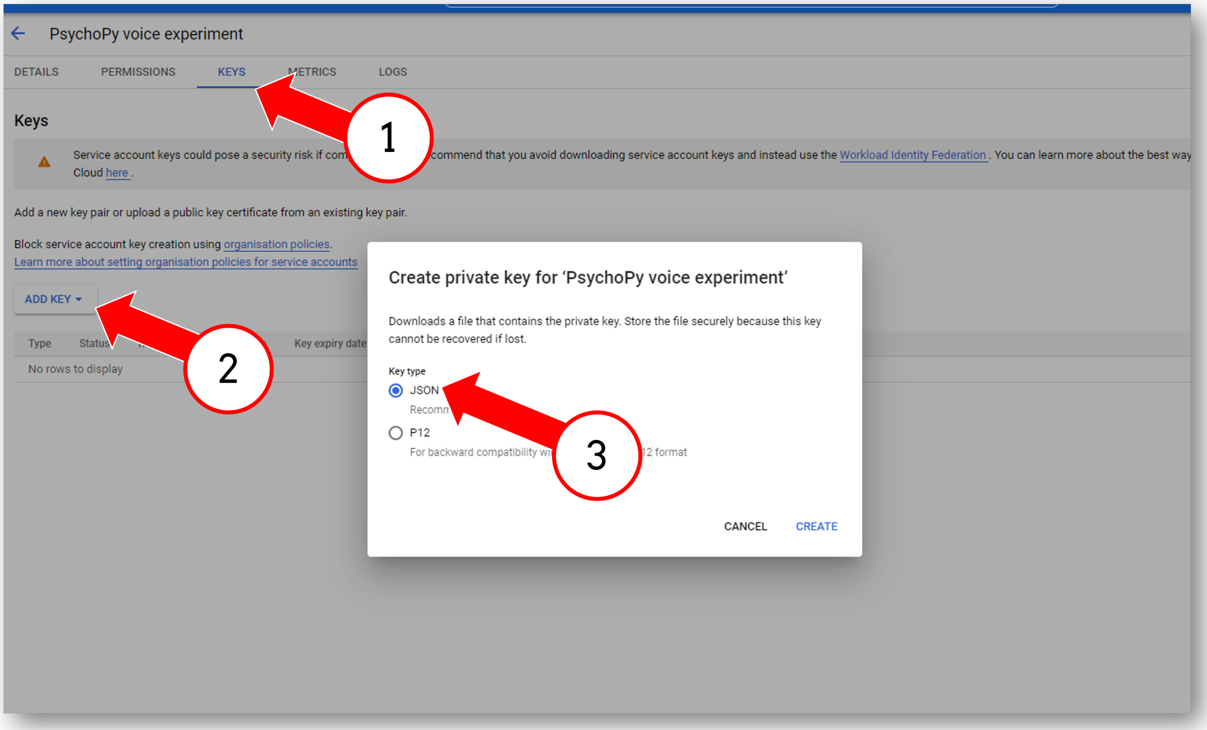
Filter spee

Cloud Speech Administrator
Grants full access to all resources in Speech-to-Text

Cloud Speech Client
Grants access to the recognition APIs.

Cloud Speech Editor
Grants access to edit resources in Speech-to-Text

Fig. 5.4: Search for "Google Speech Client" and give this account access to that API.



Keys

Service account keys could pose a security risk if com commend that you avoid downloading service account keys and instead use the Workload Identity Federation . You can learn more about the best way Cloud here .

Add a new key pair or upload a public key certificate from an existing key pair.

Block service account key creation using organisation policies.
Learn more about setting organisation policies for service accounts

Create private key for 'PsychoPy voice experiment'

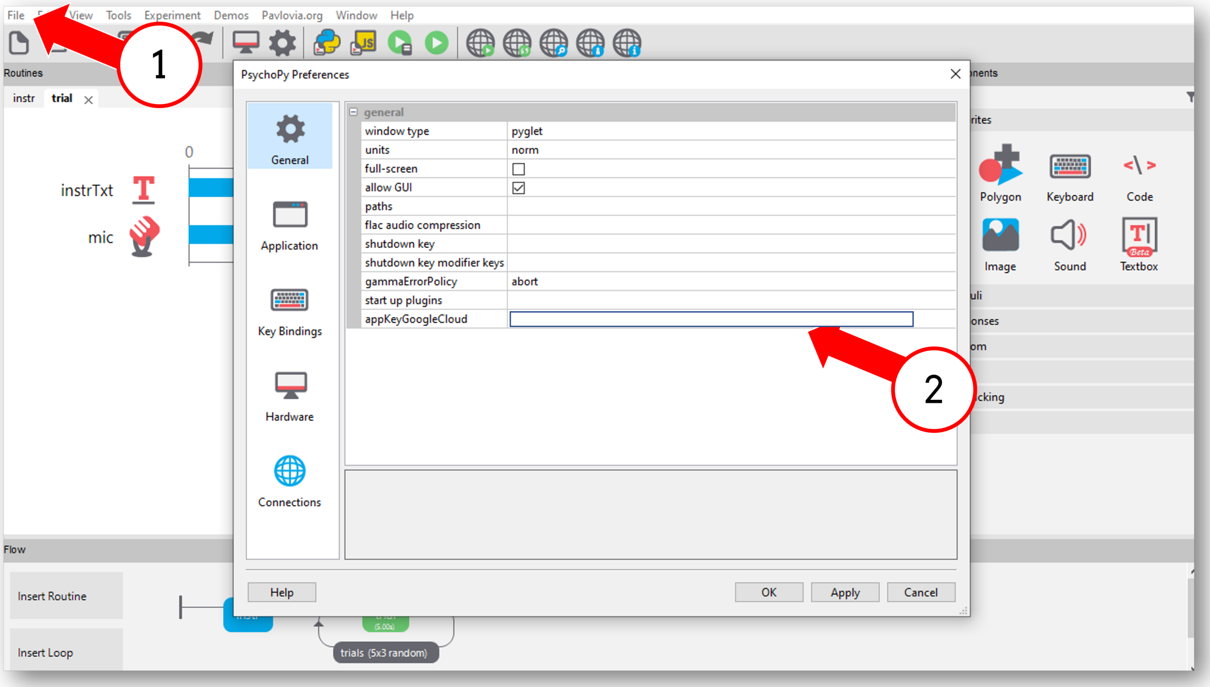
Downloads a file that contains the private key. Store the file securely because this key cannot be recovered if lost.

Key type

● JSON
Recomm

○ P12
For backward compatibility wi 12 format

Fig. 5.5: Generate a downloadable JSON for this project.



general	
window type	pyglet
units	norm
full-screen	□
allow GUI	☑
paths	
flac audio compression	
shutdown key	
shutdown key modifier keys	
gammaErrorPolicy	abort
start up plugins	
appKeyGoogleCloud	

Fig. 5.6: Setup your preferences to use your downloaded JSON - this will apply to all experiments using the mic - not just this experiment.

CHAPTER

SIX

CODER

The coder view is designed for those wishing to make scripts from scratch, either to make their experiments or do other things. Coder view does not teach you about Python *per se*, and you are recommended also to learn about that (Python has many excellent tutorials for programmers and non-programmers alike). In particular, dictionaries, lists and numpy arrays are used a great deal in most experiments.

You can program experiments in any python development environment (e.g. PyCharm, Spyder would be excellent examples of full-featured editors). So, *why use Coder view in PsychoPy?* The answer is that the PsychoPy as a standalone package also includes several common python libraries you would use when making experiments in python. In general there will therefore be fewer steps to take to configure your python environment in coder. So if you are teaching python, there should be less work to set up the environment for each student! *However* if you are teaching python for many purposes beyond making experiments, you might want to move to another IDE (Integrated Development Environment), because coder won't have *everything* you need imported.

You can learn to use the scripting interface to in several ways, and you should probably follow a combination of them:

- Check the content of our PsychoPy workshops (we currently focus on coding concepts on day 3).
- *Basic Concepts*: some of the logic of scripting
- *Tutorials*: walk you through the development of some semi-complete experiments
- demos: in the demos menu of Coder view.
- use the *Builder* to *compile a script* and see how it works (you can actually compile to Python or Javascript to learn a bit of both!). This is also useful for understanding the *Code Component*, you can write a snippet in a code component in builder and compile to see where it is written in the script (but remember exporting to coder is a one way street, you can't make edits in coder and hope that is reflected back in the builder experiment).

You should check the *Reference Manual (API)* for further details and, ultimately, go into PsychoPy and start examining the source code. It's just regular python!

6.1 Basic Concepts

6.1.1 Presenting Stimuli

Note

Before you start, tell about your monitor(s) using the *Monitor Center*. That way you get to use units (like degrees of visual angle) that will transfer easily to other computers.

Stimulus objects

Python is an 'object-oriented' programming language, meaning that most stimuli in are represented by python objects, with various associated methods and information.

Typically you should create your stimulus with the initial desired attributes once, at the beginning of the script, and then change select attributes later (see section below on setting stimulus attributes). For instance, create your text and then change its color any time you like:

```python
from psychopy import visual, core
win = visual.Window([400,400])
message = visual.TextStim(win, text='hello')
message.autoDraw = True  # Automatically draw every frame
win.flip()
core.wait(2.0)
message.text = 'world'  # Change properties of existing stim
win.flip()
core.wait(2.0)
```

Setting stimulus attributes

Stimulus attributes are typically set using either:

- a string, which is just some characters (as *message.text = 'world'* above)
- a scalar (a number; see below)
- an x,y-pair (two numbers; see below)

x,y-pair:

is very flexible in terms of input. You can specify the widely used x,y-pairs using these types:

- A Tuple (x, y) with two elements
- A List [x, y] with two elements
- A numpy array([x, y]) with two elements

However, always converts the x,y-pairs to numpy arrays internally. For example, all three assignments of pos are equivalent here:

```python
stim.pos = (0.5, -0.2)  # Right and a bit up from the center
print(stim.pos)  # array([0.5, -0.2])

stim.pos = [0.5, -0.2]
print(stim.pos)  # array([0.5, -0.2])

stim.pos = numpy.array([0.5, -0.2])
print(stim.pos)  # array([0.5, -0.2])
```

Choose your favorite :-) However, you can't assign elementwise:

```python
stim.pos[1] = 4  # has no effect
```

Scalar:

Int or Float.

Mostly, scalars are no-brainers to understand. E.g.:

```python
stim.ori = 90  # Rotate stimulus 90 degrees
stim.opacity = 0.8  # Make the stimulus slightly transparent.
```

However, scalars can also be used to assign x,y-pairs. In that case, both x and y get the value of the scalar. E.g.:

```
stim.size = 0.5
print(stim.size)  # array([0.5, 0.5])
```

Operations on attributes:

Operations during assignment of attributes are a handy way to smoothly alter the appearance of your stimuli in loops.

Most scalars and x,y-pairs support the basic operations:

```
stim.attribute += value  # addition
stim.attribute -= value  # subtraction
stim.attribute *= value  # multiplication
stim.attribute /= value  # division
stim.attribute %= value  # modulus
stim.attribute **= value # power
```

They are easy to use and understand on scalars:

```
stim.ori = 5     # 5.0, set rotation
stim.ori += 3.8  # 8.8, rotate clockwise
stim.ori -= 0.8  # 8.0, rotate counterclockwise
stim.ori /= 2    # 4.0, home in on zero
stim.ori **= 3   # 64.0, exponential increase in rotation
stim.ori %= 10   # 4.0, modulus 10
```

However, they can also be used on x,y-pairs in very flexible ways. Here you can use both scalars and x,y-pairs as operators. In the latter case, the operations are element-wise:

```
stim.size = 5            # array([5.0, 5.0]), set quadratic size
stim.size +=2            # array([7.0, 7.0]), increase size
stim.size /= 2           # array([3.5, 3.5]), downscale size
stim.size += (0.5, 2.5) # array([4.0, 6.0]), a little wider and much taller
stim.size *= (2, 0.25)  # array([8.0, 1.5]), upscale horizontal and downscale vertical
```

Operations are not meaningful for strings.

Timing

There are various ways to measure and control timing in :

- using frame refresh periods (most accurate, least obvious)
- checking the time on `Clock` objects
- using `core.wait()` commands (most obvious, least flexible/accurate)

Using core.wait(), as in the above example, is clear and intuitive in your script. But it can't be used while something is changing. For more flexible timing, you could use a `Clock()` object from the `core` module:

```
from psychopy import visual, core

# Setup stimulus
win = visual.Window([400, 400])
gabor = visual.GratingStim(win, tex='sin', mask='gauss', sf=5, name='gabor')
gabor.autoDraw = True  # Automatically draw every frame
```

(continues on next page)

(continued from previous page)

```python
gabor.autoLog = False  # Or we'll get many messages about phase change

# Let's draw a stimulus for 2s, drifting for middle 0.5s
clock = core.Clock()
while clock.getTime() < 2.0:  # Clock times are in seconds
    if 0.5 <= clock.getTime() < 1.0:
        gabor.phase += 0.1  # Increment by 10th of cycle
    win.flip()
```

Clocks are accurate to around 1ms (better on some platforms), but using them to time stimuli is not very accurate because it fails to account for the fact that one frame on your monitor has a fixed frame rate. In the above, the stimulus does not actually get drawn for exactly 0.5s (500ms). If the screen is refreshing at 60Hz (16.7ms per frame) and the *getTime()* call reports that the time has reached 1.999s, then the stimulus will draw again for a frame, in accordance with the *while* loop statement and will ultimately be displayed for 2.0167s. Alternatively, if the time has reached 2.001s, there will not be an extra frame drawn. So using this method you get timing accurate to the nearest frame period but with little consistent precision. An error of 16.7ms might be acceptable to long-duration stimuli, but not to a brief presentation. It also might also give the false impression that a stimulus can be presented for any given period. At 60Hz refresh you can not present your stimulus for, say, 120ms; the frame period would limit you to a period of 116.7ms (7 frames) or 133.3ms (8 frames).

As a result, the most precise way to control stimulus timing is to present them for a specified number of frames. The frame rate is extremely precise, much better than ms-precision. Calls to *Window.flip()* will be synchronised to the frame refresh; the script will not continue until the flip has occurred. As a result, on most cards, as long as frames are not being 'dropped' (see *Detecting dropped frames*) you can present stimuli for a fixed, reproducible period.

Note

Some graphics cards, such as Intel GMA graphics chips under win32, don't support frame sync. Avoid integrated graphics for experiment computers wherever possible.

Using the concept of fixed frame periods and *flip()* calls that sync to those periods we can time stimulus presentation extremely precisely with the following:

```python
from psychopy import visual, core

# Setup stimulus
win = visual.Window([400, 400])
gabor = visual.GratingStim(win, tex='sin', mask='gauss', sf=5,
    name='gabor', autoLog=False)
fixation = visual.GratingStim(win, tex=None, mask='gauss', sf=0, size=0.02,
    name='fixation', autoLog=False)

# Let's draw a stimulus for 200 frames, drifting for frames 50:100
for frameN in range(200):   # For exactly 200 frames
    if 10 <= frameN < 150:  # Present fixation for a subset of frames
        fixation.draw()
    if 50 <= frameN < 100:  # Present stim for a different subset
        gabor.phase += 0.1  # Increment by 10th of cycle
        gabor.draw()
    win.flip()
```

Using autoDraw

Stimuli are typically drawn manually on every frame in which they are needed, using the *draw()* function. You can also set any stimulus to start drawing every frame using *stim.autoDraw = True* or *stim.autoDraw = False*. If you use these commands on stimuli that also have *autoLog=True*, then these functions will also generate a log message on the frame when the first drawing occurs and on the first frame when it is confirmed to have ended.

6.1.2 Logging data

TrialHandler and *StairHandler* can both generate data outputs in which responses are stored, in relation to the stimulus conditions. In addition to those data outputs, can create detailed chronological log files of events during the experiment.

Log levels and targets

Log messages have various levels of severity:
ERROR, WARNING, DATA, EXP, INFO and DEBUG

Multiple *targets* can also be created to receive log messages. Each target has a particular critical level and receives all logged messages greater than that. For example, you could set the console (visual output) to receive only warnings and errors, have a central log file that you use to store warning messages across studies (with file mode *append*), and another to create a detailed log of data and events within a single study with *level=INFO*:

```
from psychopy import logging
logging.console.setLevel(logging.WARNING)
# overwrite (filemode='w') a detailed log of the last run in this dir
lastLog = logging.LogFile("lastRun.log", level=logging.INFO, filemode='w')
# also append warnings to a central log file
centralLog = logging.LogFile("C:\\psychopyExps.log", level=logging.WARNING, filemode='a')
```

Updating the logs

For performance purposes log files are not actually written when the log commands are 'sent'. They are stored in a list and processed automatically when the script ends. You might also choose to force a *flush* of the logged messages manually during the experiment (e.g. during an inter-trial interval):

```
from psychopy import logging

...

logging.flush()    # write messages out to all targets
```

This should only be necessary if you want to see the logged information as the experiment progresses.

AutoLogging

New in version 1.63.00

Certain events will log themselves automatically by default. For instance, visual stimuli send log messages every time one of their parameters is changed, and when autoDraw is toggled they send a message that the stimulus has started/stopped. All such log messages are timestamped with the frame flip on which they take effect. To avoid this logging, for stimuli such as fixation points that might not be critical to your analyses, or for stimuli that change constantly and will flood the logging system with messages, the autoLogging can be turned on/off at initialisation of the stimulus and can be altered afterwards with *.setAutoLog(True/False)*

Manual methods

In addition to a variety of automatic logging messages, you can create your own, of various levels. These can be timestamped immediately:

```
from psychopy import logging
logging.log(level=logging.WARN, msg='something important')
logging.log(level=logging.EXP, msg='something about the conditions')
logging.log(level=logging.DATA, msg='something about a response')
logging.log(level=logging.INFO, msg='something less important')
```

There are additional convenience functions for the above: logging.warn('a warning') etc.

For stimulus changes you probably want the log message to be timestamped based on the frame flip (when the stimulus is next presented) rather than the time that the log message is sent:

```
from psychopy import logging, visual
win = visual.Window([400,400])
win.flip()
logging.log(level=logging.EXP, msg='sent immediately')
win.logOnFlip(level=logging.EXP, msg='sent on actual flip')
win.flip()
```

Using a custom clock for logs

New in version 1.63.00

By default times for log files are reported as seconds after the very beginning of the script (often it takes a few seconds to initialise and import all modules too). You can set the logging system to use any given `core.Clock` object (actually, anything with a *getTime()* method):

```
from psychopy import core, logging
globalClock = core.Clock()
logging.setDefaultClock(globalClock)
```

6.1.3 Handling Trials and Conditions

TrialHandler

This is what underlies the random and sequential loop types in *Builder*, they work using the *method of constants*. The trialHandler presents a predetermined list of conditions in either a sequential or random (without replacement) order.

see *`TrialHandler`* for more details.

TrialHandlerExt (For oddball paradigms)

For handling trial sequences in a *non-counterbalanced design* (i.e. *oddball paradigms*, https://en.wikipedia.org/wiki/Oddball_paradigm). The oddball paradigm is very popular in EEG research.

Its functions are a superset of the class TrialHandler, and as such, can also be used for normal trial handling.

see *`TrialHandlerExt`* for more details.

StairHandler

This generates the next trial using an *adaptive staircase*. The conditions are not predetermined and are generated based on the participant's responses.

Staircases are predominately used in psychophysics to measure the discrimination and detection thresholds. However they can be used in any experiment which varies a numeric value as a result of a 2 alternative forced choice (2AFC) response.

The StairHandler systematically generates numbers based on staircase parameters. These can then be used to define a stimulus parameter e.g. spatial frequency, stimulus presentation duration. If the participant gives the incorrect response the number generated will get larger and if the participant gives the correct response the number will get smaller.

see `StairHandler` for more details

6.1.4 Global Event Keys

Global event keys are single keys (or combinations of a single key and one or more "modifier" keys such as Ctrl, Alt, etc.) with an associated Python callback function. This function will be executed if the key (or key/modifiers combination) was pressed.

Note

Global event keys only work with the *pyglet* backend, which is the default.

fully automatically monitors and processes key presses during most portions of the experimental run, for example during *core.wait()* periods, or when calling *win.flip()*. If a global event key press is detected, the specified function will be run immediately. You are not required to manually poll and check for key presses. This can be particularly useful to implement a global "shutdown" key, or to trigger laboratory equipment on a key press when testing your experimental script – without cluttering the code. But of course the application is not limited to these two scenarios. In fact, you can associate any Python function with a global event key.

All active global event keys are stored in *event.globalKeys*.

Adding a global event key (simple)

First, let's ensure no global event keys are currently set by calling func:*event.globalKeys.clear*.

```
>>> from psychopy import event
>>> event.globalKeys.clear()
```

To add a new global event key, you need to invoke func:*event.globalKeys.add*. This function has two required arguments: the key name, and the function to associate with that key.

```
>>> key = 'a'
>>> def myfunc():
...     pass
...
>>> event.globalKeys.add(key=key, func=myfunc)
```

Look at *event.globalKeys*, we can see that the global event key has indeed been created.

```
>>> event.globalKeys
<_GlobalEventKeys :
    [A] -> 'myfunc' <function myfunc at 0x10669ba28>
>
```

Your output should look similar. You may happen to spot We can take a closer look at the specific global key event we added.

```
>>> event.globalKeys['a']
_GlobalEvent(func=<function myfunc at 0x10669ba28>, func_args=(), func_kwargs={}, name=
↪'myfunc')
```

This output tells us that

- our key *a* is associated with our function *myfunc*
- *myfunc* will be called without passing any positional or keyword arguments (*func_args* and *func_kwargs*, respectively)
- the event name was automatically set to the name of the function.

Note

Pressing the key won't do anything unless a *psychopy.visual.Window* is created and and its :func:~`psychopy.visual.Window.flip` method or *psychopy.core.wait()* are called.

Adding a global event key (advanced)

We are going to associate a function with a more complex calling signature (with positional and keyword arguments) with a global event key. First, let's create the dummy function:

```
>>> def myfunc2(*args, **kwargs):
...     pass
...
```

Next, compile some positional and keyword arguments and a custom name for this event. Positional arguments must be passed as tists or uples, and keyword arguments as dictionaries.

```
>>> args = (1, 2)
>>> kwargs = dict(foo=3, bar=4)
>>> name = 'my name'
```

Note

Even when intending to pass only a single positional argument, *args* must be a list or tuple, e.g., *args = [1]* or *args = (1,)*.

Finally, specify the key and a combination of modifiers. While key names are just strings, modifiers are lists or tuples of modifier names.

```
>>> key = 'b'
>>> modifiers = ['ctrl', 'alt']
```

Note

Even when specifying only a single modifier key, *modifiers* must be a list or tuple, e.g., *modifiers = ['ctrl']* or *modifiers = ('ctrl',)*.

We are now ready to create the global event key.

```
>>> event.globalKeys.add(key=key, modifiers=modifiers,
... func=myfunc2, func_args=args, func_kwargs=kwargs,
... name=name)
```

Check that the global event key was successfully added.

```
>>> event.globalKeys
<_GlobalEventKeys :
    [A] -> 'myfunc' <function myfunc at 0x10669ba28>
    [CTRL] + [ALT] + [B] -> 'my name' <function myfunc2 at 0x112eecb90>
>
```

The key combination *[CTRL] + [ALT] + [B]* is now associated with the function *myfunc2*, which will be called in the following way:

```
myfunc2(1, 2, foo=2, bar=4)
```

Indexing

event.globalKeys can be accessed like an ordinary dictionary. The index keys are *(key, modifiers)* namedtuples.

```
>>> event.globalKeys.keys()
[_IndexKey(key='a', modifiers=()), _IndexKey(key='b', modifiers=('ctrl', 'alt'))]
```

To access the global event associated with the key combination *[CTRL] + [ALT] + [B]*, we can do

```
>>> event.globalKeys['b', ['ctrl', 'alt']]
_GlobalEvent(func=<function myfunc2 at 0x112eecb90>, func_args=(1, 2), func_kwargs={'foo
↪': 3, 'bar': 4}, name='my name')
```

To make access more convenient, specifying the modifiers is optional in case none were passed to `psychopy.event.globalKeys.add()` when the global event key was added, meaning the following commands are identical.

```
>>> event.globalKeys['a', ()]
_GlobalEvent(func=<function myfunc at 0x10669ba28>, func_args=(), func_kwargs={}, name=
↪'myfunc')
>>> event.globalKeys['a']
_GlobalEvent(func=<function myfunc at 0x10669ba28>, func_args=(), func_kwargs={}, name=
↪'myfunc')
```

All elements of a global event can be accessed directly.

```
>>> index = ('b', ['ctrl', 'alt'])
>>> event.globalKeys[index].func
<function myfunc2 at 0x112eecb90>
>>> event.globalKeys[index].func_args
(1, 2)
>>> event.globalKeys[index].func_kwargs
{'foo': 3, 'bar': 4}
>>> event.globalKeys[index].name
'my name'
```

Number of active event keys

The number of currently active event keys can be retrieved by passing *event.globalKeys* to the *len()* function.

```
>>> len(event.globalKeys)
2
```

Removing global event keys

There are three ways to remove global event keys:

- using `psychopy.event.globalKeys.remove()`,
- using *del*, and
- using `psychopy.event.globalKeys.pop()`.

`psychopy.event.globalKeys.remove()`

To remove a single key, pass the key name and modifiers (if any) to `psychopy.event.globalKeys.remove()`.

```
>>> event.globalKeys.remove(key='a')
```

A convenience method to quickly delete *all* global event keys is to pass *key='all'*

```
>>> event.globalKeys.remove(key='all')
```

del

Like with other dictionaries, items can be removed from *event.globalKeys* by using the *del* statement. The provided index key must be specified as described in *Indexing*.

```
>>> index = ('b', ['ctrl', 'alt'])
>>> del event.globalKeys[index]
```

`psychopy.event.globalKeys.pop()`

Again, as other dictionaries, *event.globalKeys* provides a *pop* method to retrieve an item and remove it from the dict. The first argument to *pop* is the index key, specified as described in *Indexing*. The second argument is optional. Its value will be returned in case no item with the matching indexing key could be found, for example if the item had already been removed previously.

```
>>> r = event.globalKeys.pop('a', None)
>>> print(r)
_GlobalEvent(func=<function myfunc at 0x10669ba28>, func_args=(), func_kwargs={}, name=
↪'myfunc')
>>> r = event.globalKeys.pop('a', None)
>>> print(r)
None
```

Global shutdown key

The preferences for *shutdownKey* and *shutdownKeyModifiers* (both unset by default) will be used to automatically create a global shutdown key. To demonstrate this automated behavior, let us first change the preferences programmatically (these changes will be lost when quitting the current Python session).

```
>>> from psychopy.preferences import prefs
>>> prefs.general['shutdownKey'] = 'q'
```

We can now check if a global shutdown key has been automatically created.

```
>>> from psychopy import event
>>> event.globalKeys
<_GlobalEventKeys :
    [Q] -> 'shutdown (auto-created from prefs)' <function quit at 0x10c171938>
>
```

And indeed, it worked!

What happened behind the scenes? When importing the *psychopy.event* module, the initialization of *event.globalKeys* checked for valid shutdown key preferences and automatically initialized a shutdown key accordingly. This key is associated with the :func:~`psychopy.core.quit` function, which will shut down .

```
>>> from psychopy.core import quit
>>> event.globalKeys['q'].func == quit
True
```

Of course you can very easily add a global shutdown key manually, too. You simply have to associate a key with :func:~`psychopy.core.quit`.

```
>>> from psychopy import core, event
>>> event.globalKeys.add(key='q', func=core.quit, name='shutdown')
```

That's it!

A working example

In the above code snippets, our global event keys were not actually functional, as we didn't create a window, which is required to actually collect the key presses. Our working example will thus first create a window and then add global event keys to change the window color and quit the experiment, respectively.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from __future__ import print_function
from psychopy import core, event, visual


def change_color(win, log=False):
    win.color = 'blue' if win.color == 'gray' else 'gray'
    if log:
        print('Changed color to %s' % win.color)


win = visual.Window(color='gray')
text = visual.TextStim(win,
                       text='Press C to change color,\n CTRL + Q to quit.')

# Global event key to change window background color.
event.globalKeys.add(key='c',
```

(continues on next page)

(continued from previous page)

```
                     func=change_color,
                     func_args=[win],
                     func_kwargs=dict(log=True),
                     name='change window color')

# Global event key (with modifier) to quit the experiment ("shutdown key").
event.globalKeys.add(key='q', modifiers=['ctrl'], func=core.quit)

while True:
    text.draw()
    win.flip()
```

6.2 Tutorials

6.2.1 Tutorial 1: Generating your first stimulus

A tutorial to get you going with your first stimulus display.

Know your monitor

has been designed to handle your screen calibrations for you. It is also designed to operate (if possible) in the final experimental units that you like to use e.g. degrees of visual angle.

In order to do this needs to know a little about your monitor. There is a GUI to help with this (select MonitorCenter from the tools menu of IDE or run …site-packages/monitors/MonitorCenter.py).

In the MonitorCenter window you can create a new monitor name, insert values that describe your monitor and run calibrations like gamma corrections. For now you can just stick to the [*testMonitor*] but give it correct values for your screen size in number of pixels and width in cm.

Now, when you create a window on your monitor you can give it the name 'testMonitor' and stimuli will know how they should be scaled appropriately.

Your first stimulus

Building stimuli is extremely easy. All you need to do is create a `Window`, then some stimuli. Draw those stimuli, then update the window. has various other useful commands to help with timing too. Here's an example. Type it into a coder window, save it somewhere and press run.

```
from psychopy import visual, core  # import some libraries from PsychoPy
from psychopy.hardware import keyboard

#create a window
mywin = visual.Window([800,600], monitor="testMonitor", units="deg")

#create some stimuli
grating = visual.GratingStim(win=mywin, mask="circle", size=3, pos=[-4,0], sf=3)
fixation = visual.GratingStim(win=mywin, size=0.5, pos=[0,0], sf=0, rgb=-1)

#create a keyboard component
kb = keyboard.Keyboard()

#draw the stimuli and update the window
```

(continues on next page)

(continued from previous page)

```
grating.draw()
fixation.draw()
mywin.update()

#pause, so you get a chance to see it!
core.wait(5.0)
```

Note

For those new to Python. Did you notice that the grating and the fixation stimuli both call *GratingStim* but have different arguments? One of the nice features about python is that you can select which arguments to set. GratingStim has over 15 arguments that can be set, but the others just take on default values if they aren't needed.

That's a bit easy though. Let's make the stimulus move, at least! To do that we need to create a loop where we change the phase (or orientation, or position…) of the stimulus and then redraw. Add this code in place of the drawing code above:

```
for frameN in range(200):
    grating.setPhase(0.05, '+')  # advance phase by 0.05 of a cycle
    grating.draw()
    fixation.draw()
    mywin.update()
```

That ran for 200 frames (and then waited 5 seconds as well). Maybe it would be nicer to keep updating until the user hits a key instead. That's easy to add too. In the first line add `event` to the list of modules you'll import. Then replace the line:

```
for frameN in range(200):
```

with the line:

```
while True: #this creates a never-ending loop
```

Then, within the loop (make sure it has the same indentation as the other lines) add the lines:

```
    if len(kb.getKeys()) > 0:
        break
    event.clearEvents()
```

the first line counts how many keys have been pressed since the last frame. If more than zero are found then we break out of the never-ending loop. The second line clears the event buffer and should always be called after you've collected the events you want (otherwise it gets full of events that we don't care about like the mouse moving around etc…).

Your finished script should look something like this:

```
from psychopy import visual, core, event #import some libraries from PsychoPy
from psychopy.hardware import keyboard

#create a window
mywin = visual.Window([800,600],monitor="testMonitor", units="deg")

#create some stimuli
```

(continues on next page)

(continued from previous page)

```
grating = visual.GratingStim(win=mywin, mask='circle', size=3, pos=[-4,0], sf=3)
fixation = visual.GratingStim(win=mywin, size=0.2, pos=[0,0], sf=0, rgb=-1)

#create a keyboard component
kb = keyboard.Keyboard()

#draw the stimuli and update the window
while True: #this creates a never-ending loop
    grating.setPhase(0.05, '+')#advance phase by 0.05 of a cycle
    grating.draw()
    fixation.draw()
    mywin.flip()

    if len(kb.getKeys()) > 0:
        break
    event.clearEvents()

#cleanup
mywin.close()
core.quit()
```

There are several more simple scripts like this in the demos menu of the Coder and Builder views and many more to download. If you're feeling like something bigger then go to *Tutorial 2: Measuring a JND using a staircase procedure* which will show you how to build an actual experiment.

6.2.2 Tutorial 2: Measuring a JND using a staircase procedure

This tutorial builds an experiment to test your just-noticeable-difference (JND) to orientation, that is it determines the smallest angular deviation that is needed for you to detect that a gabor stimulus isn't vertical (or at some other reference orientation). The method presents a pair of stimuli at once with the observer having to report with a key press whether the left or the right stimulus was at the reference orientation (e.g. vertical).

You can download the full code here. Note that the entire experiment is constructed of less than 100 lines of code, including the initial presentation of a dialogue for parameters, generation and presentation of stimuli, running the trials, saving data and outputting a simple summary analysis for feedback. Not bad, eh?

There are a great many modifications that can be made to this code, however this example is designed to demonstrate how much can be achieved with very simple code. Modifying existing is an excellent way to begin writing your own scripts, for example you may want to try changing the appearance of the text or the stimuli.

Get info from the user

The first lines of code import the necessary libraries. We need lots of the modules for a full experiment, as well as *numpy* (which handles various numerical/mathematical functions):

```
"""measure your JND in orientation using a staircase method"""
from psychopy import core, visual, gui, data, event
from psychopy.tools.filetools import fromFile, toFile
import numpy, random
```

Also note that there are two ways to insert comments in Python (and you should do this often!). Using triple quotes, as in *""”Here's my comment”""*, allows you to write a comment that can span several lines. Often you need that at the start of your script to leave yourself a note about the implementation and history of what you've written. For single-line comments, as you'll see below, you can use a simple # to indicate that the rest of the line is a comment.

The try:...except:... lines allow us to try and load a parameter file from a previous run of the experiment. If that fails (e.g. because the experiment has never been run) then create a default set of parameters. These are easy to store in a python dictionary that we'll call *expInfo*:

```
try:  # try to get a previous parameters file
    expInfo = fromFile('lastParams.pickle')
except:  # if not there then use a default set
    expInfo = {'observer':'jwp', 'refOrientation':0}
expInfo['dateStr'] = data.getDateStr()  # add the current time
```

The last line adds the current date to to the information, whether we loaded from a previous run or created default values.

So having loaded those parameters, let's allow the user to change them in a dialogue box (which we'll call dlg). This is the simplest form of dialogue, created directly from the dictionary above. the dialogue will be presented immediately to the user and the script will wait until they hit *OK* or *Cancel*.

If they hit *OK* then dlg.OK=True, in which case we'll use the updated values and save them straight to a parameters file (the one we try to load above).

If they hit *Cancel* then we'll simply quit the script and not save the values.

```
# present a dialogue to change params
dlg = gui.DlgFromDict(expInfo, title='simple JND Exp', fixed=['dateStr'])
if dlg.OK:
    toFile('lastParams.pickle', expInfo)  # save params to file for next time
else:
    core.quit()  # the user hit cancel so exit
```

Setup the information for trials

We'll create a file to which we can output some data as text during each trial (as well as *outputting a binary file* at the end of the experiment). actually has supporting functions to do this automatically, but here we're showing you the manual way to do it.

We'll create a filename from the subject+date+".csv" (note how easy it is to concatenate strings in python just by 'adding' them). *csv* files can be opened in most spreadsheet packages. Having opened a text file for writing, the last line shows how easy it is to send text to this target document.

```
# make a text file to save data
fileName = expInfo['observer'] + expInfo['dateStr']
dataFile = open(fileName+'.csv', 'w')  # a simple text file with 'comma-separated-values'
dataFile.write('targetSide,oriIncrement,correct\n')
```

allows us to set up an object to handle the presentation of stimuli in a staircase procedure, the *StairHandler*. This will define the increment of the orientation (i.e. how far it is from the reference orientation). The staircase can be configured in many ways, but we'll set it up to begin with an increment of 20deg (very detectable) and home in on the 80% threshold value. We'll step up our increment every time the subject gets a wrong answer and step down if they get three right answers in a row. The step size will also decrease after every 2 reversals, starting with an 8dB step (large) and going down to 1dB steps (smallish). We'll finish after 50 trials.

```
# create the staircase handler
staircase = data.StairHandler(startVal = 20.0,
                          stepType = 'db', stepSizes=[8,4,4,2],
                          nUp=1, nDown=3,  # will home in on the 80% threshold
                          nTrials=1)
```

Build your stimuli

Now we need to create a window, some stimuli and timers. We need a *~psychopy.visual.Window* in which to draw our stimuli, a fixation point and two *~psychopy.visual.GratingStim* stimuli (one for the target probe and one as the foil). We can have as many timers as we like and reset them at any time during the experiment, but I generally use one to measure the time since the experiment started and another that I reset at the beginning of each trial.

```
# create window and stimuli
win = visual.Window([800,600],allowGUI=True,
                    monitor='testMonitor', units='deg')
foil = visual.GratingStim(win, sf=1, size=4, mask='gauss',
                          ori=expInfo['refOrientation'])
target = visual.GratingStim(win, sf=1, size=4, mask='gauss',
                            ori=expInfo['refOrientation'])
fixation = visual.GratingStim(win, color=-1, colorSpace='rgb',
                              tex=None, mask='circle', size=0.2)
# and some handy clocks to keep track of time
globalClock = core.Clock()
trialClock = core.Clock()
```

Once the stimuli are created we should give the subject a message asking if they're ready. The next two lines create a pair of messages, then draw them into the screen and then update the screen to show what we've drawn. Finally we issue the command event.waitKeys() which will wait for a keypress before continuing.

```
# display instructions and wait
message1 = visual.TextStim(win, pos=[0,+3],text='Hit a key when ready.')
message2 = visual.TextStim(win, pos=[0,-3],
    text="Then press left or right to identify the %.1f deg probe." %expInfo['refOrientation'])
message1.draw()
message2.draw()
fixation.draw()
win.flip()#to show our newly drawn 'stimuli'
#pause until there's a keypress
event.waitKeys()
```

Control the presentation of the stimuli

OK, so we have everything that we need to run the experiment. The following uses a for-loop that will iterate over trials in the experiment. With each pass through the loop the `staircase` object will provide the new value for the intensity (which we will call `thisIncrement`). We will randomly choose a side to present the target stimulus using `numpy.random.random()`, setting the position of the target to be there and the foil to be on the other side of the fixation point.

```
for thisIncrement in staircase:  # will continue the staircase until it terminates!
    # set location of stimuli
    targetSide= random.choice([-1,1])  # will be either +1(right) or -1(left)
    foil.setPos([-5*targetSide, 0])
    target.setPos([5*targetSide, 0])  # in other location
```

Then set the orientation of the foil to be the reference orientation plus `thisIncrement`, draw all the stimuli (including the fixation point) and update the window.

```
    # set orientation of probe
    foil.setOri(expInfo['refOrientation'] + thisIncrement)

    # draw all stimuli
    foil.draw()
    target.draw()
    fixation.draw()
    win.flip()
```

Wait for presentation time of 500ms and then blank the screen (by updating the screen after drawing just the fixation point).

```
    # wait 500ms; but use a loop of x frames for more accurate timing
    core.wait(0.5)
```

(This is not the most precise way to time your stimuli - you'll probably overshoot by one frame - but its easy to understand. allows you to present a stimulus for acertian number of screen refreshes instead which is better for short stimuli.)

Get input from the subject

Still within the for-loop (note the level of indentation is the same) we need to get the response from the subject. The method works by starting off assuming that there hasn't yet been a response and then waiting for a key press. For each key pressed we check if the answer was correct or incorrect and assign the response appropriately, which ends the trial. We always have to clear the event buffer if we're checking for key presses like this

```
    # get response
    thisResp=None
    while thisResp==None:
        allKeys=event.waitKeys()
        for thisKey in allKeys:
            if thisKey=='left':
                if targetSide==-1: thisResp = 1  # correct
                else: thisResp = -1              # incorrect
            elif thisKey=='right':
                if targetSide== 1: thisResp = 1  # correct
                else: thisResp = -1              # incorrect
            elif thisKey in ['q', 'escape']:
                core.quit()  # abort experiment
        event.clearEvents()  # clear other (eg mouse) events - they clog the buffer
```

Now we must tell the staircase the result of this trial with its `addData()` method. Then it can work out whether the next trial is an increment or decrement. Also, on each trial (so still within the for-loop) we may as well save the data as a line of text in that .csv file we created earlier.

```
    # add the data to the staircase so it can calculate the next level
    staircase.addData(thisResp)
    dataFile.write('%i,%.3f,%i\n' %(targetSide, thisIncrement, thisResp))
    core.wait(1)
```

Output your data and clean up

OK! We're basically done! We've reached the end of the for-loop (which occurred because the staircase terminated) which means the trials are over. The next step is to close the text data file and also save the staircase as a binary file (by 'pickling' the file in Python speak) which maintains a lot more info than we were saving in the text file.

```python
# staircase has ended
dataFile.close()
staircase.saveAsPickle(fileName)  # special python binary file to save all the info
```

While we're here, it's quite nice to give some immediate feedback to the user. Let's tell them the intensity values at the all the reversals and give them the mean of the last 6. This is an easy way to get an estimate of the threshold, but we might be able to do a better job by trying to reconstruct the psychometric function. To give that a try see the staircase analysis script of *Tutorial 3*.

Having saved the data you can give your participant some feedback and quit!

```python
# give some output to user in the command line in the output window
print('reversals:')
print(staircase.reversalIntensities)
approxThreshold = numpy.average(staircase.reversalIntensities[-6:])
print('mean of final 6 reversals = %.3f' % (approxThreshold))

# give some on-screen feedback
feedback1 = visual.TextStim(
        win, pos=[0,+3],
        text='mean of final 6 reversals = %.3f' % (approxThreshold))

feedback1.draw()
fixation.draw()
win.flip()
event.waitKeys()  # wait for participant to respond

win.close()
core.quit()
```

6.2.3 Tutorial 3: Analysing data in Python

You could simply output your data as tab- or comma-separated text files and analyse the data in some spreadsheet package. But the matplotlib library in Python also allows for very neat and simple creation of publication-quality plots.

This script shows you how to use a couple of functions from to open some data files (`psychopy.gui.fileOpenDlg()`) and create a psychometric function out of some staircase data (`psychopy.data.functionFromStaircase()`).

Matplotlib is then used to plot the data.

Note

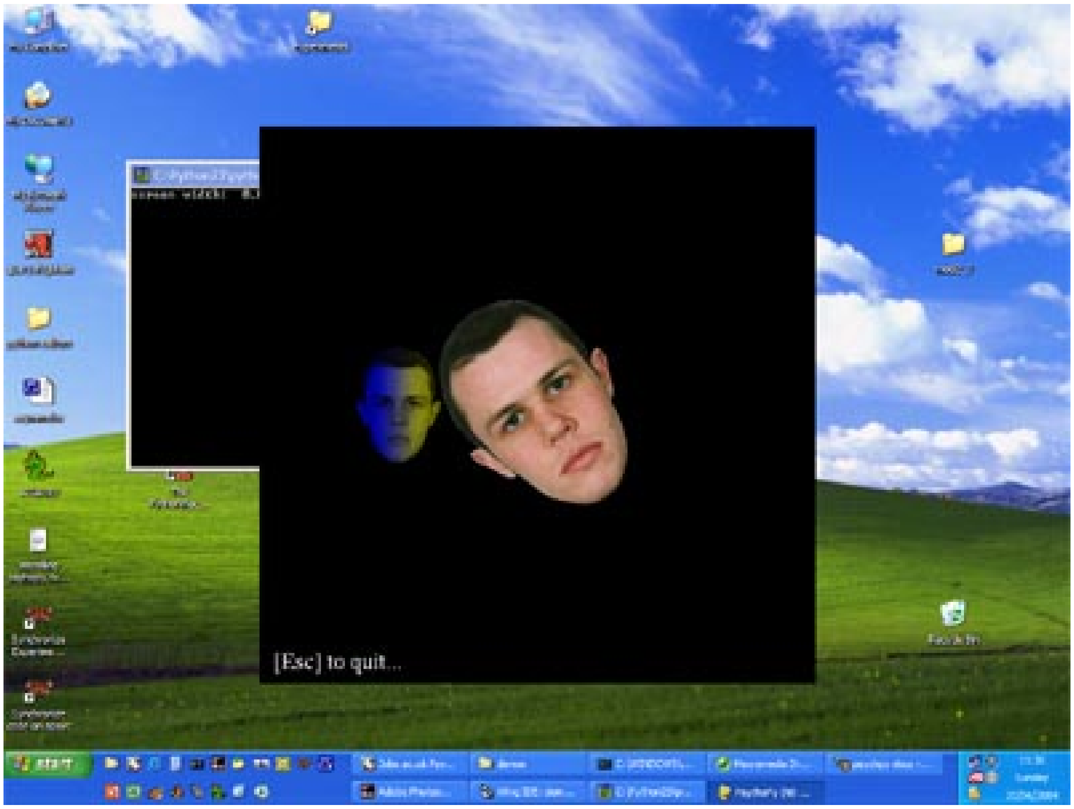
Matplotlib and `pylab`. Matplotlib is a python library that has similar command syntax to most of the plotting functions in Matlab(tm). In can be imported in different ways; the `import pylab` line at the beginning of the script is the way to import matploblib as well as a variety of other scientific tools (that aren't strictly to do with plotting *per se*).

```python
#This analysis script takes one or more staircase datafiles as input
#from a GUI. It then plots the staircases on top of each other on
#the left and a combined psychometric function from the same data
#on the right

from psychopy import data, gui, core
from psychopy.tools.filetools import fromFile
import pylab

#Open a dialog box to select files from
files = gui.fileOpenDlg('.')
if not files:
    core.quit()

#get the data from all the files
allIntensities, allResponses = [],[]
for thisFileName in files:
    thisDat = fromFile(thisFileName)
    allIntensities.append( thisDat.intensities )
    allResponses.append( thisDat.data )

#plot each staircase
pylab.subplot(121)
colors = 'brgkcmbrgkcm'
lines, names = [],[]
for fileN, thisStair in enumerate(allIntensities):
    #lines.extend(pylab.plot(thisStair))
    #names = files[fileN]
    pylab.plot(thisStair, label=files[fileN])
#pylab.legend()

#get combined data
combinedInten, combinedResp, combinedN = \
             data.functionFromStaircase(allIntensities, allResponses, 5)
#fit curve - in this case using a Weibull function
fit = data.FitWeibull(combinedInten, combinedResp, guess=[0.2, 0.5])
smoothInt = pylab.arange(min(combinedInten), max(combinedInten), 0.001)
smoothResp = fit.eval(smoothInt)
thresh = fit.inverse(0.8)
print(thresh)

#plot curve
pylab.subplot(122)
pylab.plot(smoothInt, smoothResp, '-')
pylab.plot([thresh, thresh],[0,0.8],'--'); pylab.plot([0, thresh],\
[0.8,0.8],'--')
pylab.title('threshold = %0.3f' %(thresh))
#plot points
pylab.plot(combinedInten, combinedResp, 'o')
pylab.ylim([0,1])

pylab.show()
```

6.2.4 Draw a rotating face

This demo shows a couple of uses of bitmap images. You can use them either as textures for images or as the mask for another texture (the face on the left is used as a mask for a blue and yellow grating).

To work the demo needs a square image called face.jpg (e.g. `/images/face.jpg`)



[Esc] to quit...

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from psychopy import core, visual, event

#create a window to draw in
myWin = visual.Window((600, 600), allowGUI=False, color=(-1, -1, -1))

#INITIALISE SOME STIMULI
faceRGB = visual.GratingStim(myWin, tex='face.jpg',
                             mask=None,
                             pos=(0.0, 0.0),
                             size=(1.0, 1.0),
                             sf=(1.0, 1.0))

faceALPHA = visual.GratingStim(myWin, pos=(-0.5, 0),
                               tex="sin", mask="face.jpg",
                               color=[1.0, 1.0, -1.0],
                               size=(0.5, 0.5), sf=1.0,
                               units="norm")

message = visual.TextStim(myWin, pos=(-0.95, -0.95),
```

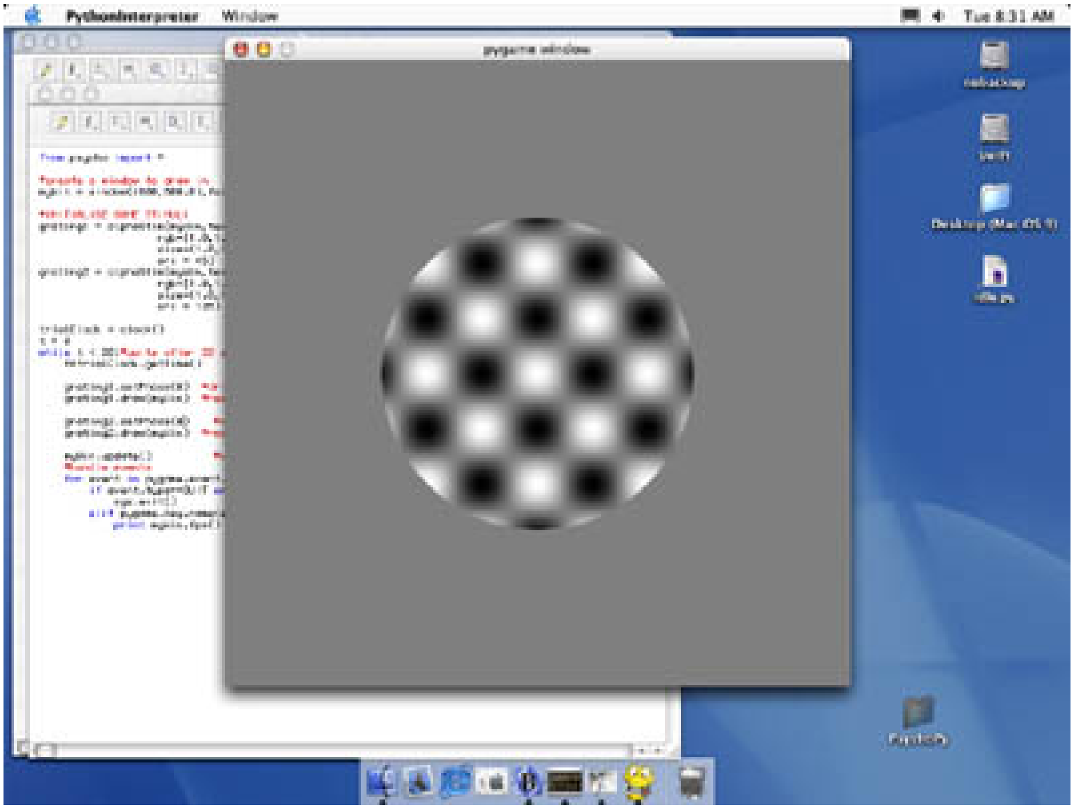
(continues on next page)

(continued from previous page)

```
                          text='[Esc] to quit', color=1,
                          alignText='left', anchorHoriz='left',
                          alignTextVert='bottom', anchorVert='bottom')

trialClock = core.Clock()
t = lastFPSupdate = 0
while True:
    t = trialClock.getTime()
    faceRGB.setOri(1, '+')#advance ori by 1 degree
    faceRGB.draw()

    faceALPHA.setPhase(0.01, "+")#advance phase by 1/100th of a cycle
    faceALPHA.draw()

    #update fps every second
    if t-lastFPSupdate > 1.0:
        lastFPS = myWin.fps()
        lastFPSupdate = t
        message.setText("%ifps, [Esc] to quit" %lastFPS)
    message.draw()

    myWin.flip()

    #handle key presses each frame
    for keys in event.getKeys():
        if keys in ['escape', 'q']:
            print(myWin.fps())
            myWin.close()
            core.quit()
    event.clearEvents()
```

6.2.5 Drifting plaid demo

The following code also checks for a keypress. The timing method simply uses a Clock object with the .getTime() call. This is very easy to code, but not quite as accurate as using the frame tick.



```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from psychopy import visual, core, event

#create a window to draw in
myWin = visual.Window((600,600), allowGUI=False)

#INITIALISE SOME STIMULI
grating1 = visual.GratingStim(myWin, mask="gauss",
                              color=[1.0, 1.0, 1.0],
                              opacity=1.0,
                              size=(1.0, 1.0),
                              sf=(4,0), ori=45)

grating2 = visual.GratingStim(myWin, mask="gauss",
                              color=[1.0, 1.0, 1.0],
                              opacity=0.5,
                              size=(1.0, 1.0),
                              sf=(4,0), ori=135)

trialClock = core.Clock()
t = 0
while t < 20:    # quits after 20 secs

    t = trialClock.getTime()
```

(continues on next page)

(continued from previous page)

```
    grating1.setPhase(1*t)  # drift at 1Hz
    grating1.draw()  #redraw it

    grating2.setPhase(2*t)    #drift at 2Hz
    grating2.draw()  #redraw it

    myWin.flip()          #update the screen

    #handle key presses each frame
    for keys in event.getKeys():
        if keys in ['escape','q']:
            core.quit()
```

CHAPTER

SEVEN

RUNNING AND SHARING STUDIES ONLINE

Online studies are realized via PsychoJS; the online counterpart of . To run your study online, these are the basic steps:

7.1 Status of online options

The table below shows you the current state of play of PsychoJS. Per feature we list whether it's:

1. Built-in Supported via the PsychoPy Builder
2. Prototype Supported via tutorials or customized experiments that can be cloned and adapted. Kudos to our users for pushing the envelope!
3. Not supported Supported by PsychoPy, but not yet supported by PsychoJS

Feature	Status	Notes
Stimuli		
Dots (RDK)	Prototype	The dots component isn't yet in PsychoJS. You could use pre-created movies instead- or try a workaround with code components here thanks to Francesco Cabiddu
Images	Built-in	Ensure to use the image extension when referencing images in your experiment e.g. ".png" ".jpg" - this will help avoid "Unknown Resource" errors
Movies	Built-in	Do check mediaFormats
Polygons	Built-in	If using circles online use a "regular" polygon with 100 vertices - rather than using the dropdown "circle" option
Text	Built-in	We recommend using "Text" rather than "TextBox" for static text online - since TextBox is still in beta
Textbox	Built-in	For versions preceding 2022.1 textbox needed a code component with *textbox.refresh()* in the "Begin Routine" to be used on several trials
Grating	Built-in	
	Not supported	Apertures, Envelope Gratings, Noise, Panoramic
Responses		
Form	Built-in	
Pavlovia Surveys	Built-in	
Gyroscope	Prototype	Measures the orientation of tablets and smartphones. Try it out
Eye-tracking	Prototype	Try it out
Keyboard	Built-in	
Mouse	Built-in	Mouse components translate to touch responses on touch screens
Slider	Built-in	Use slider and not "rating" for online studies
TextBox	Built-in	see above
Brush	Built-in	

continues on next page

Table 7.1 – continued from previous page

Feature	Status	Notes
Microphone	Built-in	available in 2021.2 onward
	Not supported	Joystick, Button boxes (Cedrus & IO Labs), Button component
Data		
CSV files	Built-in	These can easily be imported into analysis software, such as Matlab, R, JAMOVI, or JASP
Log files	Built-in	Low-level logs. These offer detailed information, but are hard to analyze
MongoDB	Built-in	Similar to CSV, but stored in a database instead of files
	Not supported	XLSX
Flow and Logic		
Code	Built-in	Insert snippets of programming code, which can be automatically translated from Python to JavaScript
Loops	Built-in	Loops allow randomization and importing condition files.
Staircases	Prototype	Adapt aspects of a trial based on earlier responses of a participant. You can use Multistair but specify only a single staircase - see below. Or try out a "Just Noticeable Difference" staircase via staircase-demo
Multistair	Built-in	Interleave several basic staircases.
QUEST staircases	Built-in	This is currently supported via jsQUEST you can try a demo and access the gitlab project to build on for your own research
External Tools		
AMT	Built-in	Amazon Mechanical Turk. See instructions in this forum post
Prolific	Built-in	See instructions at *Recruiting with Prolific*
Qualtrics	Built-in	There are many guides available for integrating Qualtrics on our forum
Sona	Built-in	See instructions at the Sona Systems website

Thanks go out to Anastasia Carter, Arnon Weinberg, Francesco Cabiddu, Lindsay Santacroce, and Wakefield Carter; they made tutorials and/or demo experiments available that we referenced in the list above.

Anything else we should add to the list above? Built a cool prototype? Please tell us via the PsychoPy Forum.

7.2 Creating online experiments from Builder

7.2.1 Export the HTML files

To generate an online experiment from Builder you can either:

- go to *>File>Export HTML*, or
- press 'sync' from the globe icons (see Fig. **??**, button 2).

Both of these will generate all the necessary files (HTML and JS) that you need for your study, however sync will also create a project on . NB - By default, the sync button exports an online experiment, but this can be changed via the experiment settings.

7.2.2 Synchronizing for the first time

The first time you sync your experiment, a dialog box will appear. The dialog box informs you that your .psyexp file does not belong to an existing project. Click "Create a project" if you wish to create a project, or click "Cancel" if you wish to return to your experiment in Builder. See Fig. **??**.

If you clicked the "Create a project" button, another window will appear. This window is designed to collect important metadata about your project, see Fig. **??**.

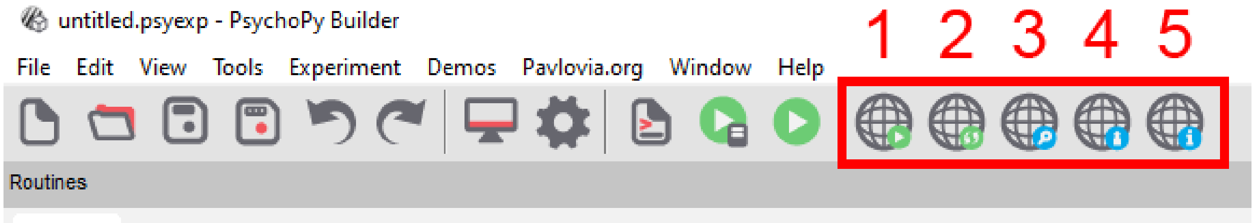
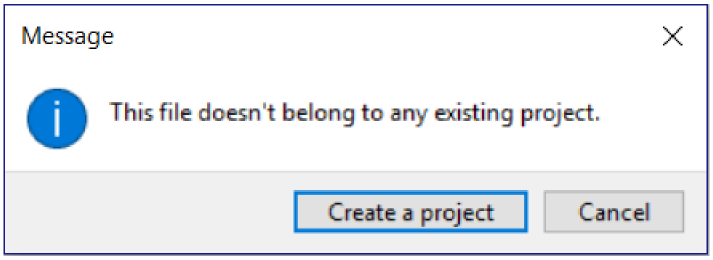
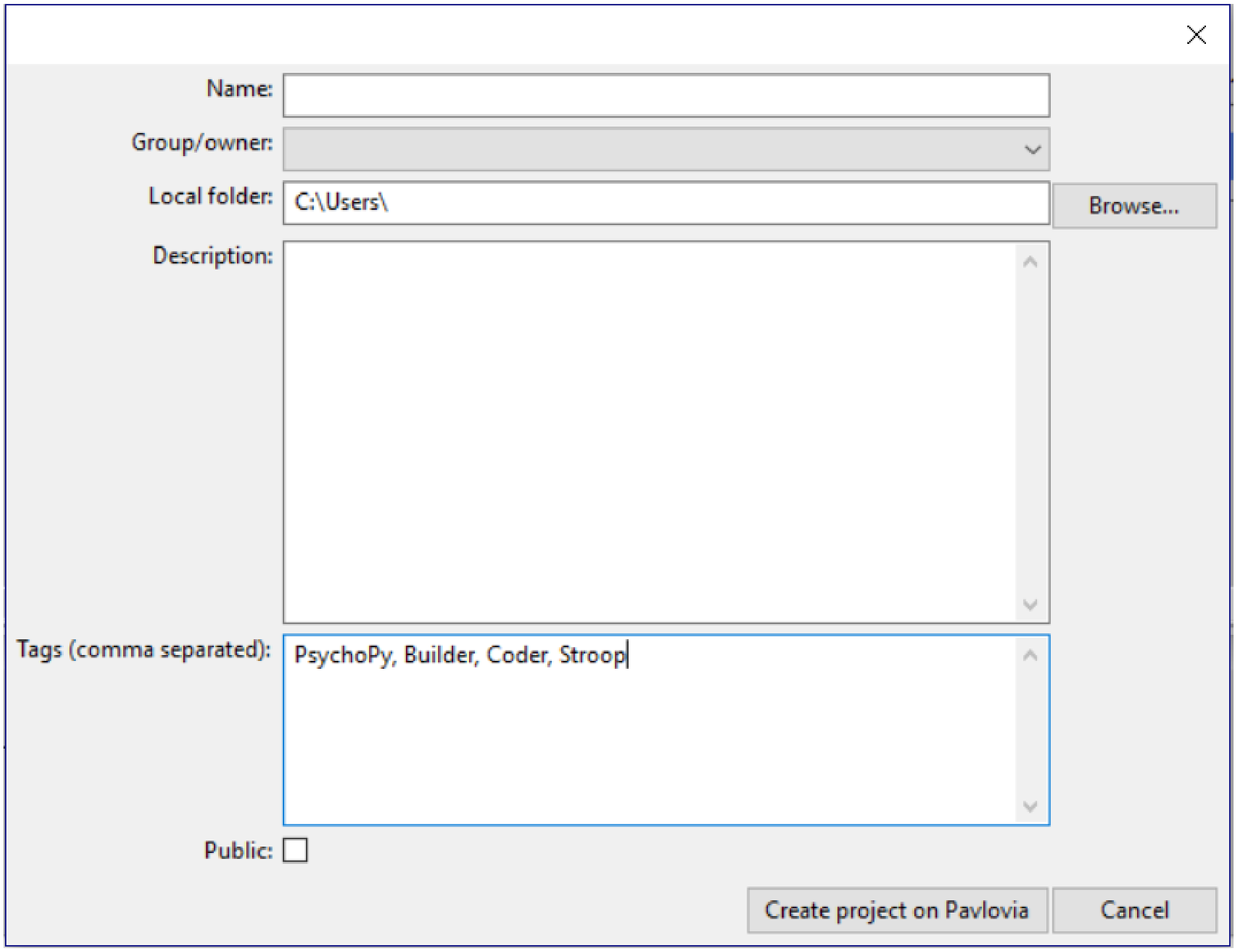


Fig. 7.1: Buttons for running an online study from the Builder.



This file doesn't belong to any existing project.

Fig. 7.2: The dialog that appears when an online project does not exist.



Name:

Group/owner:

Local folder: C:\Users\

Description:

Tags (comma separated): PsychoPy, Builder, Coder, Stroop

Public: □

Fig. 7.3: Dialog for creating your project on Pavlovia.org

Use this window to add information to store your project on Pavlovia:

- **Name.** This is the name of your project on Pavlovia
- **Group/Owner.** The user or group that may manage the project
- **Local folder.** The (local) project path on your computer. Use the Browse button to find your local directory, if required. Every file in this directory (and subdirectory) will be uploaded to pavlovia, so be sure you've only got files in there that are required by your experiment.
- **Description.** A brief description of your experiment.
- **Tags (comma separated).** These tags will be used to filter and search for experiments by keywords.
- **Public.** Tick this box if you would like to make your project public, which means that anyone can see and clone it.

When you have completed all fields in the Project window, click "Create project on Pavlovia" button to push your experiment to an online repository. Click "Cancel" if you wish to return to your experiment in Builder.

7.2.3 Viewing your experiment files

Once your experiment is online you will see your experiment in your Pavlovia Dashboard in the Experiments tab. After clicking your experiment you can set its status to "Pilotting" or "Running". Read more about the Experiment page here.

7.2.4 Running your experiment on Pavlovia.org from Builder

If you wish to run your experiment online, in a web-browser, you have two options. You can run your experiment directly from pavlovia.org, as described above, or you can run your experiment directly from Builder. There is also the option to send your experiment URL; more on that in *How to recruit participants and connect with online services*.

To run your experiment on via Builder, you must first ensure you have a valid internet connection, are logged in, and have created a repository for your project on . Once you have completed these steps, simply click button 1 in Fig. **??**.

7.2.5 Fetching your data

By default. the data are saved in a data folder next to the html file. You should see CSV files there that are similar to output files. There won't be any psydat files though. You could just download the data folder, or synchronize your experiment using the Builder and your data will be fetched to your local computer.

Alternatively, in the experiment dashboard, you can specify storing the data into a database. After specifying so, any data collected in the future can be downloaded as a ZIP file.

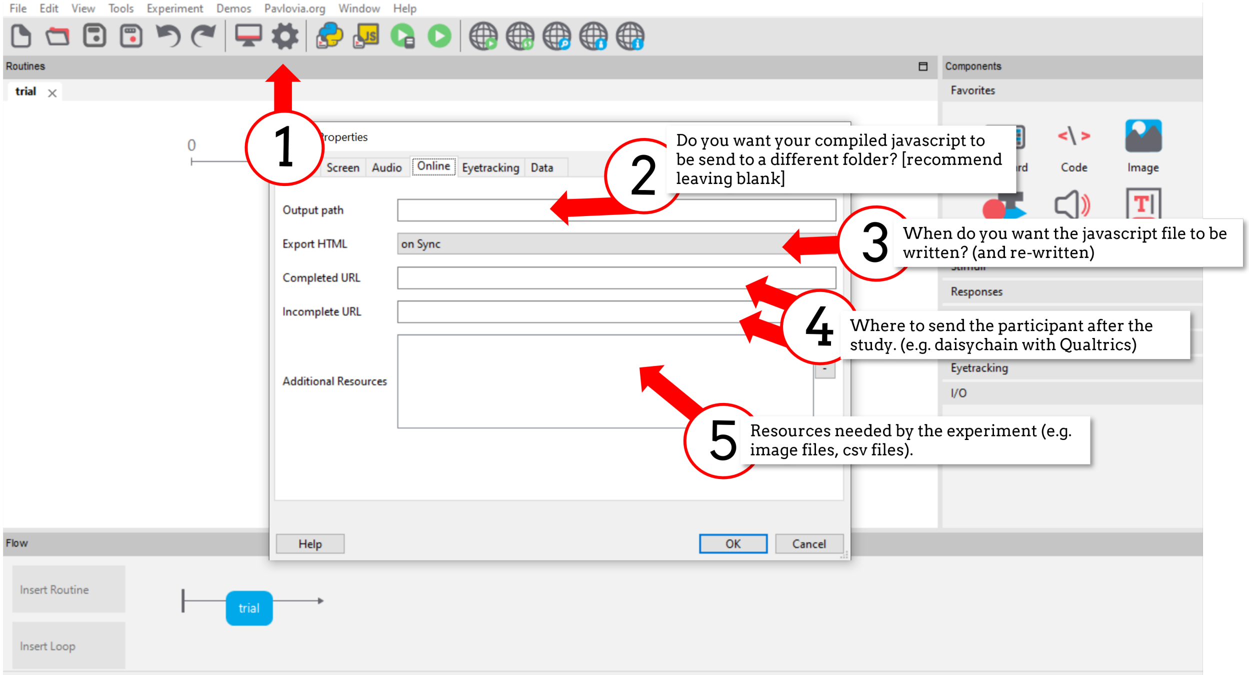
7.3 Configure the online settings of your experiment

For the most part, making a study to go online is identical to making a study to run locally. However, if you are making a study to run online, you *cannot code your experiment in pure python*. This is because your experiment needs to be created in a programming language that can be interpreted by your browser, and most browsers don't understand python, they understand JavaScript. If you make your experiment in Builder view, this GUI will write both a python and JavaScript version of your experiment. So it can be run online - without needing to learn JavaScript! So, to get started, we highly recommend you familiarize yourself with the Builder components of Builder.

So, you've made your study in Builder. To run the study online you want to start by configuring your online settings, these can be accessed through Experiment Settings:

This Online tab has the following parameters:

- **Output path**: Where the JavaScript version of your experiment (and accompanying html) will be written. *Recommended: leave blank*



Output path

Export HTML: on Sync

Completed URL

Incomplete URL

Additional Resources

1

2 Do you want your compiled javascript to be send to a different folder? [recommend leaving blank]

3 When do you want the javascript file to be written? (and re-written)

4 Where to send the participant after the study. (e.g. daisychain with Qualtrics)

5 Resources needed by the experiment (e.g. image files, csv files).

Fig. 7.4: The "Online" tab of Experiment settings.

- **Export HTML** - when will a new JavaScript file and html be exported. *Recommended - On Sync* * *On Sync* - every time you sync to pavlovia. * *On Save* - every time you save the psyexp file. * *Manually* - when you select the "compile to JS" icon.
- **Completed URL** - where will the participant be redirected once they complete the experiment? This is useful for if you are daisy chaining your experiment with recruitment websites
- **Incomplete URL** - where will the participant be redirected if they use the "esc" key to quit before the study completes.

Note

For completed and incomplete URLs, participants will only be redirected once their data has saved and they are presented with a green "Thank-You" message where they click "OK" - you might wish to emphasize this in your end of task instructions.

- **Additional Resources** - a list of resources used by the experiment, this can include conditions spreadsheets, images, sound files, movie files. Any resource added here will be loaded at the beginning of the study when it is loaded in the browser. It is highly recommended to add any resources that your study might need to the "Additional Resources" tab in your online settings. This can avoid "Unknown Resource" errors when running studies online.

Warning

Be mindful of how many resources your experiment has. If you have very large resource files (e.g. long movies) or a large number of files (e.g. >500 images) this can result in it taking a very long time for resources to load at the beginning of your experiment. You might want to consider looking into the *Resource Manager Component* component or the *Static Component*.

7.4 Launch your study on Pavlovia.org

Pavlovia.org is a site created by the PsychoPy team to make it easy to:

- run studies online
- host your experiments securely and easily without knowing about server technologies
- share studies with other scientists with collaborators of publicly (and find public studies shared by others)
- version control your work (using Git)

Site licenses and credits purchased from also provide a source of revenue to sustain what is a free software for the majority of our users. Read more about our sustainable open-source model here.

Most of the main tasks you will perform with can be carried out either in the PsychoPy application or on the website. Synchronizing your files can also be done with any Git client if you prefer.

To create and log in to your account on , you will need an active Internet connection. If you have not created your account, you can either

- go to and create your account, or
- click the login button highlighted in Fig. **??**, and create an account through the dialog box.

Once you have an account on and are logged in, your user name should appear in the corresponding menu (tpronk in Fig. **??**).

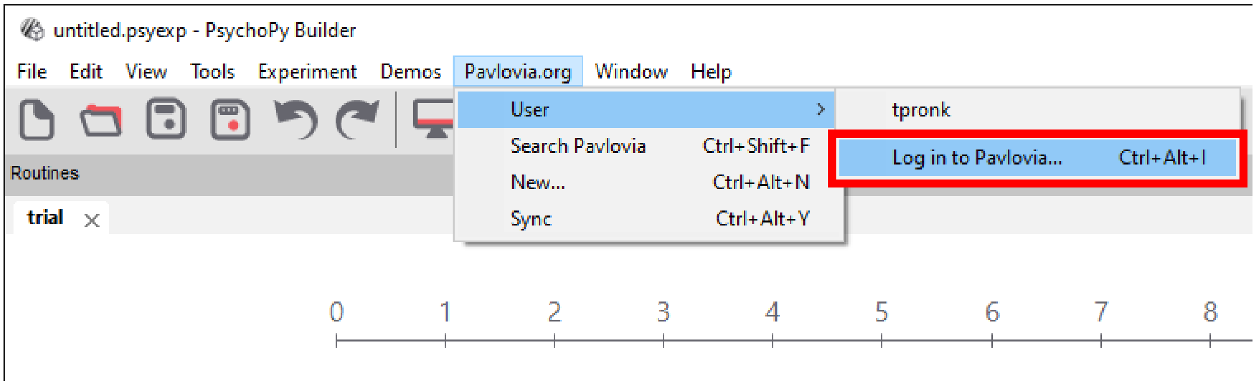


Fig. 7.5: Logging in on Pavlovia via the PsychoPy 3 Builder

7.4.1 Interacting with from the Builder App

When running your study online, the globe icons will allow you to directly interact with from the Builder GUI.

Synchronizing for the first time (creating a project)

The first time you sync your experiment, a dialog box will appear. The dialog box informs you that your .psyexp file does not belong to an existing project. Click “Create a project” if you wish to create a project, or click “Cancel” if you wish to return to your experiment in Builder. See Fig. **??**.

Editing project details from PsychoPy

PsychoPy versions 2020 and 2021

In versions preceding the 2022 release project details were created on the first synchronization. After clicking the “Create a project” button, another window would appear. This window was designed to collect important metadata about your project, see Fig. **??**.

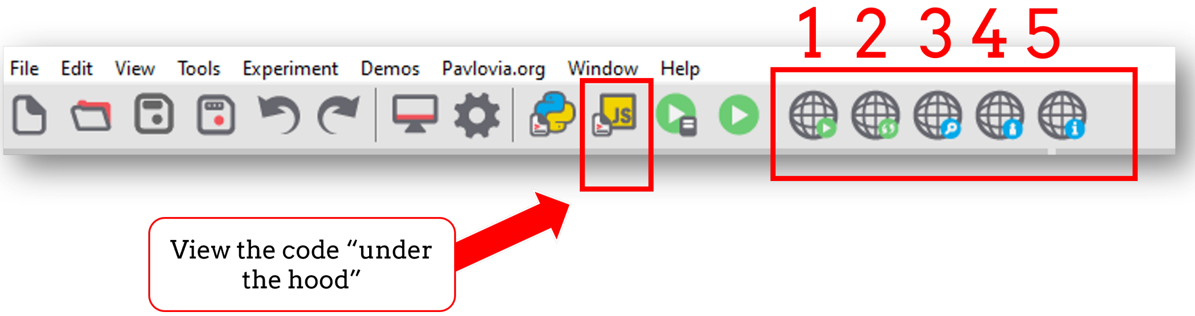
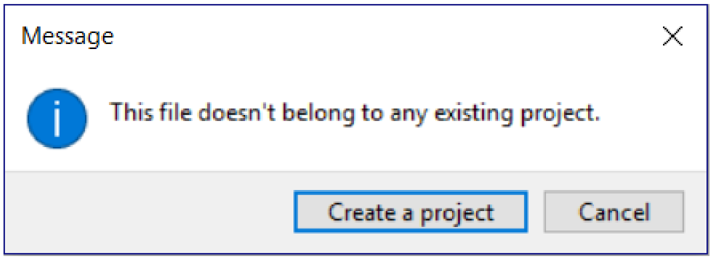


Fig. 7.6: Buttons for running an online study from the PsychoPy Builder. From left to right 1) Run study in browser directly (not project status must be "running", not "piloting" or "inactive"). 2) Sync project/create project on 3) Browse existing projects shared by others on 3) Check what account you are signed in on 4) Project information (note this will only be populated if the currently opened psyexp file has been synced to pavlovia.org)


Message

This file doesn't belong to any existing project.

Fig. 7.7: The dialog that appears when an online project does not exist.

From this window you could add information to store your project on Pavlovia:

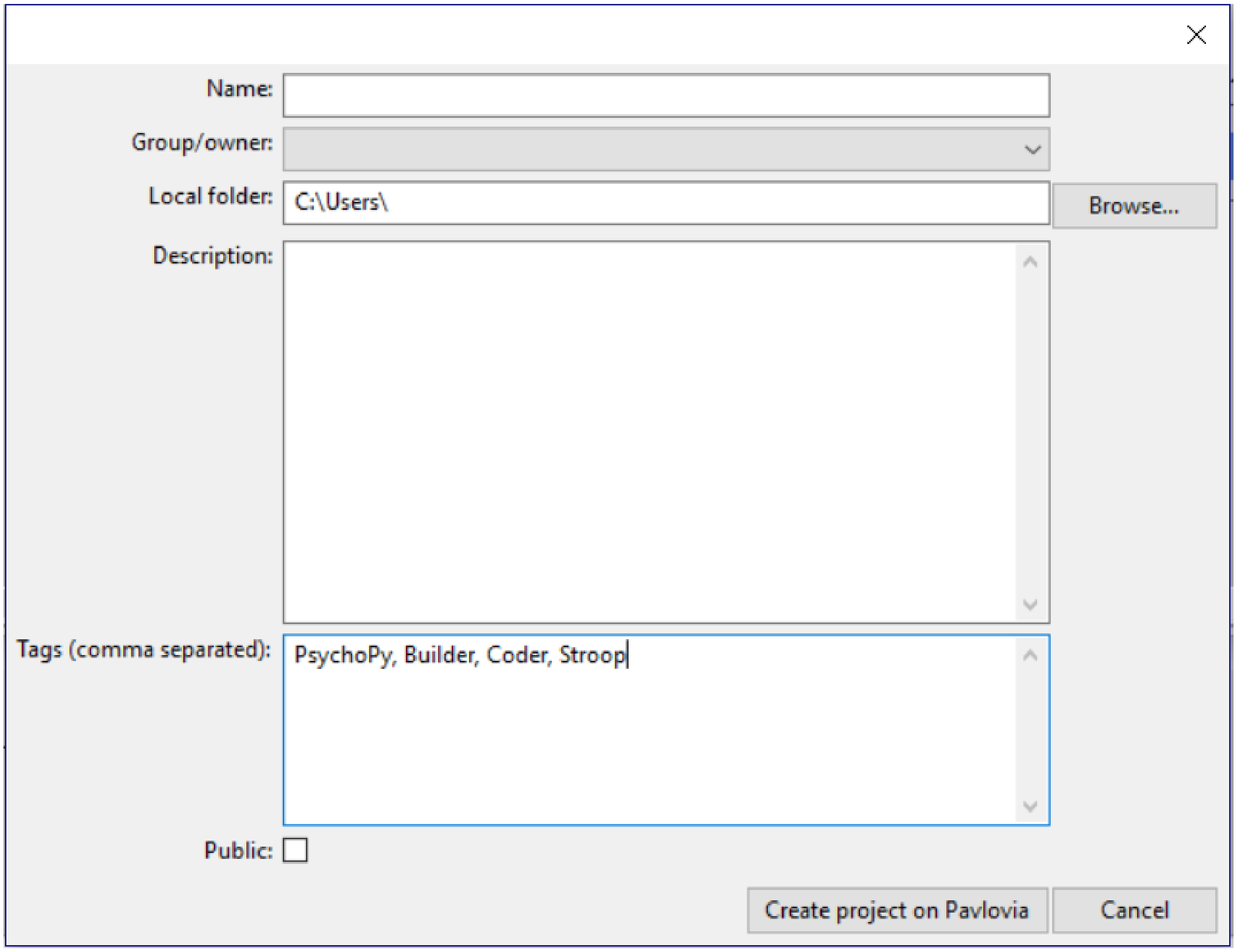
- **Name.** The name of your project on Pavlovia
- **Group/Owner.** The user or group that may manage the project
- **Local folder.** The (local) project path on your computer. Use the Browse button to find your local directory, if required. Every file in this directory (and subdirectory) will be uploaded to pavlovia, so be sure you've only got files in there that are required by your experiment.
- **Description.** A brief description of your experiment.
- **Tags (comma separated).** These tags will be used to filter and search for experiments by keywords.
- **Public.** Tick this box if you would like to make your project public, which means that anyone can see and clone it.

When you have completed all fields in the Project window, click "Create project on Pavlovia" button to push your experiment to an online repository. Click "Cancel" if you wish to return to your experiment in Builder.

PsychoPy version 2022

Version 2022 reduced the number of boxes that are presented on the initial sync of your project. In this version, when you select "Create project on Pavlovia" you will be presented with a dialog box that takes only two parameters:

You might think "why did we reduce the number of options to control the settings of the project". The answer to this is, we didn't! you just have to look somewhere else. To change your project settings directly from PsychoPy Builder, You can select the project information icon. Here, you can edit all the features of your project at any time (not just on the first synchronization!).



Name:

Group/owner:

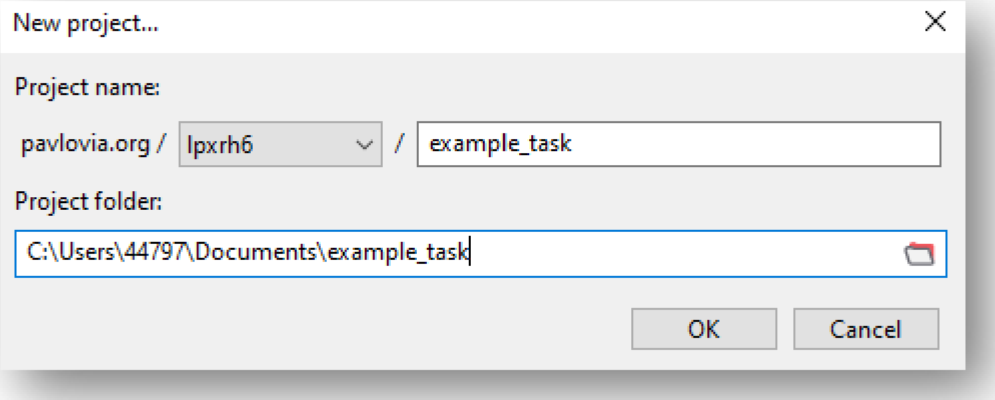
Local folder: C:\Users\

Description:

Tags (comma separated): PsychoPy, Builder, Coder, Stroop

Public: □

Fig. 7.8: Dialog for creating your project on Pavlovia.org



New project...

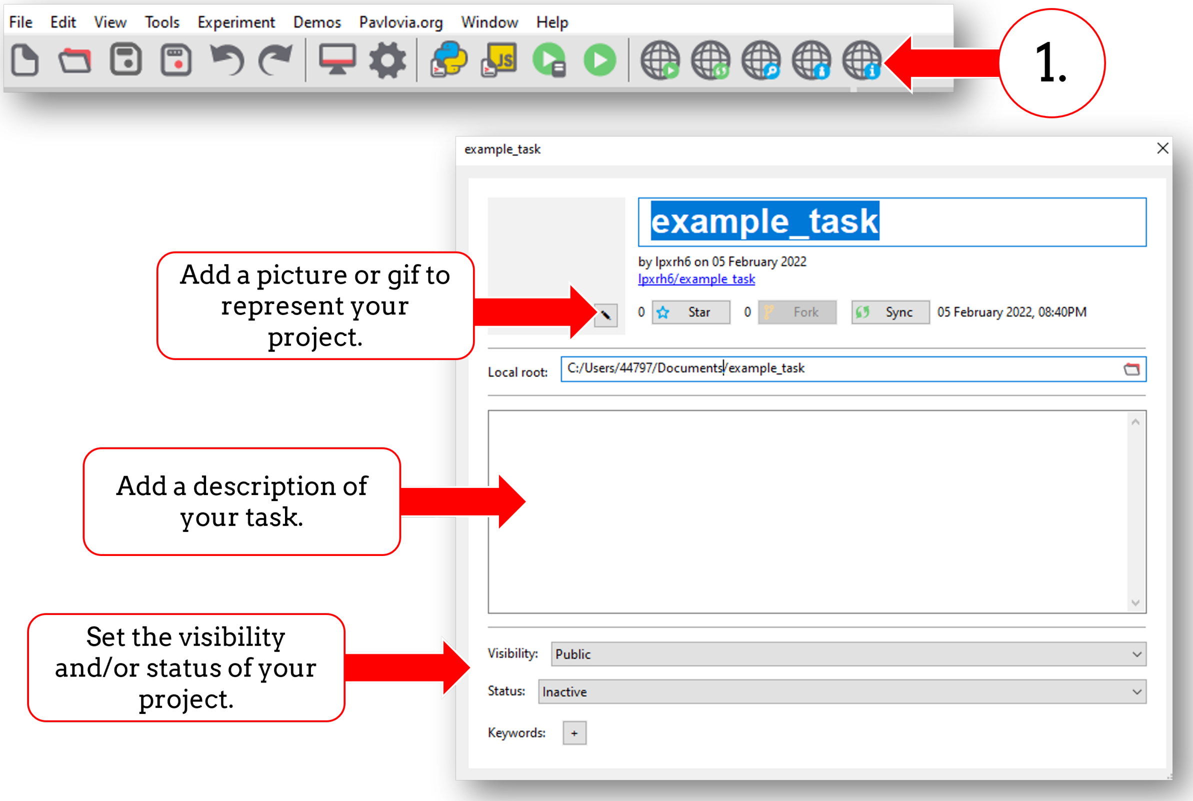
Project name:

pavlovia.org / lpxrh6 / example_task

Project folder:

C:\Users\44797\Documents\example_task

Fig. 7.9: The project creation dialog box in version 2022. The top row illustrates what the URL of your task will look like to your participant; all URLs will take the form of pavlovia.org/USERNAME/EXPERIMENT NAME. You can also set where the local root for your project is.



example_task

example_task

by lpxrh6 on 05 February 2022

lpxrh6/example_task

0 Star 0 Fork Sync 05 February 2022, 08:40PM

Local root: C:/Users/44797/Documents/example_task

Visibility: Public

Status: Inactive

Keywords: +

Add a picture or gif to represent your project.

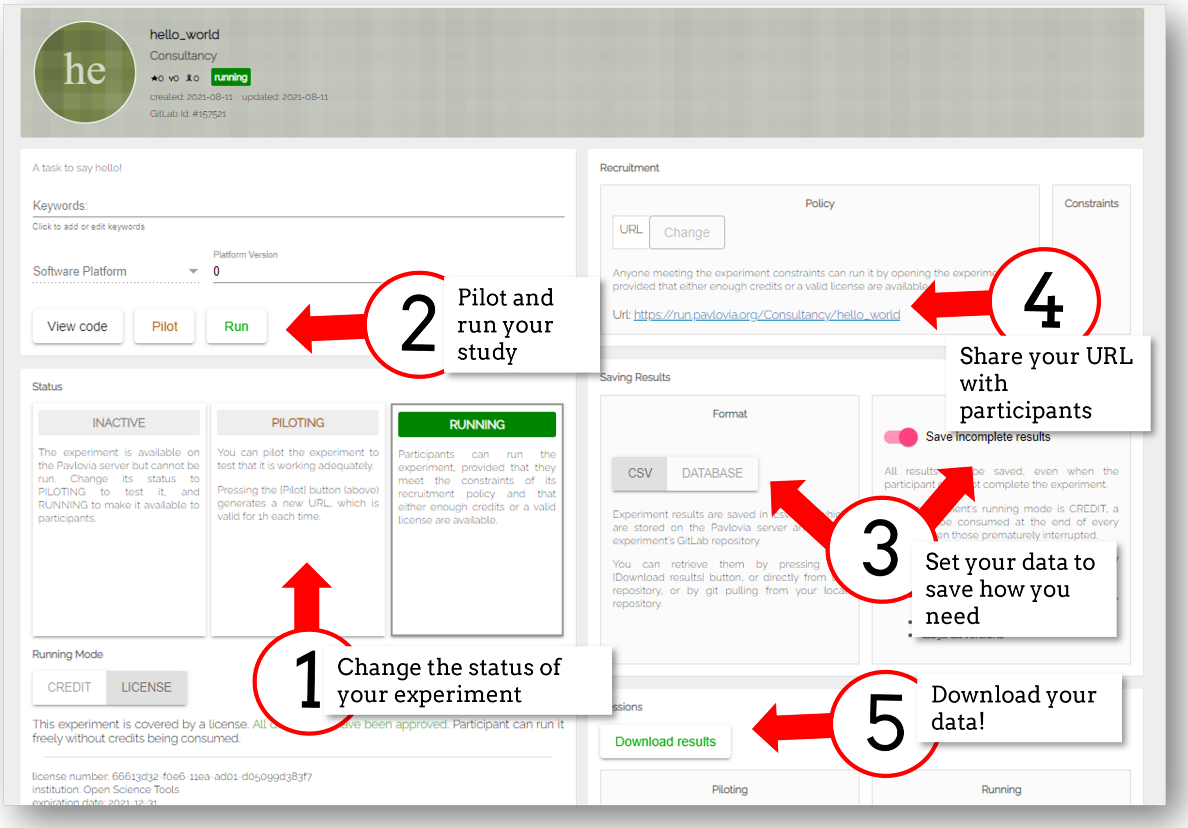
Add a description of your task.

Set the visibility and/or status of your project.

Fig. 7.10: The project information dialogue box.

7.4.2 Viewing your project on pavlovia.org

Once your experiment is online you will see your experiment in your Pavlovia Dashboard in the Experiments tab. After clicking your experiment you can set its status to “Pilotting” or “Running”. Read more about the Experiment page here.



The experiment is available on the Pavlovia server but cannot be run. Change its status to PILOTING to test it, and RUNNING to make it available to participants.

You can pilot the experiment to test that it is working adequately.

Pressing the [Pilot] button (above) generates a new URL, which is valid for 1h each time.

Participants can run the experiment, provided that they meet the constraints of its recruitment policy and that either enough credits or a valid license are available.

This experiment is covered by a license. All c… …ave been approved. Participant can run it freely without credits being consumed.

license number: 66613d32-f0e6-11ea-ad01-d05099d383f7
institution: Open Science Tools
expiration date: 2021-12-31

Anyone meeting the experiment constraints can run it by opening the experim… provided that either enough credits or a valid license are available.

Url: https://run.pavlovia.org/Consultancy/hello_world

Experiment results are saved in csv… …hic… are stored on the Pavlovia server a… experiment's GitLab repository.

You can retrieve them by pressing [Download results] button, or directly from … repository, or by git pulling from your local repository.

All results … be saved, even when the participant … t complete the experiment.

…ent's running mode is CREDIT, a … be consumed at the end of every … en those prematurely interrupted.

Fig. 7.11: Your experiment dashboard from pavlovia.org in browser.

7.4.3 Running your experiment on Pavlovia.org from Builder

If you wish to run your experiment online, in a web-browser, you have two options. You can run your experiment directly from pavlovia.org, as described above, or you can run your experiment directly from Builder. There is also the option to send your experiment URL; more on that in *How to recruit participants and connect with online services*.

To run your experiment on via Builder, you must first ensure you have a valid internet connection, are logged in, and have created a repository for your project on . Once you have completed these steps, simply click button 1 in Fig. **??**.

7.4.4 Fetching your data

Once you have run your study online, there are three ways you can fetch your data:

1. You can use the “download results” button on your experiment dashboard in browser.
2. You can sync your project from psychopy builder. Syncing is bi-directional, it *pushes* things that don’t exist online but do locally to your online project, and *pulls* things that exist online in your repository but not locally (i.e. data) to your local repository.
3. You can select “view code” from your page and look at the data file in the repository. This is handy if you only want to download a single file or two.

Note

Options 2 and 3 here will only work if you have data saving set to "csv" rather than "database". If you are using a component that stores data files other than a csv (for example a microphone component that saves audio recordings) you should also use option 1 only.

Database or csv?

A common question researchers have is "should I use database or csv as my preferred data saving mode?" To answer this consider the following:

1. Do you want to make your project public eventually? and, if so, do you also want to share the data files? if the answer to this second question is no - use database, as no csv files will be stored to the underlying gitlab repository - which is made public for public project.
2. Do you want a single file for each participant, or all participants data concatenated into one file? for separate files use csv, for one file use database.

When making an experiment to run online, there are a few important considerations to make and we **highly** recommend reading through the considerations below, as they could save a lot of time in the long run!

7.5 Resources in online studies

The loading of resources (images, sounds, movies etc.) in online experiments is necessarily different to the loading of resources in local studies (whatever the software package).

In a locally-executed experiment all the files that might be required by the study are already available on the computer. With an online experiment the HTML/JavaScript code needs to know what files in your project should be fetched from the server and sent to the participant ready for them to run the study.

PsychoPy/JS will try by default to *Auto-detect files and download at the start*, but sometimes you might still bump into an "Unknown Resource" error, and it's good to know why these occur and how to approach them. PsychoPy/PsychoJS now provides multiple options for how to control what resources are loaded in your experiment and when:

1. *Specifying "Additional Resources"* (fine for not-too-many resources)
2. *Specify the files to download at the start* (e.g. fetch during instructions slides)
3. *Specify the files to download each trial* (e.g. fetch a single image during fixation)

Also, take note of the advice below on choosing and converting to *appropriate media formats for your resources*

Warning

Critically, you should note that you can combine Methods 2 and 3, explicitly specifying resources that need to be fetched both at the start and on each trial, but you CANNOT combine the automatic method (1) with the explicit methods (2 and 3).

7.5.1 Auto-detect files and download at the start

Example use case: if you don't have a huge number of possible resources in your project then you could use this approach. The files will all download during the initial dialog, where participants input their details.

This is method requires no manual intervention, it is performed automatically. PsychoPy tries to work out what files you need in advance and codes these in to be fetched during the initial dialog box. To manage auto-detection, PsychoPy scans all your Components (e.g. Image Components, Sound Copmonents etc, and your conditions files) for signs of filenames that are being used and then adds those to a list of required resources for the study.

At times, PsychoPy won't be able to detect that you need a particular image or other resource, because you've specified the filename using code. For instance you may list the image name as being *"stim{N}.png".format(thisNumber)* but PsychoPy could never know what range of values *thisNumber* may take so it can't know what filenames to look for. In these cases you can manually extend the list of files that will be fetched using the *Extra Files* setting in the *Online* tab of the *Experiment Settings*.

7.5.2 Specifying "Additional Resources"

Example use case: You don't have that many resources (e.g. <200 pictures), but some of your resources might be specified through code or are conditional base on experimental group. The files will all download during the initial dialog, where participants input their details.

This is the only method of manually controlling resource loading that has been made easily available to users of PsychoPy prior to PsychoPy version 2022.1. If you don't have that many resources (e.g. <200 pictures), but you encounter an "Unknown Resource" error, this is probably the easiest fix (but we recommend you consider Methods 2 and 3 below). The reason the "Unknown Resource" error occurs is probably because you have a resource specified through code somewhere in your experiment (take for example the *"stim{N}.png".format(thisNumber)* case).

You can manually specify what resources your experiment will need when you *Configure the online settings of your experiment*. However, if you have a large number of files, we recommend you either *Specify the files to download at the start* using the *Resource Manager Component* or *Specify the files to download each trial* using *Static Component*.

7.5.3 Specify the files to download at the start

(added version 2022.1.0)

Example use case: fetch a whole set of movies, possibly a custom list, for the participant *while they read your instructions* (i.e. within the experiment rather than at the start). You can start them loading before the first instructions screen and then make sure they have all downloaded before the trials actually begin. You could even load yor files in two sets - download a few files during instructions and then fetch the rest during practice trials!

While the automatic method is easy, it suffers if you have lots of resources (the participant sits waiting on that dialog box while the resources are fetched) or if each participant uses only a subset of resources. PsychoPy has a new Component called the *Resource Manager Component* that allows you to specify the files you need and the time you want them to start and/or confirm downloading.

> ⚠ **Warning**
>
> If you use this method you do then need to list ALL of the files you want to download. PsychoPy won't do a combination of automatically fetching files as well as letting you specify them.

7.5.4 Specify the files to download each trial

(added version 2022.1.0)

Example use case: Fetch the image file during the fixation period on each trial.

If each resource can be retrieved relatively quickly (e.g. an imagefile over a broadband connection) then you might want to fetch the stimulus on each trial. This has the advantage that you don't need to prespecify anything and you could even choose the stimulus to download dynamically, based on the previous response!

The other nice thing about this method is that it can be used either using a *Resource Manager Component* Component, or by simply setting the stmulus to update using a *Static Component* where that Static Component lasts during your fixation period.

⚠ **Warning**

If you request a file and it takes too long to arrive then the onset of the trial will occur at a different time. PsychoPy will pause and will take this in to account in its response times etc. but if you absolutely must have the stimulus appear at very regular times then you should make sure you download your stimuli a long way in advance (as above) or with a very generous time window to allow for slower connections.

7.6 Media formats suitable for online studies

When you want to present images, sounds, or movies online, two things are important to take into account:

1. Web-browsers may support different formats than PsychoPy does
2. Because all media need to be downloaded via internet, it can be handy to use formats that compress your media, so that they produce smaller files.

Below are some recommended formats and pointers how to convert your media with free and open source software.

7.6.1 Images: PNG or JPG

Web-browsers support a large variety of image formats; see an overview here. Two widely supported formats are:

- PNG. This format applies “lossless” compression, which means that the compressed image is an exact reproduction of the original image. PNG is good at compressing pictures with geometric shapes, but natural scenes may yield relatively large files.
- JPG. This format applies “lossy” compression, which means that the compressed image approximates the original image. JPG can compress natural scenes very well. When encoding to JPG, you can adjust quality settings to produce larger (and more detailed) or smaller (and less detailed) files.

For converting images to PNG and JPG, you could use GIMP. See this tutorial about GIMP for instructions on how to convert multiple images at once using GIMP. By picking “Change Format and Compression” in step 4 of the tutorial you can select which format to save the images in.

7.6.2 Sounds: MP3

Here you can find an overview of audio formats supported by web browsers. MP3 is the most widely supported format. MP3 performs lossy compression, so the sound may lose some detail, but you can adjust the quality level. At higher qualities, the loss in detail is negligible.

For converting sound to MP3, you could use VLC Player. See this tutorial about VLC for instructions on how to convert multiple sounds at once using VLC.

NB - Presently, PsychoPy does not yet support MP3.

7.6.3 Movies: MP4 + H.264 & MP3

Here you can find an overview of video formats supported by web browsers. MP4 + H.264 + MP3 is the most widely supported format.

- MP4 is a format that can contain video and audio
- H.264 is a format that encodes video
- MP3 is format that encodes audio

Both H.264 and MP3 perform lossy compression, so the video and audio may lose some detail, but you can adjust the quality level. At higher qualities, the loss in detail is negligible.

For converting movies, you could use VLC Player. See this tutorial for instructions on how to convert multiple movies at once using VLC. To set up the output format correctly, we recommend making a new profile at step 4 in the tutorial above (see Fig. **??**):

1. Click the "New Profile" icon, then pick a name for your profile.
2. In the "Encapsulation" tab, select "MP4/MOV"
3. In the "Video codec" tab:
   a. Tick "Video" checkbox
   b. Select "H-264" as "Codec"
   c. Higher bitrates mean video that is of higher quality, but also larger files. Here are some bitrate guidelines
4. In the "Audio codec" tab:
   a. Tick the "Audio" checkbox
   b. Select "MP3" as "Codec"
   c. Higher bitrates mean audio that is of higher quality, but also larger files.
   d. For Sample Rate, 44100 Hz is a good choice
5. Finally, save your profile by clicking the "Create" button.

7.7 Media formats suitable for online studies

When you want to present images, sounds, or movies online, two things are important to take into account:

1. Web-browsers may support different formats than does
2. Because all media need to be downloaded via internet, it can be handy to use formats that compress your media, so that they produce smaller files.

Below are some recommended formats and pointers how to convert your media with free and open source software.

7.7.1 Images: PNG or JPG

Web-browsers support a large variety of image formats; see an overview here. Two widely supported formats are:

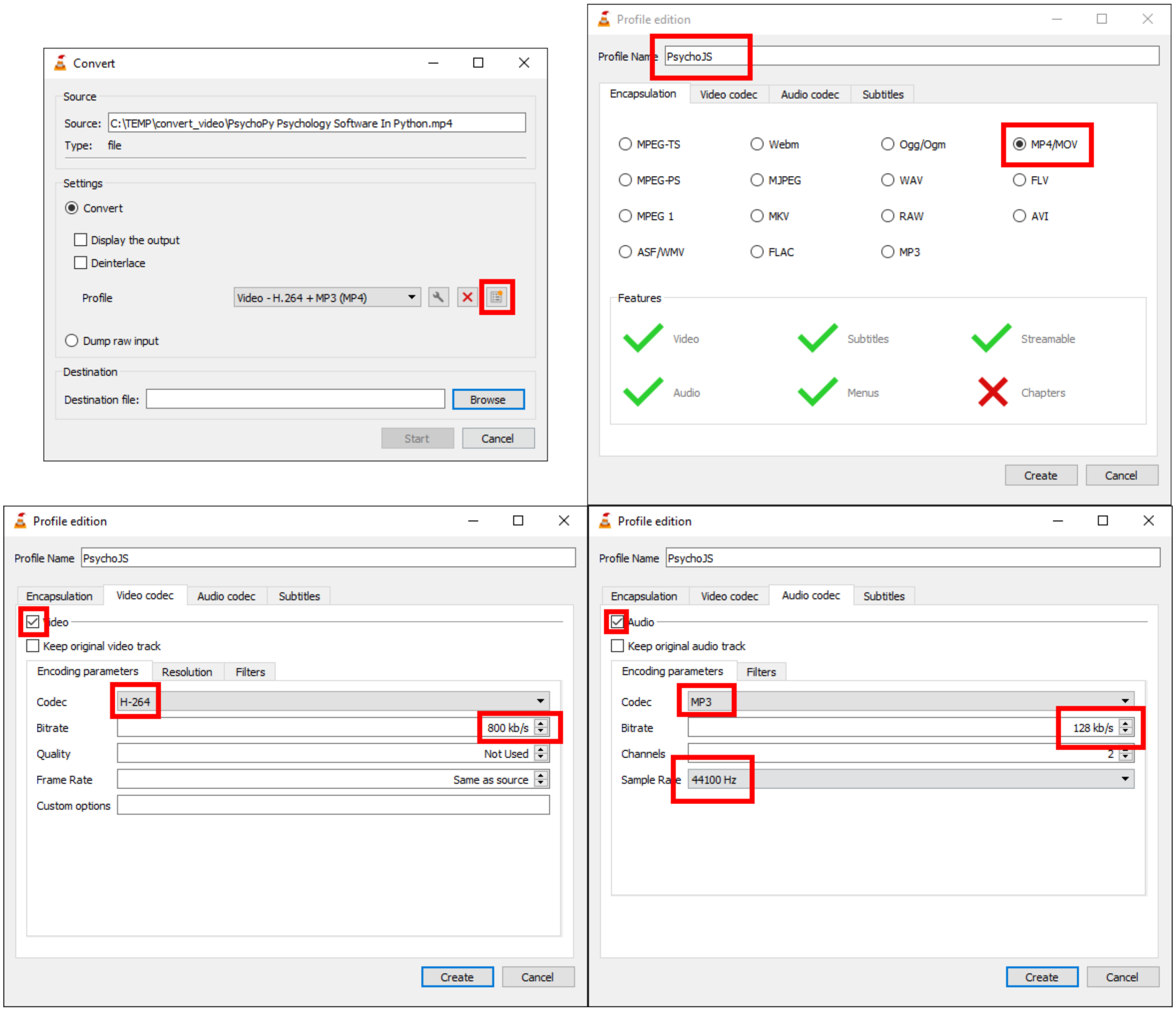
- PNG. This format applies "lossless" compression, which means that the compressed image is an exact reproduction of the original image. PNG is good at compressing pictures with geometric shapes, but natural scenes may yield relatively large files.
- JPG. This format applies "lossy" compression, which means that the compressed image approximates the original image. JPG can compress natural scenes very well. When encoding to JPG, you can adjust quality settings to produce larger (and more detailed) or smaller (and less detailed) files.

For converting images to PNG and JPG, you could use GIMP. See this tutorial for instructions on how to convert multiple images at once using GIMP. By picking "Change Format and Compression" in step 4 of the tutorial you can select which format to save the images in.

7.7.2 Sounds: MP3

Here you can find an overview of audio formats supported by web browsers. MP3 is the most widely supported format. MP3 performs lossy compression, so the sound may lose some detail, but you can adjust the quality level. At higher qualities, the loss in detail is negligible.

For converting sound to MP3, you could use VLC Player. See this tutorial for instructions on how to convert multiple sounds at once using VLC.



Convert

Source

Source: C:\TEMP\convert_video\PsychoPy Psychology Software In Python.mp4

Type: file

Settings

● Convert

□ Display the output

□ Deinterlace

Profile: Video - H.264 + MP3 (MP4)

○ Dump raw input

Destination

Destination file:

Profile edition

Profile Name: PsychoJS

Encapsulation

○ MPEG-TS ○ Webm ○ Ogg/Ogm ● MP4/MOV

○ MPEG-PS ○ MJPEG ○ WAV ○ FLV

○ MPEG 1 ○ MKV ○ RAW ○ AVI

○ ASF/WMV ○ FLAC ○ MP3

Features

✓ Video ✓ Subtitles ✓ Streamable

✓ Audio ✓ Menus ✗ Chapters

Profile edition

Profile Name: PsychoJS

Video codec

☑ Video

□ Keep original video track

Codec: H-264

Bitrate: 800 kb/s

Quality: Not Used

Frame Rate: Same as source

Custom options:

Profile edition

Profile Name: PsychoJS

Audio codec

☑ Audio

□ Keep original audio track

Codec: MP3

Bitrate: 128 kb/s

Channels: 2

Sample Rate: 44100 Hz

Fig. 7.12: Profile settings for encoding video to MP4 + H.264 & MP3

NB - Presently, does not yet support MP3.

7.7.3 Movies: MP4 + H.264 & MP3

Here you can find an overview of video formats supported by web browsers. MP4 + H.264 + MP3 is the most widely supported format.

- MP4 is a format that can contain video and audio
- H.264 is a format that encodes video
- MP3 is format that encodes audio

Both H.264 and MP3 perform lossy compression, so the video and audio may lose some detail, but you can adjust the quality level. At higher qualities, the loss in detail is negligible.

For converting movies, you could use VLC Player. See this tutorial for instructions on how to convert multiple movies at once using VLC. To set up the output format correctly, we recommend making a new profile at step 4 in the tutorial above (see Fig. **??**):

1. Click the "New Profile" icon, then pick a name for your profile.
2. In the "Encapsulation" tab, select "MP4/MOV"
3. In the "Video codec" tab:
   a. Tick "Video" checkbox
   b. Select "H-264" as "Codec"
   c. Higher bitrates mean video that is of higher quality, but also larger files. Here are some bitrate guidelines
4. In the "Audio codec" tab:
   a. Tick the "Audio" checkbox
   b. Select "MP3" as "Codec"
   c. Higher bitrates mean audio that is of higher quality, but also larger files.
   d. For Sample Rate, 44100 Hz is a good choice
5. Finally, save your profile by clicking the "Create" button.

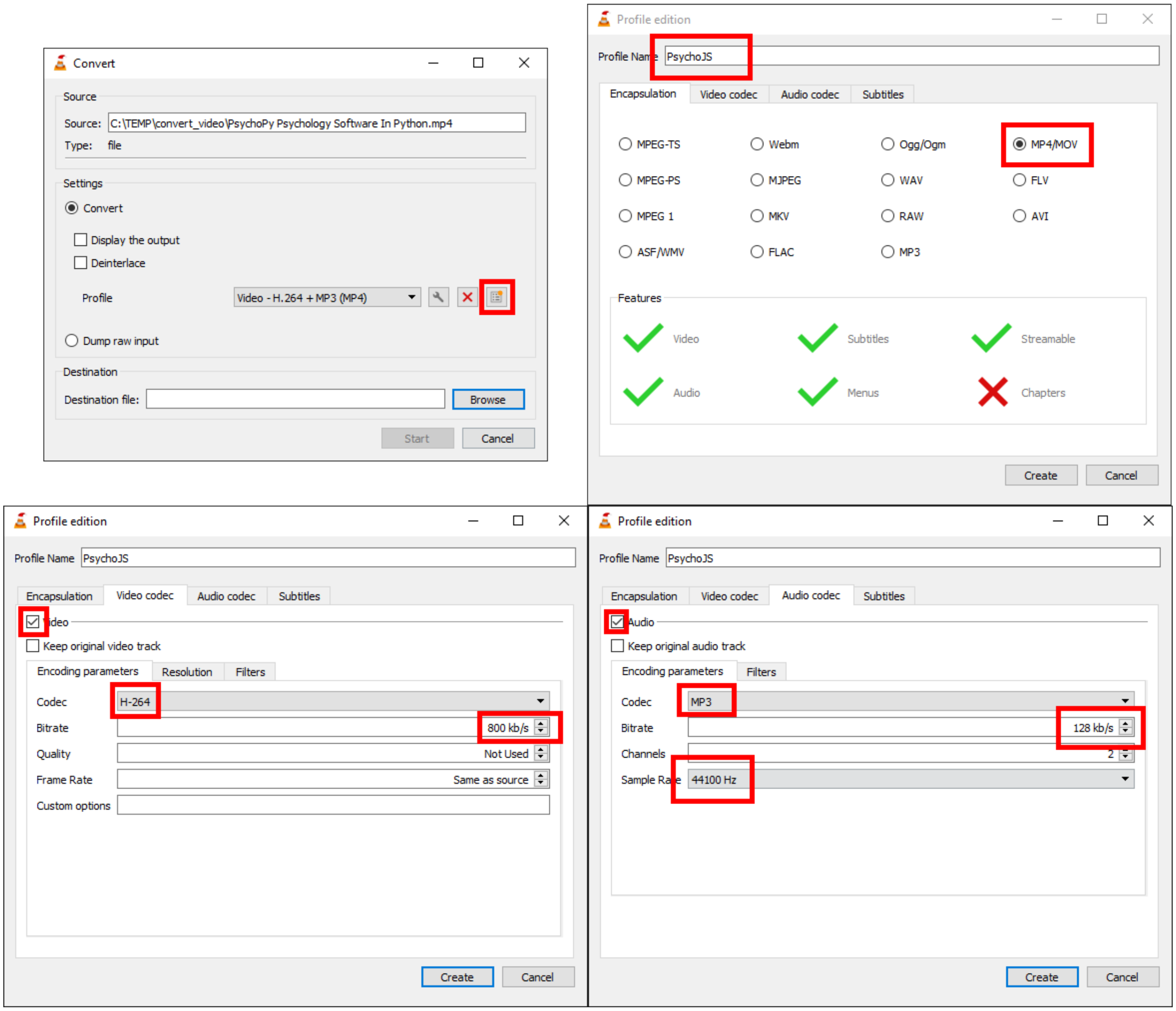
7.8 Using the Shelf for Multi-session testing, Counterbalancing and more online

PsychoPy Version 2022.2 or later required

Note: this is a new feature and currently in beta mode. Currently you can interact with the shelf through code components only. If you have feedback on use cases please share them *!*

The Shelf is a flexible, multiuse tool to aid with online studies where information needs sharing between studies, either in real time or across sessions. Use cases for the shelf include:

- *Multi-session testing*
- *Counterbalancing*
- *Multi-player games*
- *Leaderboards*



Convert

Source

Source: C:\TEMP\convert_video\PsychoPy Psychology Software In Python.mp4

Type: file

Settings

● Convert

□ Display the output

□ Deinterlace

Profile: Video - H.264 + MP3 (MP4)

○ Dump raw input

Destination

Destination file:

Profile edition

Profile Name: PsychoJS

○ MPEG-TS ○ Webm ○ Ogg/Ogm ● MP4/MOV

○ MPEG-PS ○ MJPEG ○ WAV ○ FLV

○ MPEG 1 ○ MKV ○ RAW ○ AVI

○ ASF/WMV ○ FLAC ○ MP3

Features

✓ Video ✓ Subtitles ✓ Streamable

✓ Audio ✓ Menus ✗ Chapters

Profile edition

Profile Name: PsychoJS

Video codec

☑ Video

□ Keep original video track

Codec: H-264

Bitrate: 800 kb/s

Quality: Not Used

Frame Rate: Same as source

Custom options:

Profile edition

Profile Name: PsychoJS

Audio codec

☑ Audio

□ Keep original audio track

Codec: MP3

Bitrate: 128 kb/s

Channels: 2

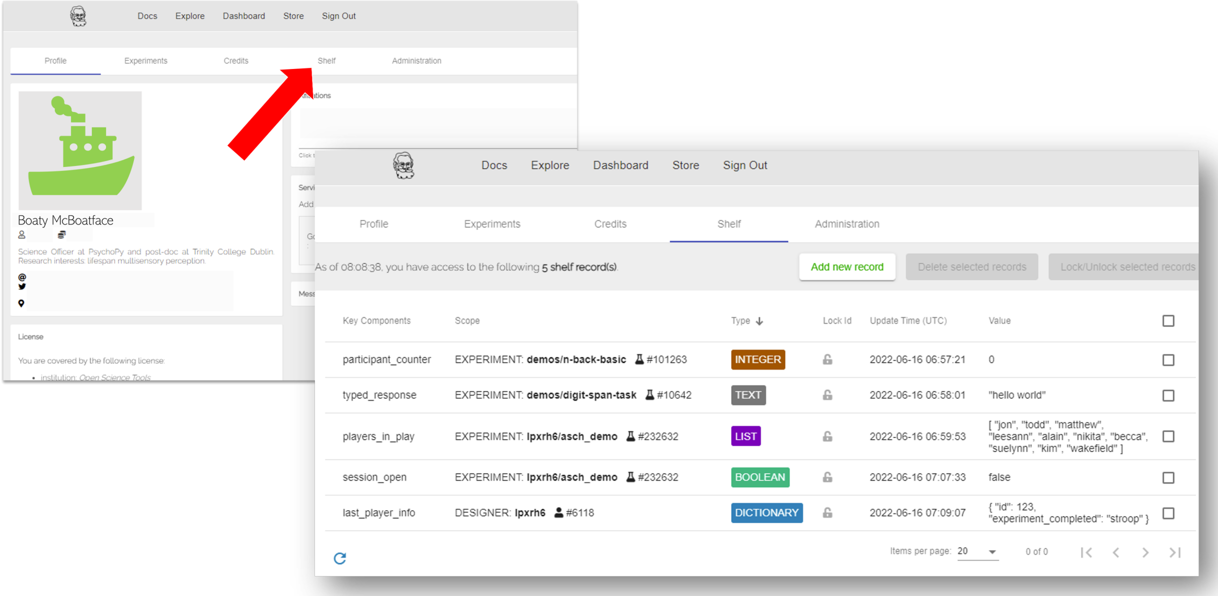
Sample Rate: 44100 Hz

Fig. 7.13: Profile settings for encoding video to MP4 + H.264 & MP3

Here we will walk through some of the use cases and how to implement them. At the moment, we must interact with the Shelf through *Code Components*. In the longer term, once we better understand the ways in which scientists are using the Shelf, we hope to make this more accessible by making a *Builder* component.

7.8.1 Basic examples

You can access the Shelf in your Pavlovia.org account by selecting Dashboard > Shelf.



Boaty McBoatface

Science Officer at PsychoPy and post-doc at Trinity College Dublin. Research interests: lifespan multisensory perception.

License

You are covered by the following license:

- institution: *Open Science Tools*

As of 08:08:38, you have access to the following **5 shelf record(s)**.

Key Components	Scope	Type	Lock Id	Update Time (UTC)	Value	□
participant_counter	EXPERIMENT: **demos/n-back-basic** #101263	INTEGER		2022-06-16 06:57:21	0	□
typed_response	EXPERIMENT: **demos/digit-span-task** #10642	TEXT		2022-06-16 06:58:01	"hello world"	□
players_in_play	EXPERIMENT: **lpxrh6/asch_demo** #232632	LIST		2022-06-16 06:59:53	["jon", "todd", "matthew", "leesann", "alain", "nikita", "becca", "suelynn", "kim", "wakefield"]	□
session_open	EXPERIMENT: **lpxrh6/asch_demo** #232632	BOOLEAN		2022-06-16 07:07:33	false	□
last_player_info	DESIGNER: **lpxrh6** #6118	DICTIONARY		2022-06-16 07:09:07	{ "id": 123, "experiment_completed": "stroop" }	□

How to access the Shelf from your pavlovia.org account. To begin with your Shelf will be empty. The value of each Record is a Json format, so be careful when formatting (that means use double quotations rather than single quotations!).

You can then add one or several "Records" to your Shelf. Each Record can be one of several variable types; Integer, Boolean, Text, List or Dictionary. The type of Record you create is up to you and will depend on the type of experiment you are trying to create. Each Record can be made available either to a single *Experiment* or to the *Designer*, meaning this Record is available to all of your experiments (for instance if you would like several experiments to interact with one another). Each Record can also be unlocked, meaning it can be interacted with and edited by your experiments, or locked, meaning it is frozen and not open to further edits. The way in which you choose to use Shelf is flexible and up to you! but we walk through some guidance to get started below.

Interacting with Integer Records

Demo link

Demo experiment files

Imagine the simple case of wanting to count how many participants have completed your task. You would make an Integer Record, which starts at 0 and assign the scope of the Record to the experiment of interest.

From within your experiment you can use several methods to interact with Integers including (though not limited to; see all methods here):

- `psychoJS.shelf.getIntegerValue()`
- `psychoJS.shelf.setIntegerValue()`
- `psychoJS.shelf.addIntegerValue()`

We can therefore add a code component to our experiment, **make the code component type JS** and use `psychoJS.shelf.addIntegerValue({key: ['participant_counter'], delta: 1})` where `["participant_counter"]` corresponds to the key name of our Record, and 1 is the amount we wish to increment by. You might want to use this code snippet in the *Begin Experiment* tab if you want to increment your participant counter at the start of the experiment, or the *End Experiment* tab if you wish to increment at the end of the session. If we wanted to fetch the value we would use `participantN = await psychoJS.shelf.getIntegerValue({key: ['participant_counter'], defaultValue: 0})` **Note that it is important to use** `await` this is because these functions are known and JavaScript Promises, and we must wait for the Promise to be fulfilled until we have the value to display.

Interacting with Boolean Records

Demo link

Demo experiment files

Boolean Records are perhaps the easiest to interact with, by means that they only have two values (`true` or `false`, and therefore have a limited number of ways in which you interact with them. The most useful methods you may use when using Boolean Records are:

- `psychoJS.shelf.getBooleanValue()`
- `psychoJS.shelf.setBooleanValue()`
- `psychoJS.shelf.flipBooleanValue()`

Imagine you have an experiment that can be "opened" or "closed" by a host. You could add a Record called "session_open", ensure it is Boolean, and in your experiment make it such that the participant can sign in as a host (with the power to open/close the session) or as a participant (who, for now, passively watches the session opening or closing).

In our experiment we could get the session status and show it by adding a code component (ensure it's code type is JS) and using `psychoJS.shelf.getBooleanValue(["session_open"])`. We can allow the host to open or close the session using a simple routine with a response component (in our demo we use a mouse) and in the End Routine tab using `psychoJS.shelf.flipBooleanValue(["session_open"])`. In a separate routine (the one the participant views) we might repeatedly check what the value of the "session_open" record is so that we can use it to control something in our experiment, in our case, a picture of a door that opens/closes.

Interacting with Text Records

Demo link

Demo experiment files

Reading and writing Text Records from the Shelf requires two main functions:

- `psychoJS.shelf.getTextValue()`
- `psychoJS.shelf.setTextValue()`

Quite simply - we use these to check the text currently on the shelf and fetch it respectively!

Interacting with List Records

Demo link

Demo experiment files

Imagine you have an experiment where you wish for many players to interact with one another. To start with, you might want a list of players and you might want all players who are signed in to be able to see other players screen names. To achieve this, you would add Record to your Shelf and set the type to List. When interacting with this Record from your experiment, the main functions of interest are:

- psychoJS.shelf.getListValue()
- psychoJS.shelf.setListValue()
- psychoJS.shelf.appendListValue()
- psychoJS.shelf.popListValue()

First imagine you want to allow the player to clear the list of preexisting players (in our demo we achieve this though a drop down). We would do that using `psychoJS.shelf.setListValue({key: ["player_list"], value: []})`. Then imagine we want to add this players screen name to the existing list of screen names, that is achieved using `psychoJS.shelf.appendListValue({key: ["player_list"], elements: expInfo["screen name"]})` finally, to fetch the screen names (and we may wish to do this periodically) we can use `players = await psychoJS.shelf.getListValue({key: ["player_list"], defaultValue:[]})` (remember, it is important to use `await` in order to retrieve the value once the JS Promise has been fulfilled.

Interacting with Dictionary Records

Demo link

Demo experiment files

The main functions of interest for use with a Dictionary Record are:

- `psychoJS.shelf.setDictionaryFieldValue()` - for *either* updating an existing Dictionary field *or* creating a new Dictionary field within the Dictionary Record.
- `psychoJS.shelf.getDictionaryFieldValue()` - for fetching the value associated with a specific Dictionary field

When you create a Dictionary Record, that dictionary is blank. Imagine we want to populate this with two things, a list of player names and a list of completed sessions (note, we could actually achieve this same thing through two List Records, but for this example we will stick to a single Dictionary). In our demo when a participant joins, we want to check how many times they have joined previously, and increment that to consider this session.

First, we check, has this participant taken part at all? We can do that by checking the existing fieldnames in our Dictionary Record `existing_participants = await psychoJS.shelf.getDictionaryFieldNames({key: ["session_tracker"]})` (where "session_tracker" is the name of our Dictionary Record). Then, we check if this participant ID (retrieved from the startup gui) exists in `existing_participants`. If not, add this participant to the Dictionary, otherwise, increment the existing value associated with this participant:

```
if(!existing_participants.includes(expInfo['participant'])){
    psychoJS.shelf.setDictionaryFieldValue({key: ["session_tracker"], fieldName: expInfo[
↪'participant'], fieldValue :1})
    }else{
        //increase the number of sessions completed by this participant
        session_number = await psychoJS.shelf.getDictionaryFieldValue({key: ["session_
↪tracker"], fieldName:expInfo['participant'], defaultValue:'no sessions detected'})
        session_number = session_number + 1
        console.log('session_number2', session_number)
        psychoJS.shelf.setDictionaryFieldValue({key: ["session_tracker"], fieldName:␣
↪expInfo['participant'], fieldValue :session_number})
    }
```

Note

Remember to watch the capitalisation of functions when interacting with the shelf e.g. `fieldName` rather than `fieldname` .

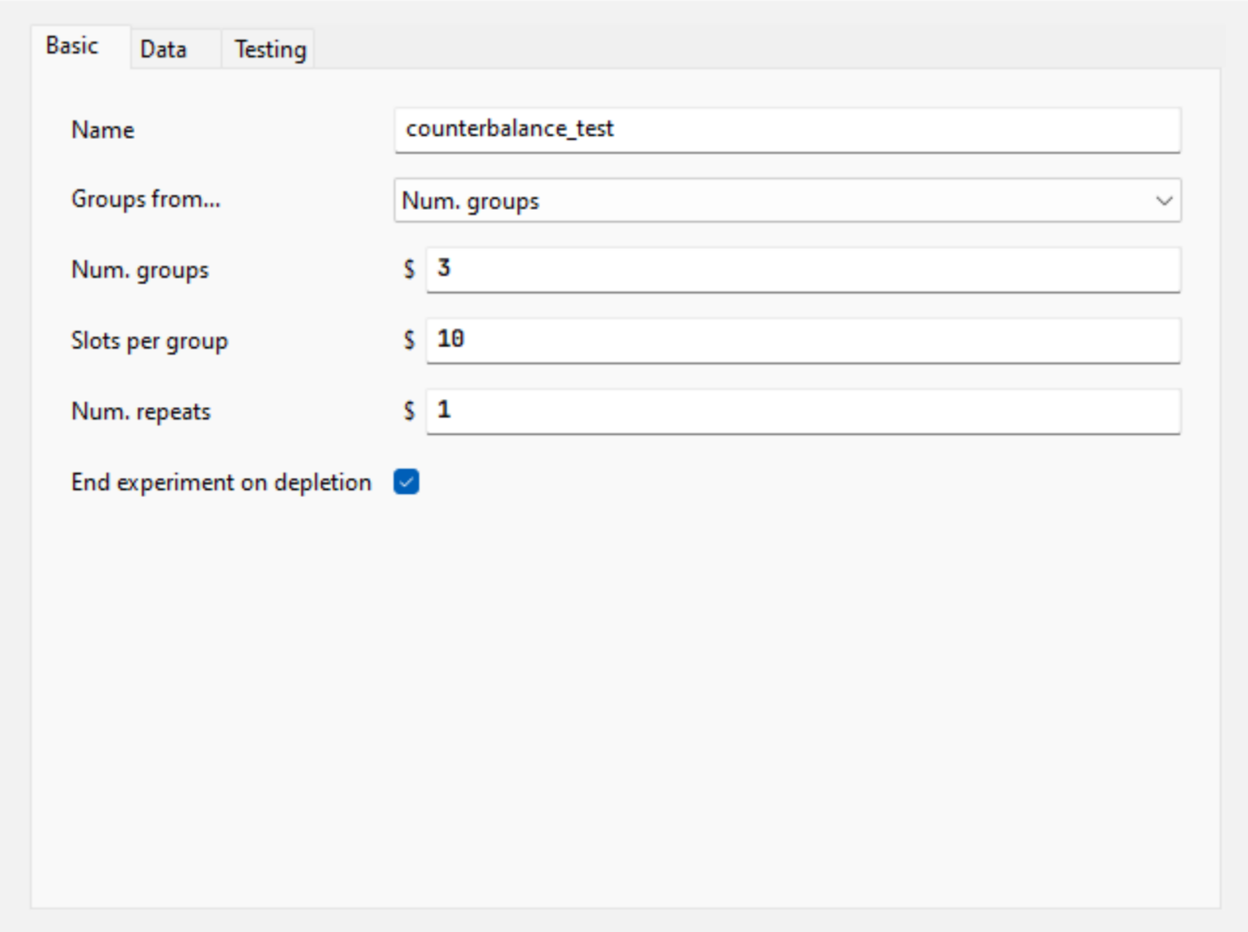
Counterbalancing

PsychoPy Version 2024.1 or later required

Demo link

Demo experiment files

We now have a Counterbalance Routine where you can set up your counterbalance groups in Builder Mode and interact with the Shelf with the record type, Counterbalance.



Name: counterbalance_test

Groups from...: Num. groups

Num. groups: $ 3

Slots per group: $ 10

Num. repeats: $ 1

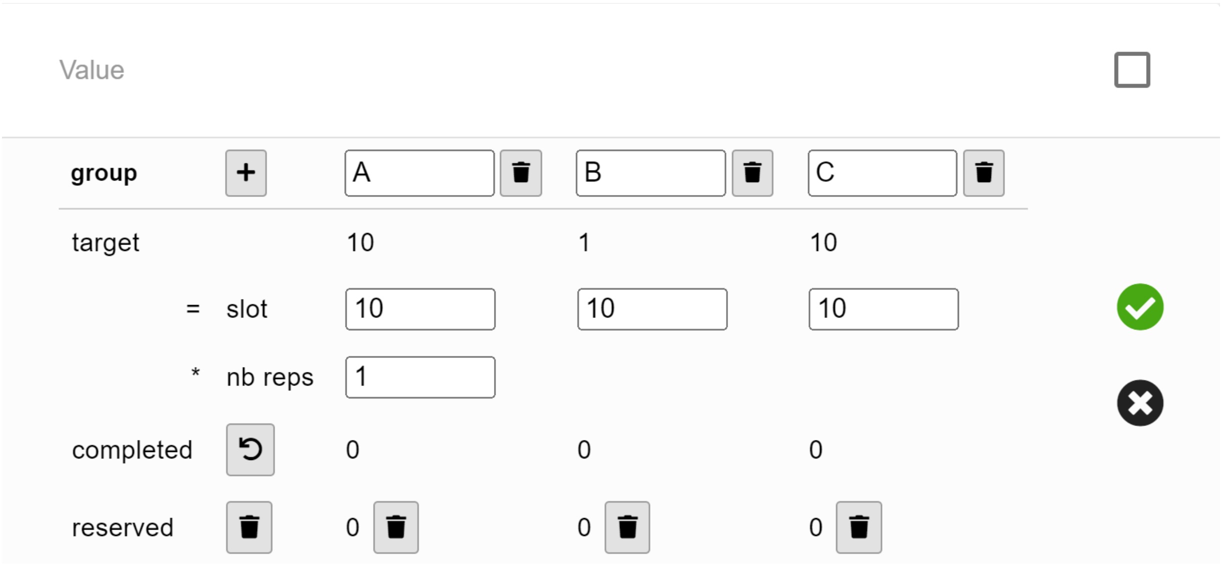
End experiment on depletion ☑

To set up your Counterbalance Shelf, you would need to first upload your task to Pavlovia and set it to Pilot/Running Mode.

In your Shelf view of your Dashboard, click on Add Record. In Key, add the name of your Counterbalance Routine as in your Builder task. For Scope, choose Experiment and select the name of your Builder task. For Type, select Counterbalance (*you might need to scroll down*).

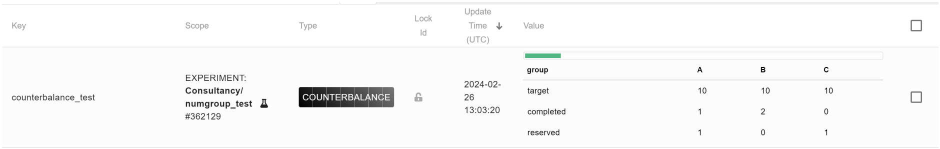
Once you click on Ok, you will see an empty table in Value. Here, set up the same group parameters as in your Builder task.

Your resulting Shelf record should look like this:



Value □

group	+	A	B	C	
target		10	1	10	
= slot		10	10	10	✓
* nb reps		1			✖
completed	↺	0	0	0	
reserved	🗑	0 🗑	0 🗑	0 🗑	



Key	Scope	Type	Lock Id	Update Time ↓ (UTC)	Value	□
counterbalance_test	EXPERIMENT: **Consultancy/ numgroup_test** #362129	COUNTERBALANCE		2024-02-26 13:03:20	group A B C target 10 10 10 completed 1 2 0 reserved 1 0 1	□

> **See also**
>
> *Counterbalance Routine*

Multiplayer experiments

Demo link (Note: Requires multiple browser windows or multiple people accessing the link at once!)

Demo experiment files

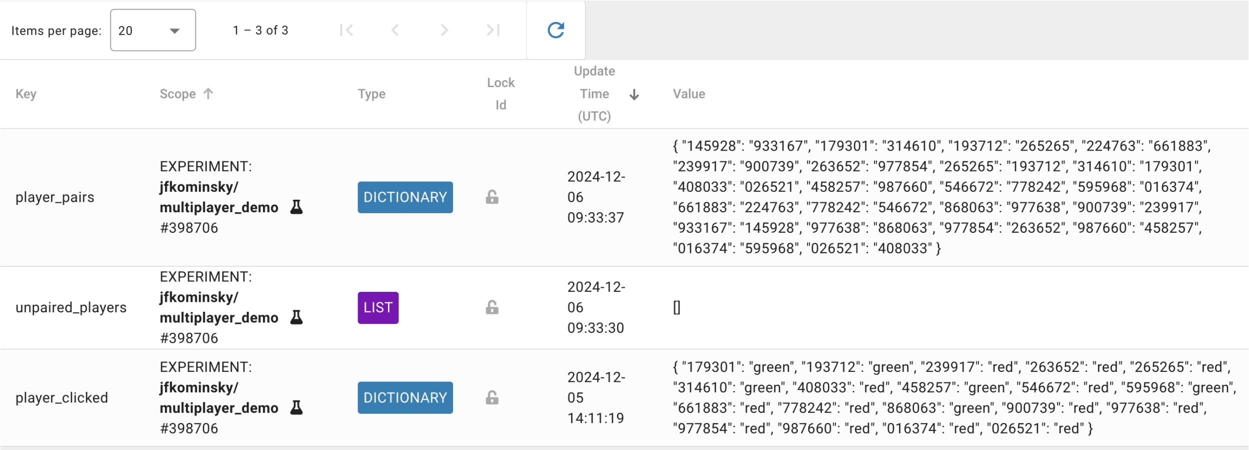
Using the shelf, you can create synchronous multi-participant experiments, i.e., multiplayer activities. First, two important notes:

- The shelf is **slow**. It can take up to 60 seconds for an update from one participant to be received by another participant via updating and checking the shelf.
- If you are planning to copy the demo, you will need to set up your own lists and dictionaries (described below) in your own account's shelf.

Multiplayer activities first require a matchmaking routine to pair a participant with another participant (or, in principle, multiple other participants), and once the other player has been found, whatever communication is required by the task itself. This demo uses a very simple coordination game in which each player must select a red or green card, and both players win if they choose the same color.

The demo requires three entries on the shelf (which you will need to re-create on your own shelf if you want to create your own copy of the demo):

1. `player_pairs`: A **dictionary** consisting of key:value pairs where one player ID is the key and the partnered player is the value. This contains one key per player, so each pair has two entries in this dictionary. This keeps track of the matches once they have been made.
2. `unpaired_players`: A **list** of Pavlovia IDs for players who have not yet been matched up.

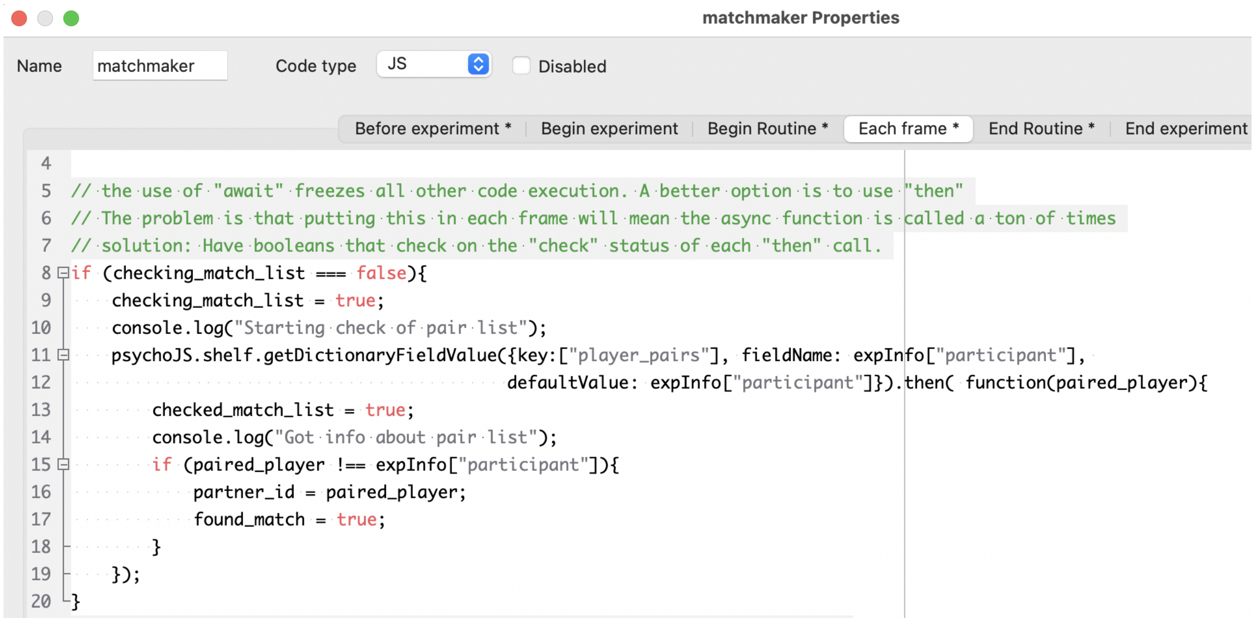


Key	Scope ↑	Type	Lock Id	Update Time ↓ (UTC)	Value
player_pairs	EXPERIMENT: **jfkominsky/ multiplayer_demo** #398706	DICTIONARY		2024-12-06 09:33:37	{ "145928": "933167", "179301": "314610", "193712": "265265", "224763": "661883", "239917": "900739", "263652": "977854", "265265": "193712", "314610": "179301", "408033": "026521", "458257": "987660", "546672": "778242", "595968": "016374", "661883": "224763", "778242": "546672", "868063": "977638", "900739": "239917", "933167": "145928", "977638": "868063", "977854": "263652", "987660": "458257", "016374": "595968", "026521": "408033" }
unpaired_players	EXPERIMENT: **jfkominsky/ multiplayer_demo** #398706	LIST		2024-12-06 09:33:30	[]
player_clicked	EXPERIMENT: **jfkominsky/ multiplayer_demo** #398706	DICTIONARY		2024-12-05 14:11:19	{ "179301": "green", "193712": "green", "239917": "red", "263652": "red", "265265": "red", "314610": "green", "408033": "red", "458257": "green", "546672": "red", "595968": "green", "661883": "red", "778242": "red", "868063": "green", "900739": "red", "977638": "red", "977854": "red", "987660": "red", "016374": "red", "026521": "red" }

3. `player_clicked`: A **dictionary** consisting of key:value pairs where one player ID is the key and the color that player clicked (red or green) is the value.

The experiment starts with a matchmaking routine. This routine displays a message to the participant to let them know that matchmaking is happening, a timer to show them that the experiment is not frozen, and a code component that conducts matchmaking in three repeating steps:

1. Check if the current player's Pavlovia ID is already present as a key in the `player_pairs` dictionary. If so, record the value of that entry as the partner ID.



Name: matchmaker

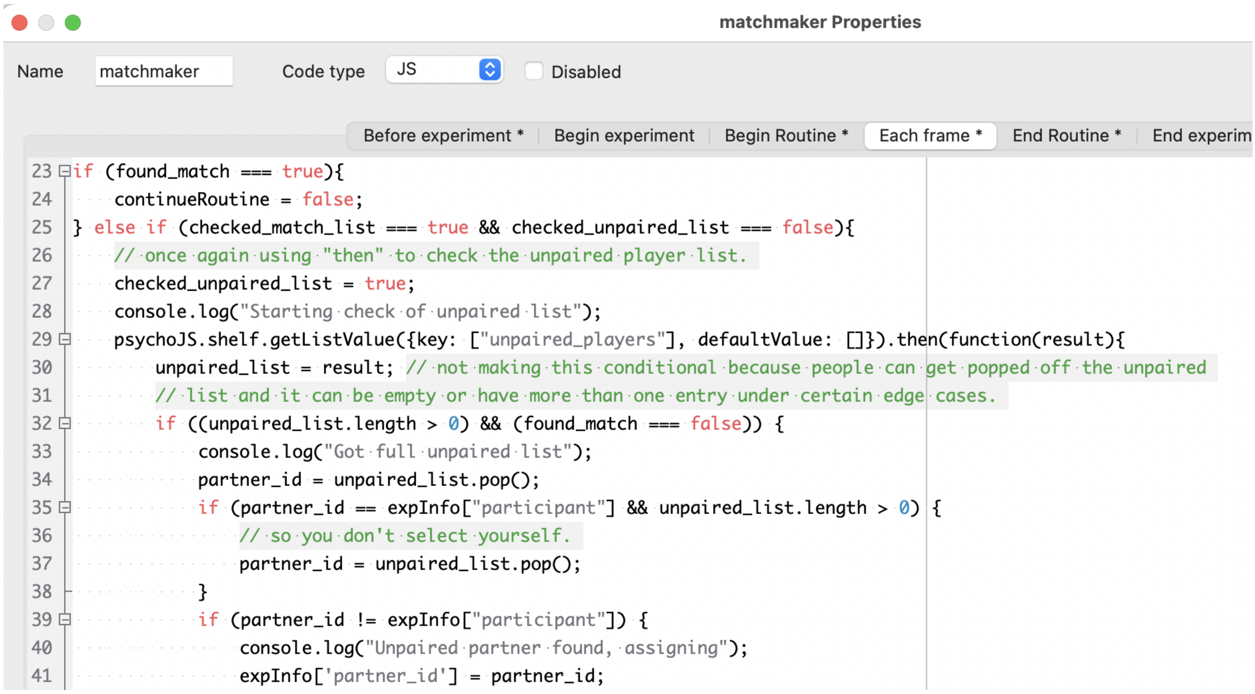
Code type: JS

☐ Disabled

```

// the use of "await" freezes all other code execution. A better option is to use "then"
// The problem is that putting this in each frame will mean the async function is called a ton of times
// solution: Have booleans that check on the "check" status of each "then" call.
if (checking_match_list === false){
    checking_match_list = true;
    console.log("Starting check of pair list");
    psychoJS.shelf.getDictionaryFieldValue({key:["player_pairs"], fieldName: expInfo["participant"],
                                           defaultValue: expInfo["participant"]}).then( function(paired_player){
        checked_match_list = true;
        console.log("Got info about pair list");
        if (paired_player !== expInfo["participant"]){
            partner_id = paired_player;
            found_match = true;
        }
    });
}
```

2. If the current player's Pavlovia ID is not a key in the `player_pairs` dictionary, check the `unpaired_players` list. If there is already an unpaired player listed, record them as the partner ID, remove them from the `unpaired_players` list, and add two matched entries to `player_pairs`, one for the current player and one for the partner.
3. If the `unpaired_players` list is empty, add the current player's ID to the `unpaired_players` list, and repeat from step 1.

To prevent the code from freezing every time it checks the shelf, this code uses `.then(function(result){})` asynchronous code. The difference between this and the `await` keyword used with other shelf demos is that `await` freezes

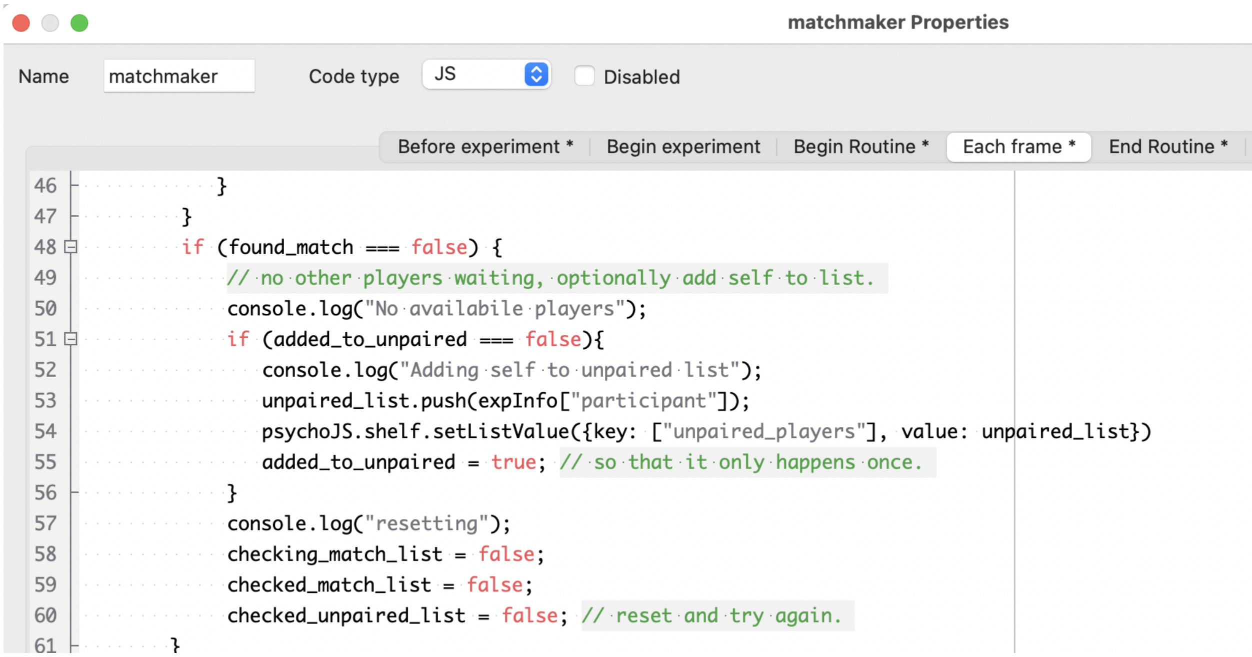


Name matchmaker

Code type JS

□ Disabled

```
if (found_match === true){
    continueRoutine = false;
} else if (checked_match_list === true && checked_unpaired_list === false){
    // once again using "then" to check the unpaired player list.
    checked_unpaired_list = true;
    console.log("Starting check of unpaired list");
    psychoJS.shelf.getListValue({key: ["unpaired_players"], defaultValue: []}).then(function(result){
        unpaired_list = result; // not making this conditional because people can get popped off the unpaired
        // list and it can be empty or have more than one entry under certain edge cases.
        if ((unpaired_list.length > 0) && (found_match === false)) {
            console.log("Got full unpaired list");
            partner_id = unpaired_list.pop();
            if (partner_id == expInfo["participant"] && unpaired_list.length > 0) {
                // so you don't select yourself.
                partner_id = unpaired_list.pop();
            }
            if (partner_id != expInfo["participant"]) {
                console.log("Unpaired partner found, assigning");
                expInfo['partner_id'] = partner_id;
```



Name matchmaker

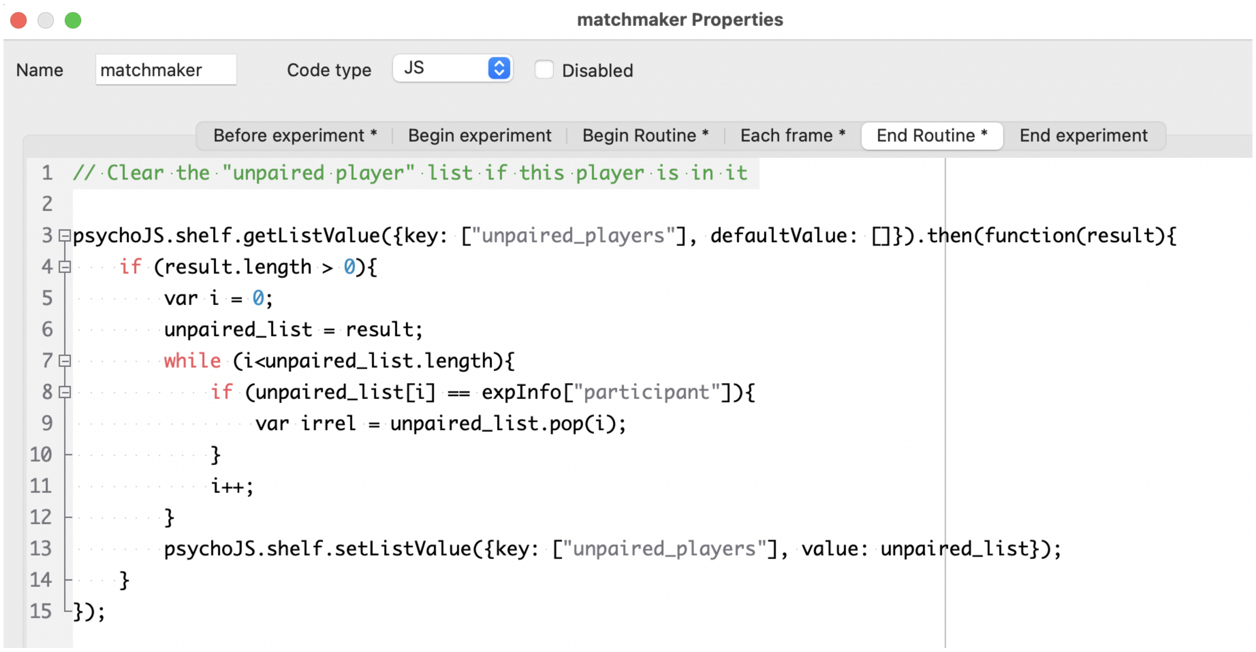
Code type JS

□ Disabled

```
            }
        }
        if (found_match === false) {
            // no other players waiting, optionally add self to list.
            console.log("No available players");
            if (added_to_unpaired === false){
                console.log("Adding self to unpaired list");
                unpaired_list.push(expInfo["participant"]);
                psychoJS.shelf.setListValue({key: ["unpaired_players"], value: unpaired_list})
                added_to_unpaired = true; // so that it only happens once.
            }
            console.log("resetting");
            checking_match_list = false;
            checked_match_list = false;
            checked_unpaired_list = false; // reset and try again.
        }
```

the rest of the experiment until the communication with the shelf is complete, whereas `.then(function(result){})` executes when the communication is complete while letting the frame loop continue uninterrupted in the meantime. Note that in step 3, it uses neither of these, which means that the code **does not know when the shelf has finished updating with the current player's ID added to the unpaired player list**. In this case that's fine because nothing in this code depends on that completing, and there is a boolean that makes sure that the player's ID is only added to the list once regardless of whether or not the update has finished from one frame to the next.

As a safety measure to ensure that the unpaired player list is cleared, the code component also includes an "end routine" step that makes sure the current player ID is removed from the list. Because it does not use `await`, this can happen while the experiment proceeds to the next routine.

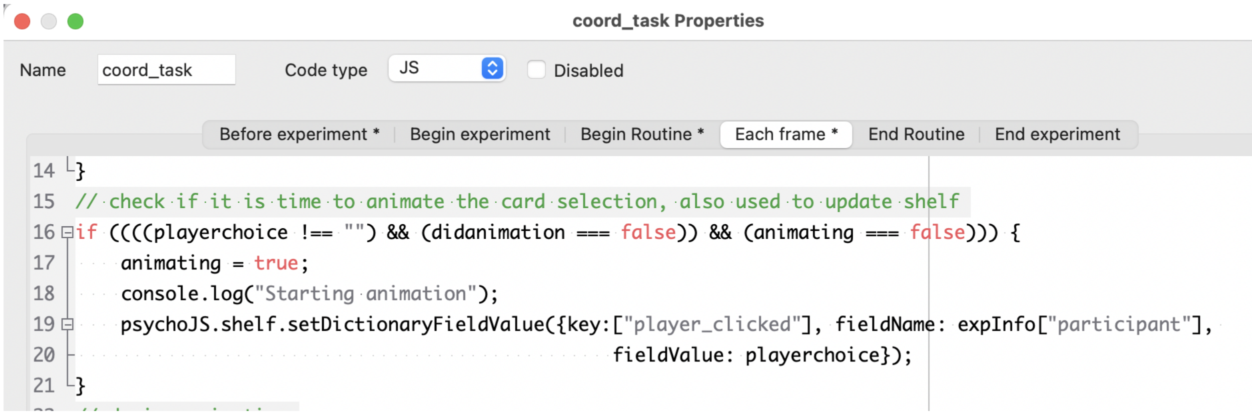


Name matchmaker Code type JS □ Disabled

```
// Clear the "unpaired player" list if this player is in it

psychoJS.shelf.getListValue({key: ["unpaired_players"], defaultValue: []}).then(function(result){
    if (result.length > 0){
        var i = 0;
        unpaired_list = result;
        while (i<unpaired_list.length){
            if (unpaired_list[i] == expInfo["participant"]){
                var irrel = unpaired_list.pop(i);
            }
            i++;
        }
        psychoJS.shelf.setListValue({key: ["unpaired_players"], value: unpaired_list});
    }
});
```

After a partner has been found, the participant moves on to the coordination game, where they can click either the red or green card. Two black cards on the other side of the screen represent the partner's cards. The code component in this routine does three things:

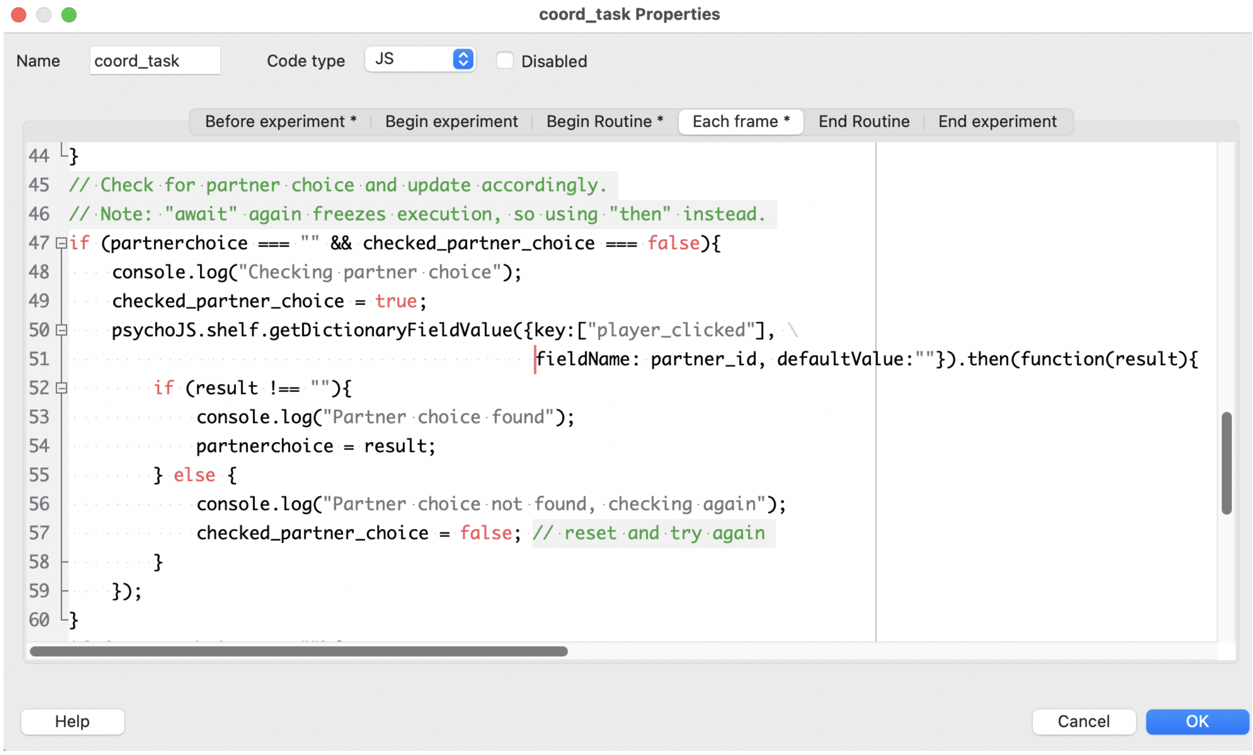
1. When the current player clicks a card, add an entry to the `player_clicked`: dictionary with the current player's ID as a key and the value equal to the color that the current player clicked.



Name coord_task Code type JS □ Disabled

```
}
// check if it is time to animate the card selection, also used to update shelf
if ((((playerchoice !== "") && (didanimation === false)) && (animating === false))) {
    animating = true;
    console.log("Starting animation");
    psychoJS.shelf.setDictionaryFieldValue({key:["player_clicked"], fieldName: expInfo["participant"],
                                              fieldValue: playerchoice});
}
```

2. Check whether the partner ID has appeared as a key in the `player_clicked` dictionary, and if so, animate one of the partner's cards (always the left one) as moving toward the center (and record what color they actually chose in a separate variable).



Name: coord_task

Code type: JS

□ Disabled

```
}
// Check for partner choice and update accordingly.
// Note: "await" again freezes execution, so using "then" instead.
if (partnerchoice === "" && checked_partner_choice === false){
    console.log("Checking partner choice");
    checked_partner_choice = true;
    psychoJS.shelf.getDictionaryFieldValue({key:["player_clicked"], \
                                            fieldName: partner_id, defaultValue:""}).then(function(result){
        if (result !== ""){
            console.log("Partner choice found");
            partnerchoice = result;
        } else {
            console.log("Partner choice not found, checking again");
            checked_partner_choice = false; // reset and try again
        }
    });
}
```

3. When both (1) and (2) have occurred, end routine and go to the routine that presents the outcome.

The logic here once again uses `.then(function(result){})` to make sure each player can make their own choice without the code freezing waiting for the other player's choice.

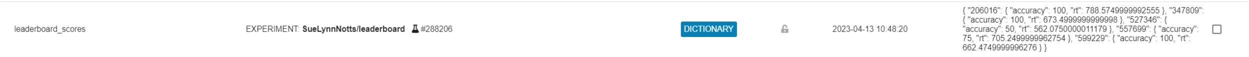
Leaderboard

Demo link

Demo experiment files

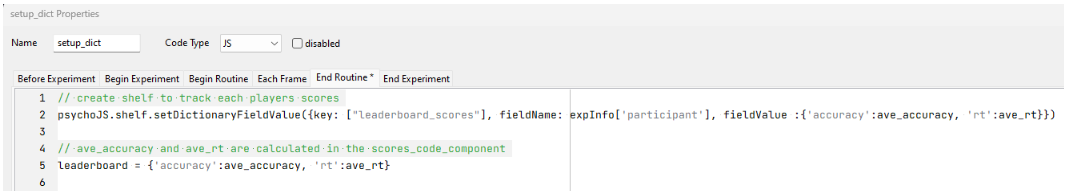
Leaderboards are a fun way of adding an element of gamification to your tasks! You can do this by using a Dictionary type shelf record. Just like in the counterbalancing example, the Key Component (on your Pavlovia shelf) and the `key` within the code component of your PsychoPy task needs to match and have a meaningful name. Since the demo task records both the reaction times and accuracy data, the name used is "leaderboard_scores".

You would not need to add any fields within the shelf record on Pavlovia as they will automatically be populated when the task is completed. As more people complete the task, the shelf record would look like so:



leaderboard_scores	EXPERIMENT: **SueLynnNotts/leaderboard** #288206	DICTIONARY		2023-04-13 10:48:20	{ "206016": { "accuracy": 100, "rt": 788.5749999992555 }, "347809": { "accuracy": 100, "rt": 673.4999999999998 }, "527346": { "accuracy": 50, "rt": 562.0750000011179 }, "557699": { "accuracy": 75, "rt": 705.2499999962754 }, "599229": { "accuracy": 100, "rt": 662.4749999996276 } }	□

If you would like to just record each participants' scores, you would only need the following code component:



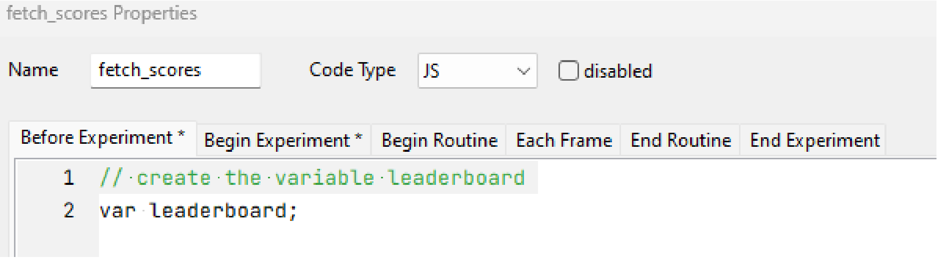
Name: setup_dict

Code Type: JS

□ disabled

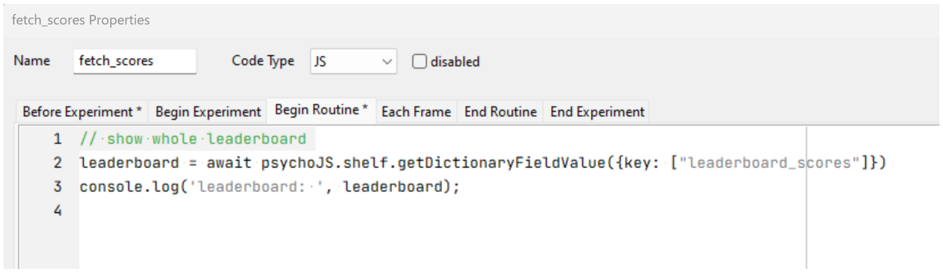
```
// create shelf to track each players scores
psychoJS.shelf.setDictionaryFieldValue({key: ["leaderboard_scores"], fieldName: expInfo['participant'], fieldValue :{'accuracy':ave_accuracy, 'rt':ave_rt}})

// ave_accuracy and ave_rt are calculated in the scores_code_component
leaderboard = {'accuracy':ave_accuracy, 'rt':ave_rt}

```

This is how you would fetch all the records that's stored within the leaderboard.



Name: fetch_scores
Code Type: JS
☐ disabled

```
// create the variable leaderboard
var leaderboard;
```

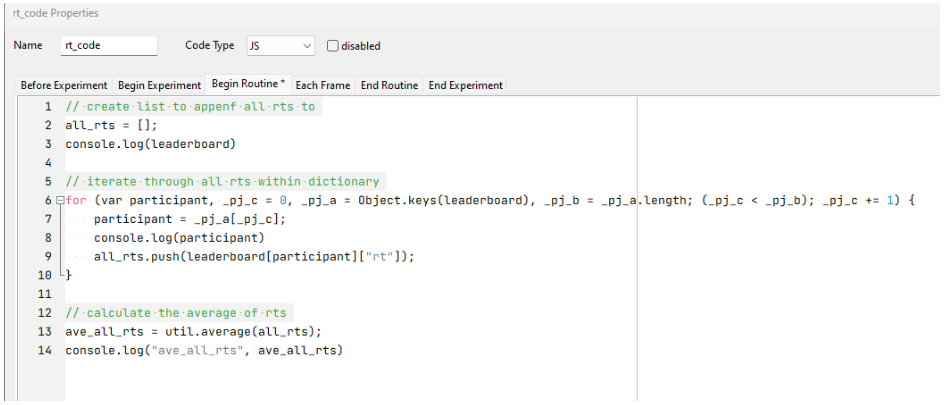


Name: fetch_scores
Code Type: JS
☐ disabled

```
// show whole leaderboard
leaderboard = await psychoJS.shelf.getDictionaryFieldValue({key: ["leaderboard_scores"]})
console.log('leaderboard: ', leaderboard);

```

Average Reaction Times

This is an example JavaScript snippet to fetch all the reaction times recorded and calculate the average reaction times:



Name: rt_code
Code Type: JS
☐ disabled

```
// create list to appenf all rts to
all_rts = [];
console.log(leaderboard)

// iterate through all rts within dictionary
for (var participant, _pj_c = 0, _pj_a = Object.keys(leaderboard), _pj_b = _pj_a.length; (_pj_c < _pj_b); _pj_c += 1) {
    participant = _pj_a[_pj_c];
    console.log(participant)
    all_rts.push(leaderboard[participant]["rt"]);
}

// calculate the average of rts
ave_all_rts = util.average(all_rts);
console.log("ave_all_rts", ave_all_rts)
```

Ranked Accuracy

This is an example JavaScript snippet to fetch all the accuracy stored and sort them in descending order:

The above code component only sorts the accuracies of each participant but doesn't return the participants' IDs. To get the sorted IDs, you would need the following code component:

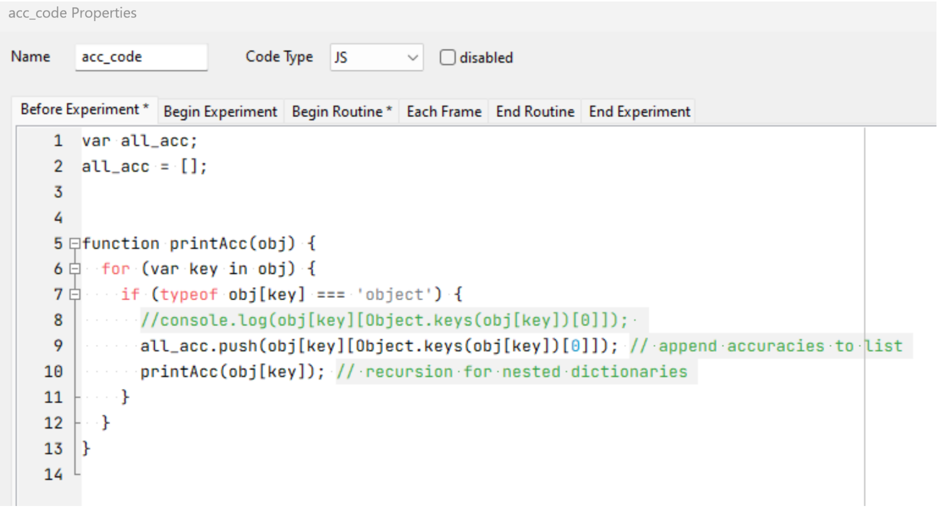
The IDs and accuracy scores are stored in the separate lists (in descending order) and therefore can be indexed. In this example, we index the first 5 IDs and accuracy scores.

Checking existing participant IDs

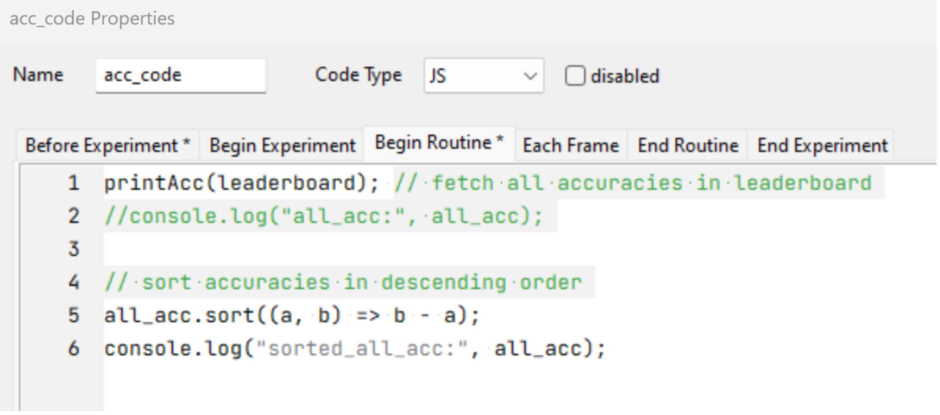
Demo link

Demo experiment files

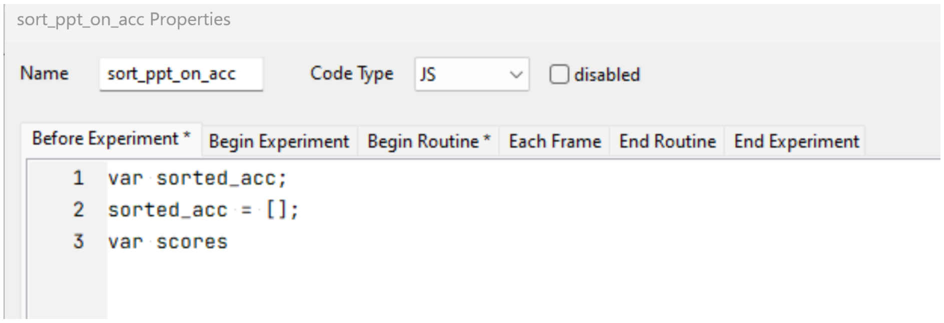
When running multi-session experiments online, it is sometimes difficult to tell if the person accessing the link is a participant from a previous session. This participant ID checker using the List type Shelf uses a prepopulated list of IDs to first check if the participant ID entered at the startup dialog box exists in the prepopulated list (see list below for accepted IDs) before either showing a message saying "Welcome back!" or "Sorry, your id couldn't be found."



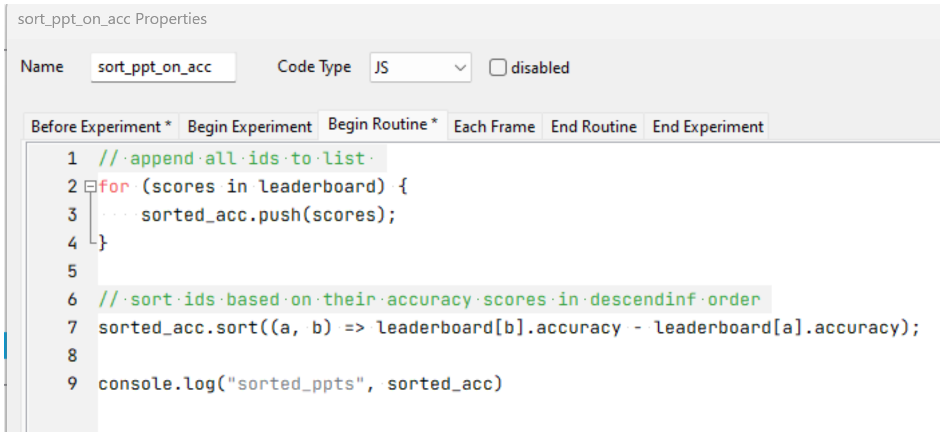
```
var all_acc;
all_acc = [];


function printAcc(obj) {
  for (var key in obj) {
    if (typeof obj[key] === 'object') {
      //console.log(obj[key][Object.keys(obj[key])[0]]);
      all_acc.push(obj[key][Object.keys(obj[key])[0]]); // append accuracies to list
      printAcc(obj[key]); // recursion for nested dictionaries
    }
  }
}

```



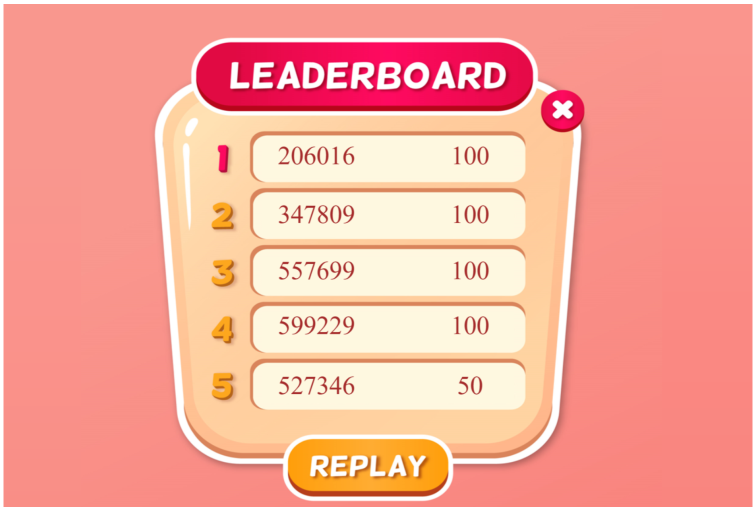
```
printAcc(leaderboard); // fetch all accuracies in leaderboard
//console.log("all_acc:", all_acc);

// sort accuracies in descending order
all_acc.sort((a, b) => b - a);
console.log("sorted_all_acc:", all_acc);
```



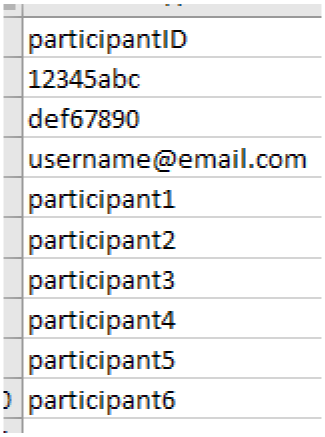
```
var sorted_acc;
sorted_acc = [];
var scores
```



```
// append all ids to list
for (scores in leaderboard) {
    sorted_acc.push(scores);
}

// sort ids based on their accuracy scores in descendinf order
sorted_acc.sort((a, b) => leaderboard[b].accuracy - leaderboard[a].accuracy);

console.log("sorted_ppts", sorted_acc)
```

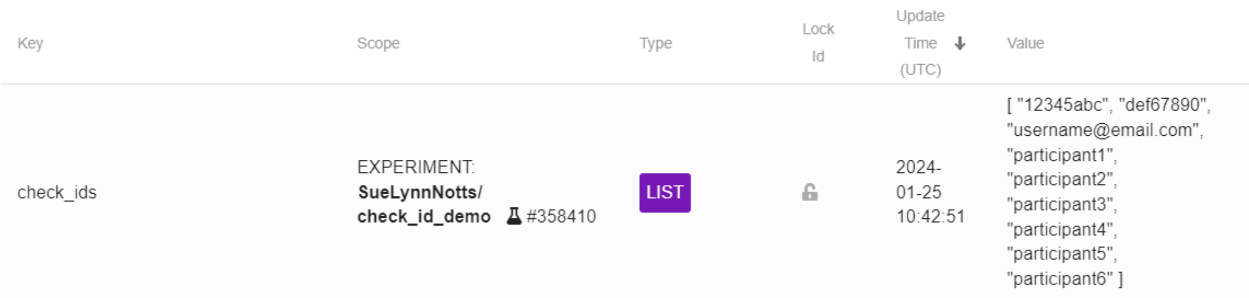


1	206016	100
2	347809	100
3	557699	100
4	599229	100
5	527346	50



participantID
12345abc
def67890
username@email.com
participant1
participant2
participant3
participant4
participant5
participant6

In the experiment files, there's a spreadsheet which automatically formats the IDs to be copied into the Shelf record (see below for an example).



Key	Scope	Type	Lock Id	Update Time ↓ (UTC)	Value
check_ids	EXPERIMENT: **SueLynnNotts/** **check_id_demo** #358410	LIST	🔓	2024-01-25 10:42:51	["12345abc", "def67890", "username@email.com", "participant1", "participant2", "participant3", "participant4", "participant5", "participant6"]

7.9 Caveats and cautions

The first caution to be aware of here is that PsychoJS was only written in 2016. It hasn't been widely battle-tested and almost certainly has some rough edges. Use it carefully and check your study does what you expect.

For an in-depth examination of the pros and cons of running studies online (including a consideration of timing issues), see The timing mega-study (Bridges et al., 2020).

7.9.1 Differences between and PsychoJS studies

The PsychoJS library looks much like its PsychoPy (Python) equivalent; it has classes like *Window* and *ImageStim* and these have the same attributes. So, from that aspect, things are relatively similar and if you already know your way around a PsychoPy script then reading PsychoJS code should be fairly intuitive.

Obviously there are some syntax changes that you'd need to understand (e.g. JavaScript requires semi-colons between lines and uses *{}* to indicate code blocks). A further issue is that the translation is usually not as simple as a line-by-line conversion

There are a few key differences that you need to understand moving from Python code to the equivalent PsychoJS script.

- Builder does not convert your Python code into JavaScript. It writes the JavaScript from scratch from the experiment logic. If you use code in a Builder Component then that code will need to be valid JavaScript code. We hope, in the future, to perform rudimentary automated conversion. Even then, that will only convert the syntax but will not be able to find equivalent function names
- Resources for your study (images, conditions files etc.) must be in the *resources* folder next to the html file that outputs. In most cases can find these and include them automatically but if your study uses code components then the resource files needed will need to be specified.

Most of the issues below affect your study if you have additional code components inserted into the study and do not affect pure Builder-based designs.

7.9.2 Timing expectations

In general timing of web-based experiments will be poorer than locally-run studies. That said, we think PsychoJS will have timing about as good as it can be! What are the specific considerations?

Variable internet connection: Surprisingly no, this isn't one to worry about! PsychoJS will make sure that the script and all the recourses for your study (image files etc) are downloaded to the computer beforehand. On a slow internet connection it may take longer for your study to start but the performance won't be limited by that while it runs. Happy days!

Visual stimuli: PsychoJS is using WebGL (high performance web rendering using advanced graphics card features) and we have confirmed that PsychoJS is able to run with frame-precise timing. That means, if you ask for a stimulus to last for, say, 6 frames then you will get exactly 100 ms of stimulus presentation.

Response timing: Again this won't be affected by your internet connection (because the keypresses are being time-stamped locally, at your computer, not by the web server). The major problem, as with any software, is that keyboards have long and variable latencies (10-30 ms typically). On a local lab-based setup you can get around this by using custom hardware (a button box) but this obviously isn't possible when your user is anywhere in the world!

7.9.3 Schedulers

A Python script runs essentially in sequence; when one line of code is called the script waits for that line to finish and then the next lines begins. JavaScript is designed to be asynchronous; all parts of your web page should load at once.

As a result, PsychoJS needed something to control the running order of the different parts of the script (e.g. the trials need to occur one after the other, waiting for the previous one to finish). To do this PsychoJS adds the concept of the *Scheduler*. For instance, you could think of the Flow in PsychoPy as being a Schedule with various items being added to it. Some of those items, such as trial loops also schedule further events (the individual trials to be run) and these can even be nested: the Flow could schedule some blocks, which could schedule a trials loop, which would schedule each individual trial.

If you export a script from one of your Builder experiments you can examine this to see how it works.

7.9.4 Functions

Some people will be delighted to see that in PsychoJS scripts output by Builder there are functions specifying what should happen at different parts of the experiment (a function to begin the Routine, a function for each frame of the Routine etc.). The essence of the PsychoJS script is that you have any number of these functions and then add them to your scheduler to control the flow of the experiment.

In fact, many experienced programmers might feel that this is the "right" thing to do and that we should change the structure of the Python scripts to match this. The key difference that makes it easy in the JavaScript, but not in the Python version, is that variables in JS are inherently *global*. When a stimulus is created during the Routine's initialization function it will still be visible to the each-frame function. In the PsychoPy Python script we would have to use an awful lot of *global* statements and users would probably have a lot of confusing problems. So, no, we aren't about to change it unless you have a good solution to that issue.

7.9.5 Supported browsers

We'd recommend running on an updated browser but pretty much any modern browser (released since roughly 2011) should be able to run these studies. It needs to support HTML5/canvas and ideally it would support WebGL (if not Canvas then will used but that is less efficient).

Specifically, these support Canvas (minimum requirement):

- Firefox 10+ (released 2012)
- Chrome 11+ (2011)
- Safari 2.0+ (2005)
- Opera 12+ (2011)
- Internet Explorer 9+ (released in 2011) but we really recommend you avoid it!

Browsers supporting WebGL (hardware-accelerated graphics in the browser):

- Firefox 15+ (2012)
- Chrome 11+ (2011)

- Safari 5.1+ (2011/12?)
- Opera 19+
- Microsoft Edge
- IE 11+ (2013)

PsychoPy and Pavlovia can also be used in conjunction with external servies, see below for guides for some common ones:

7.10 Integrating with Amazon's Mechanical Turk (MTurk)

Check out these instructions on how to integrate your Pavlovia with MTurk written by Arnon Weinberg!

Warning

Using Mechanical truk (MTurk)

Note that Mechanical Turk was not designed with behavioural science in mind, but as a way for Amazon to test computing technologies in cases where a human was needed to push the buttons. They don't particularly care about the quality of this behavioral data as a result, nor about the ethics of the participants involved.

To get better quality data, and to run studies that your local ethical review board is less likely to be concerned about, then we would suggest you use a dedicated service like Prolific Academic instead.

7.11 Recruiting with Prolific

Prolific is a dedicated service designed specifically for behavioural scientists. It aims to provide improved data quality over the likes of Mechanical Turk, with better participant selection and screening, and to provide more ethical pay levels to participants in your study.

As described in the page *Recruiting participants and daisy-chaining with other platforms*, connecting Prolific to PsychoPy is simply a matter of telling Prolific the URL for your study (including parameters to receive the Study ID etc) and then telling PsychoPy the URL to use when the participant completes the study.

Example link to provide **to Prolific** as your study URL (you will need to replace *myUserName* and *myStudyName*):

```
http://run.pavlovia.org/myUserName/myStudyName/index.html?participant={{%PROLIFIC_PID%}}&
↪study_id={{%STUDY_ID%}}&session={{%SESSION_ID%}}
```

Example link to provide **to PsychoPy** as your completion URL (you will need to change your study ID number):

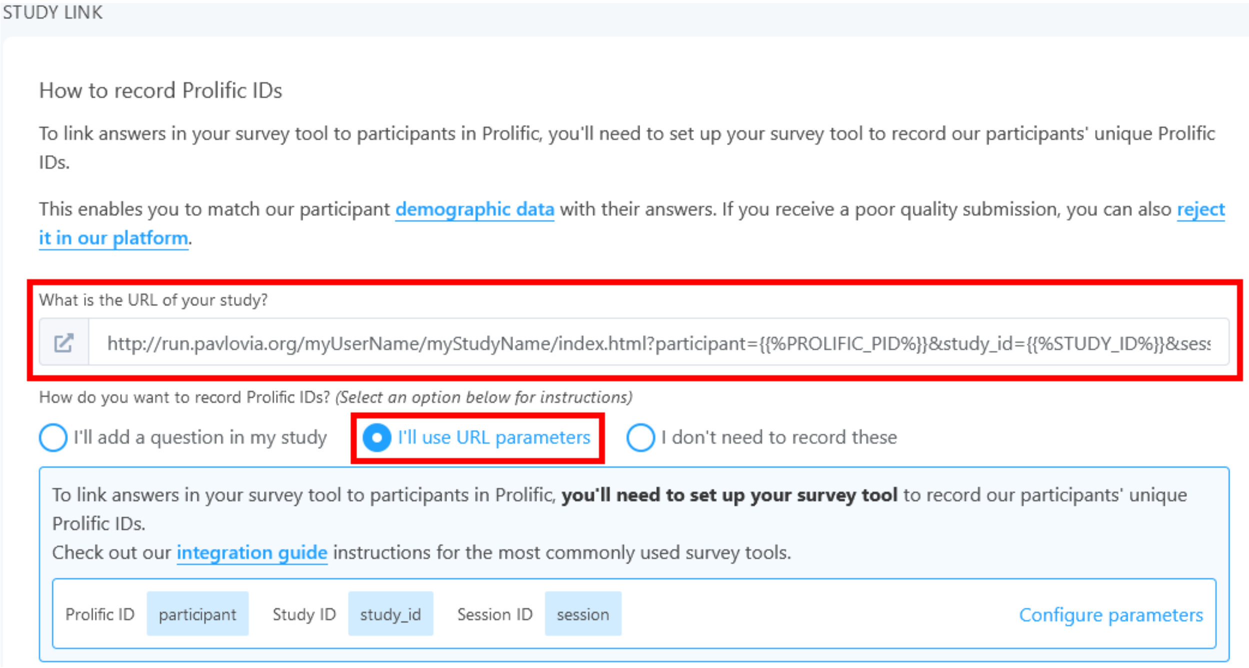
```
https://app.prolific.co/submissions/complete?cc=T8ZI42EG
```

Further details on how to find and set these links and parameters are as follows. See also Integrating Prolific with your study

7.11.1 Setting the study URL in Prolific

To recruit participants to your PsychoPy study you should see a screen as in Fig. **??** while creating/modifying your study at https://prolific.co:

Note in the above that I have set the *participant*, *session* and *study_id* for our study using a URL query string. These values will be populated by Prolific when participants are sent to the study URL. Prolific will help you to format these correctly if you tick the *Include URL Parameters?* box which will bring up the following dialog. I've changed the



STUDY LINK

How to record Prolific IDs

To link answers in your survey tool to participants in Prolific, you'll need to set up your survey tool to record our participants' unique Prolific IDs.

This enables you to match our participant demographic data with their answers. If you receive a poor quality submission, you can also reject it in our platform.

What is the URL of your study?

http://run.pavlovia.org/myUserName/myStudyName/index.html?participant={{%PROLIFIC_PID%}}&study_id={{%STUDY_ID%}}&sess

How do you want to record Prolific IDs? *(Select an option below for instructions)*

○ I'll add a question in my study ● I'll use URL parameters ○ I don't need to record these

To link answers in your survey tool to participants in Prolific, **you'll need to set up your survey tool** to record our participants' unique Prolific IDs.
Check out our integration guide instructions for the most commonly used survey tools.

Prolific ID participant Study ID study_id Session ID session Configure parameters

Fig. 7.14: Prolific settings for integration for PsychoPy

default values that PsychoPy will use to store the variables (e.g. to be *participant* and *session* which are the default names for these in PsychoPy):

In each of the boxes in Fig. **??**, you can see the name that Prolific gives to this value (e.g. *PROLIFIC_PID*) and the name that we want PsychoPy to use to store it (e.g. *participant*).

7.11.2 Setting the completion URL in PsychoPy

Do not show the completion code to your participants before they have completed your study. Displaying the completion code may result in data loss, since it encourages your participants to return to Prolific before they have completed your study. Instead copy the *Completion URL* from the main control panel above and paste that into the online tab for your PsychoPy *Experiment Settings* as in Fig. **??**:

7.12 Daisy-chaining with Qualtrics

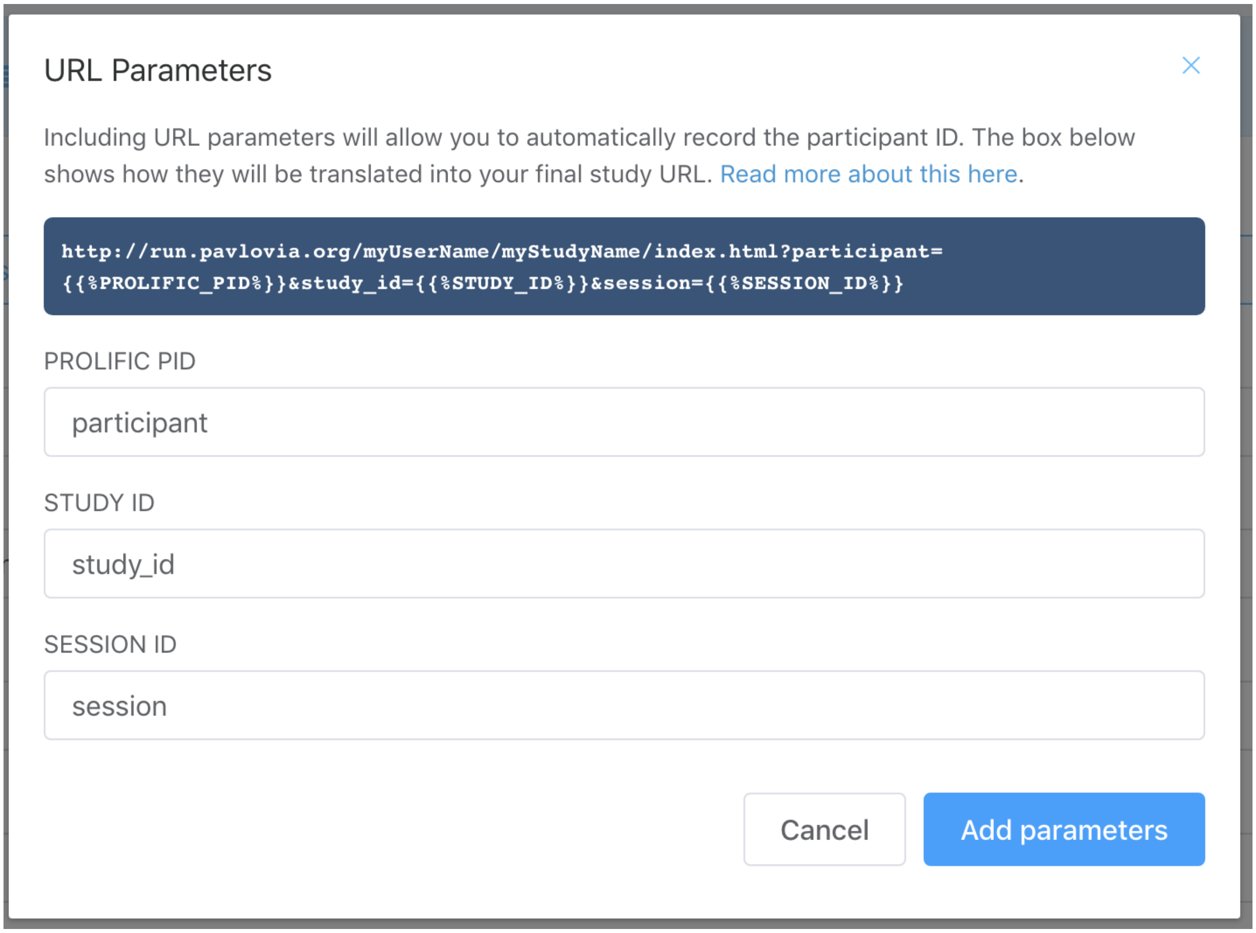
Sorry, the documentation for this hasn't yet been written.

The features are in place to do this, and the general description of how to make it work are on the page *Recruiting participants and daisy-chaining with other platforms*.

7.13 Recruiting with Sona Systems

Sona Systems is a great platform, particularly for the case where your students are all encouraged to sign up to the platform (e.g. for course credits) and the nrun studies for each other.

Sona Systems have shared an excellent guide to using Sona with PsychoPy and we would refer you to that for tips to get started.



URL Parameters

Including URL parameters will allow you to automatically record the participant ID. The box below shows how they will be translated into your final study URL. Read more about this here.

```
http://run.pavlovia.org/myUserName/myStudyName/index.html?participant=
{{%PROLIFIC_PID%}}&study_id={{%STUDY_ID%}}&session={{%SESSION_ID%}}
```

PROLIFIC PID

participant

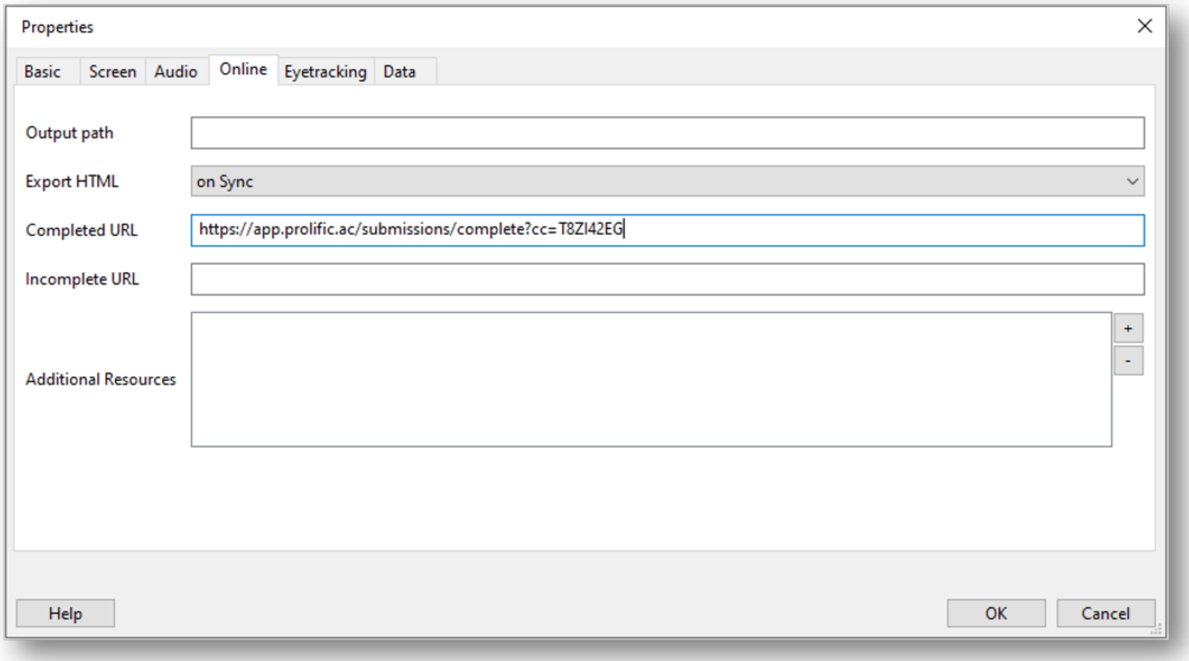
STUDY ID

study_id

SESSION ID

session

Fig. 7.15: Prolific settings (inserting parameters dialog box)



Properties

Output path

Export HTML: on Sync

Completed URL: https://app.prolific.ac/submissions/complete?cc=T8ZI42EG

Incomplete URL

Additional Resources

Fig. 7.16: The completion URL pasted into PsychoPy Experiment Settings

CHAPTER

EIGHT

RELATED LINKS

8.1 Troubleshooting Online Studies

Sometimes experiments might work perfectly locally, when created and run in the PsychoPy application, but the same experiment might not behave as you expect when you try to run them online, through pavlovia.org. While this page cannot hope to address all of the possible issues you may encounter, it should help you understand the different types of errors and help you give more detailed information if you ask for support on the PsychoPy forums.

8.1.1 Getting Started

PsychoPy Builder is your friend

1. Check whether the features you are using are supported online via our *Status of online options* page.
2. Don't try to code in PsychoJS directly.
3. Don't try to edit JavaScript files on Pavlovia directly. Make changes via Builder.
4. Each Builder (psyexp) file should be in its own dedicated local folder, which should not be in an area currently under version control (e.g.a github project, Google drive or Onedrive). This folder should only contain subfolders that pertain to the experiment.
5. Upload your files to Gitlab by synchronising using PsychoPy Builder, rather than using Git commands.
6. Code components should be set to Auto translate ("Code Type" > Auto > JS) unless you know why you need to use different code for Python and JavaScript.
7. Code components should normally be moved to the top of their respective routines. Your code is executed in order from left to right (in the flow) and from top to bottom (within each routine).
8. Experiment Settings / Online / Output path should be blank.
9. Resources (spreadsheets, images, etc.) should be in the same folder as the psyexp file or a sub-folder. Resources that are selected via code components should be added via Experiment Settings / Online / Additional Resources (see how to *Configure the online settings of your experiment*) or a Resource Manager Component. See *Resources in online studies* for more information.

Running the latest version of your experiment

When you synchronise changes to your experiment, you may need to clear your browser cache to see those changes online (using Ctrl-F5, Ctrl-Shift-R or equivalent). If this does not work use an incognito browser tab. A participant will not need to do this, so long as they have not already tried a previous version of your experiment.

Developer Tools

Use Developer Tools (Ctl-Shift-I in Windows/Chrome, Cmd-Opt-J or Cmd-Opt-I in Mac/Chrome, F12 in IE/Edge, Ctrl-Shift-J in Windows/Safari, Ctrl-Opt-J in Mac/Safari) to view errors via the browser console if you aren't getting sufficient information from PsychoPy. You can also add `print(X)` (which translates to `console.log(X);` where `X` refers to the name of your variable) to check the value of a variable `X` at a particular point.

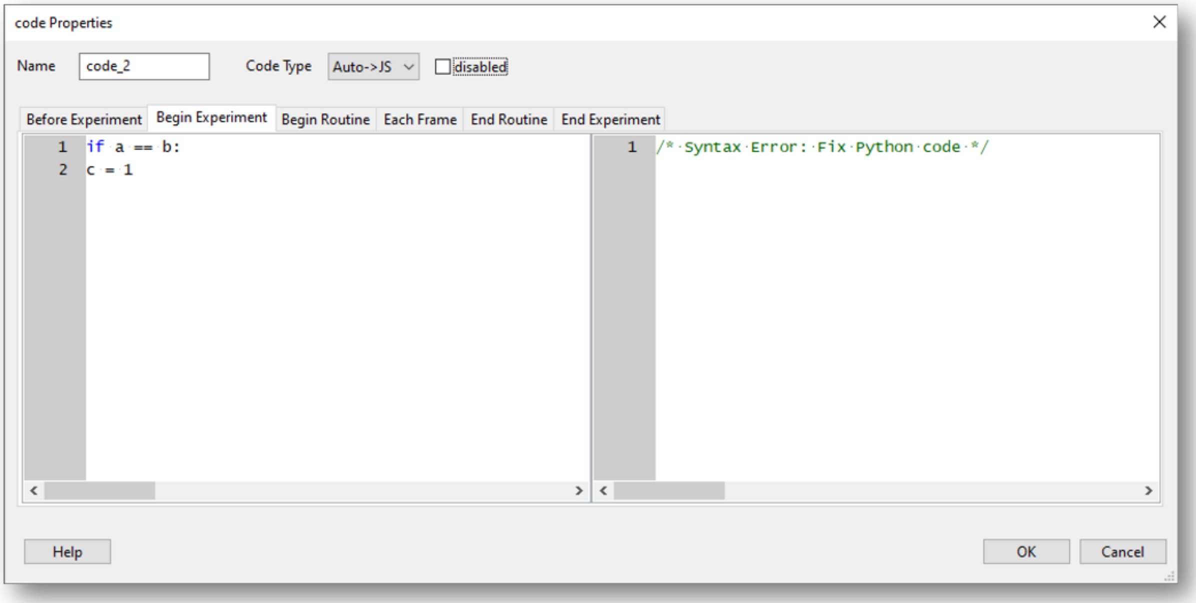
Tutorial tutorial_js_console_log

8.1.2 Types of Errors

Errors in your experiment can manifest in multiple ways. The easiest way to categorise the different types of error message is based on where they appear.

- **Builder Errors**
  - Python syntax errors
  - Builder runtime errors
  - Synchronisation errors
- **Browser Errors**
  - Launch errors
  - Resource errors
  - Semantic errors
- Unexpected Behaviour

Python Syntax Errors (seen in Auto-translate code components)



Name: code_2

Code Type: Auto->JS

☐ disabled

```
if a == b:
c = 1
```

```
/* Syntax Error: Fix Python code */
```

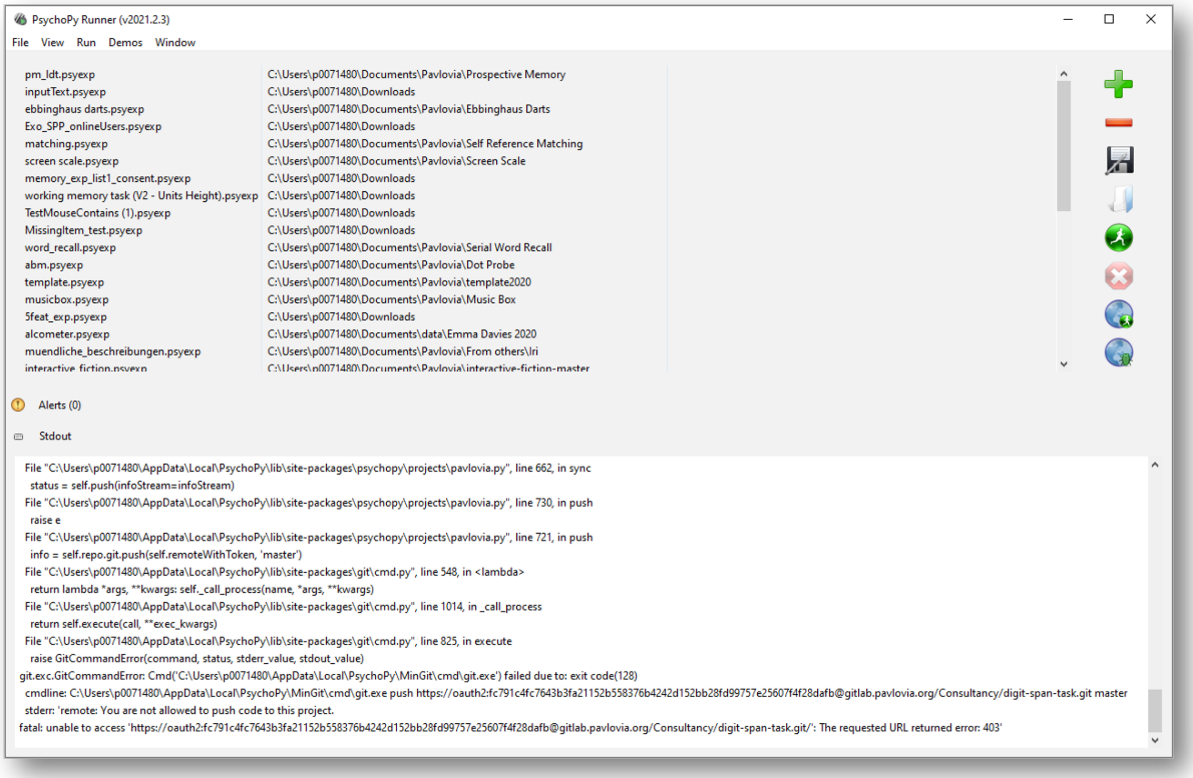
Fig. 8.1: A code component used in PsychoPy Builder. In this example, "Code Type" is set to "Auto > JS" meaning python code (on the left) will transpile to JavaScript (on the right). In this example there is a python coding error, which means the transpilation cannot occur.

One of the advantages of using auto translate code components is that the transpiler is continuously checking your code in order to translate it to JavaScript. If you have a syntax error in your Python code, the JavaScript translation will be `/* Syntax Error:  Fix Python code */`. If you get this type of error then your Python code probably won't run locally, and no translated code will be added to the JavaScript version.

Note

"Old style" string formatting (using a % operator) works in Python but gives a syntax error in JavaScript but string interpolation (f-strings) is fine.

Synchronisation Errors (seen in the PsychoPy Runner Stdout)



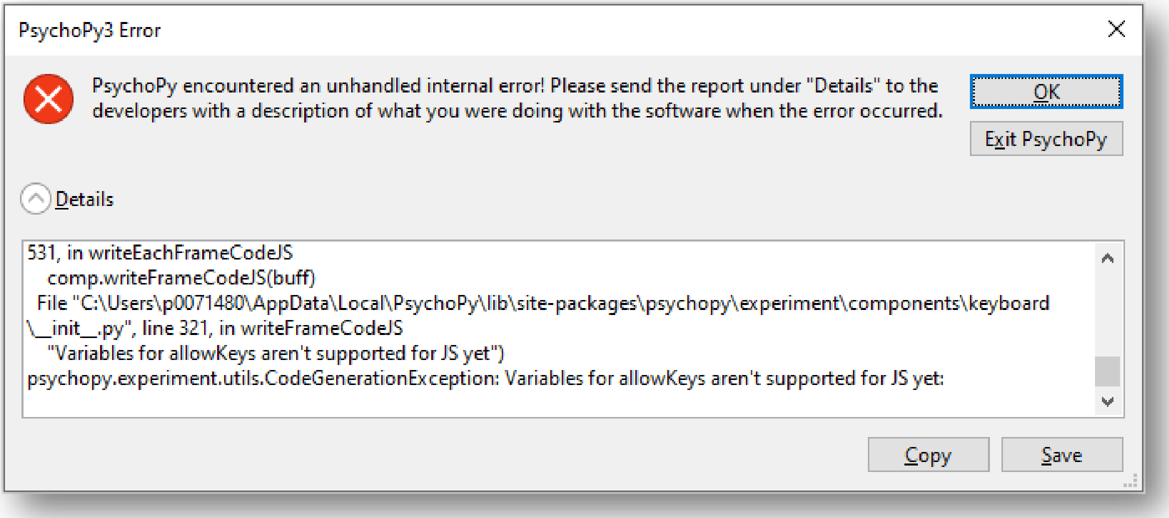
pm_ldt.psyexp	C:\Users\p0071480\Documents\Pavlovia\Prospective Memory
inputText.psyexp	C:\Users\p0071480\Downloads
ebbinghaus darts.psyexp	C:\Users\p0071480\Documents\Pavlovia\Ebbinghaus Darts
Exo_SPP_onlineUsers.psyexp	C:\Users\p0071480\Downloads
matching.psyexp	C:\Users\p0071480\Documents\Pavlovia\Self Reference Matching
screen scale.psyexp	C:\Users\p0071480\Documents\Pavlovia\Screen Scale
memory_exp_list1_consent.psyexp	C:\Users\p0071480\Downloads
working memory task (V2 - Units Height).psyexp	C:\Users\p0071480\Downloads
TestMouseContains (1).psyexp	C:\Users\p0071480\Downloads
MissingItem_test.psyexp	C:\Users\p0071480\Downloads
word_recall.psyexp	C:\Users\p0071480\Documents\Pavlovia\Serial Word Recall
abm.psyexp	C:\Users\p0071480\Documents\Pavlovia\Dot Probe
template.psyexp	C:\Users\p0071480\Documents\Pavlovia\template2020
musicbox.psyexp	C:\Users\p0071480\Documents\Pavlovia\Music Box
5feat_exp.psyexp	C:\Users\p0071480\Downloads
alcometer.psyexp	C:\Users\p0071480\Documents\data\Emma Davies 2020
muendliche_beschreibungen.psyexp	C:\Users\p0071480\Documents\Pavlovia\From others\Iri
interactive_fiction.psyexp	C:\Users\p0071480\Documents\Pavlovia\interactive-fiction-master

```
 File "C:\Users\p0071480\AppData\Local\PsychoPy\lib\site-packages\psychopy\projects\pavlovia.py", line 662, in sync
  status = self.push(infoStream=infoStream)
 File "C:\Users\p0071480\AppData\Local\PsychoPy\lib\site-packages\psychopy\projects\pavlovia.py", line 730, in push
  raise e
 File "C:\Users\p0071480\AppData\Local\PsychoPy\lib\site-packages\psychopy\projects\pavlovia.py", line 721, in push
  info = self.repo.git.push(self.remoteWithToken, 'master')
 File "C:\Users\p0071480\AppData\Local\PsychoPy\lib\site-packages\git\cmd.py", line 548, in <lambda>
  return lambda *args, **kwargs: self._call_process(name, *args, **kwargs)
 File "C:\Users\p0071480\AppData\Local\PsychoPy\lib\site-packages\git\cmd.py", line 1014, in _call_process
  return self.execute(call, **exec_kwargs)
 File "C:\Users\p0071480\AppData\Local\PsychoPy\lib\site-packages\git\cmd.py", line 825, in execute
  raise GitCommandError(command, status, stderr_value, stdout_value)
git.exc.GitCommandError: Cmd('C:\Users\p0071480\AppData\Local\PsychoPy\MinGit\cmd\git.exe') failed due to: exit code(128)
 cmdline: C:\Users\p0071480\AppData\Local\PsychoPy\MinGit\cmd\git.exe push https://oauth2:fc791c4fc7643b3fa21152b558376b4242d152bb28fd99757e25607f4f28dafb@gitlab.pavlovia.org/Consultancy/digit-span-task.git master
 stderr: 'remote: You are not allowed to push code to this project.
fatal: unable to access 'https://oauth2:fc791c4fc7643b3fa21152b558376b4242d152bb28fd99757e25607f4f28dafb@gitlab.pavlovia.org/Consultancy/digit-span-task.git/': The requested URL returned error: 403'
```

Fig. 8.2: An example "synchronisation error" as shown in PsychoPy Runner. In this example the experimenter is attempting to synchronise an experiment while logged into a different Pavlovia account in PsychoPy Builder.

Errors that occur here during synchronisation are often related to the connection to the gitlab repository on Pavlovia. The Stdout will contain a number of messages. Focus on errors (not warnings) which appear near the top or bottom of the output that has just been generated. If you need to recreate a new project then you may need to delete the local hidden .git folder to sever the old connection. If the error message is not related to the git connection, this flow chart might be helpful.

Synchronisation Errors (seen in a pop-up when synchronising)



PsychoPy3 Error

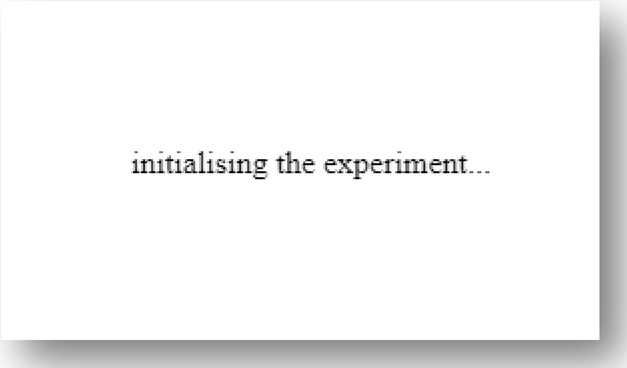
PsychoPy encountered an unhandled internal error! Please send the report under "Details" to the developers with a description of what you were doing with the software when the error occurred.

```
531, in writeEachFrameCodeJS
    comp.writeFrameCodeJS(buff)
  File "C:\Users\p0071480\AppData\Local\PsychoPy\lib\site-packages\psychopy\experiment\components\keyboard
\__init__.py", line 321, in writeFrameCodeJS
    "Variables for allowKeys aren't supported for JS yet")
psychopy.experiment.utils.CodeGenerationException: Variables for allowKeys aren't supported for JS yet:
```

Fig. 8.3: An example "synchronisation error" as shown in PsychoPy Builder. In this example the experimenter has set the *Allowed keys* of a keyboard component as a variable, which is not yet supported in PsychoJS.

Errors occur here when PsychoPy is unable to create a JavaScript file from your Builder file. They are usually related to your custom code components, but can be caused by unexpected parameters in your other components. These errors will prevent your JavaScript files from being created and therefore stop you making any changes to previous versions you may have successfully synchronised. See *Launch your study on Pavlovia.org* for more information.

Launch Errors (stuck on "initialising the experiment")



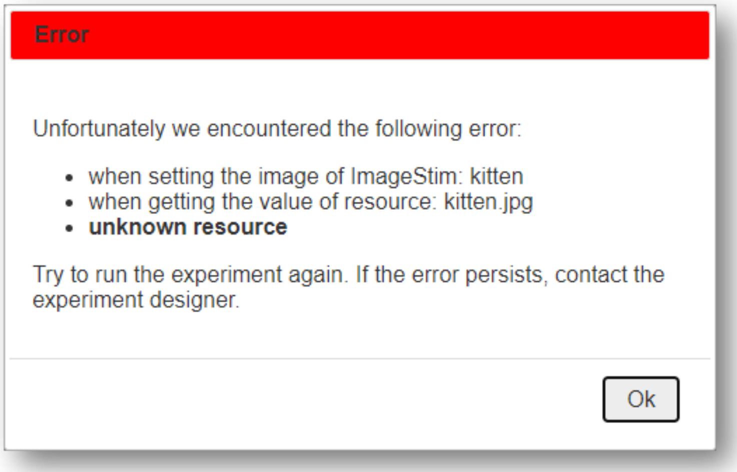
initialising the experiment...

Fig. 8.4: The "initialising the experiment" message shown when launching and experiment in pavlovia.org.

If, when you try to launch your experiment, it is stuck on "initialising the experiment" then Pavlovia has encountered a syntax error in your JavaScript file that wasn't caught by the checks during synchronisation. The most common cause for this error is that you are trying to import a Python library, such as random or numpy, which don't exist in JavaScript. Use Developer Tools to look for more information.

Tutorial tutorial_js_syntax_error experiment

Resource Errors



Error

Unfortunately we encountered the following error:

- when setting the image of ImageStim: kitten
- when getting the value of resource: kitten.jpg
- **unknown resource**

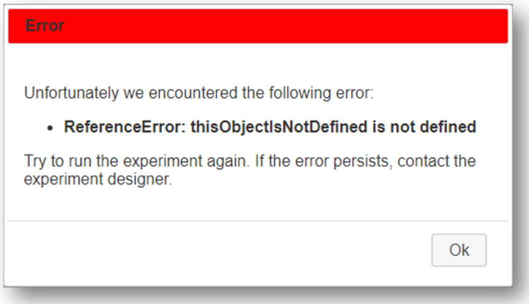
Try to run the experiment again. If the error persists, contact the experiment designer.

Fig. 8.5: An example "unknown resource" error message as shown in pavlovia.org. In this example the experiment cannot locate an image.

To understand resource errors it is really important to understand *Resources in online studies* - and we recommend you check out this information to understand how to properly load resources in your experiment. This occurs when an additional resource such as a spreadsheet or image file hasn't been made available to the experiment. This can either occur because the file couldn't be found when requested, or because there was an attempt to use the file without downloading it first. These errors are often referred to as network errors, but this does not mean that they are caused by general connectivity issues.

Tutorial tutorial_js_network_error experiment

Semantic Errors



Error

Unfortunately we encountered the following error:

- **ReferenceError: thisObjectIsNotDefined is not defined**

Try to run the experiment again. If the error persists, contact the experiment designer.

Fig. 8.6: An example "semantic error" where something is not defined (Typically a variable name).

These errors occur when a variable has not been defined or declared in the JavaScript version of your experiment. There are typically two reasons for this error.

1. You may have used a python library of PsychoPy object that does not exist, and is therefore not defined, in JavaScript. For example if you used `np.average([1, 2, 3])` in a code component, you would get the error message "np is not defined" (to avoid this specific error use `average([1, 2, 3])` - dropping the reference to numpy).

2. To define a variable in simply add something like `X = 1` in the Begin Experiment or Begin Routine tab of an auto translate code component.

Most semantic errors can be solved by searching for the text of the error message on the discourse forum. You can also use the Developer Tools to help identify which command is causing the error.

Tutorial tutorial_js_semantic_error experiment

Unexpected Behaviour

Sometimes your experiment will run without any error messages but something will be missing or wrong. This can occur if:

1. you try to use a component that doesn't yet work online
2. you have code components set to Python only.
3. you use a python function that might work subtly differently in python and JavaScript (for example `pop(0)` will remove the first thing from a list in python, but the last thing from a list in Javascript.

If you're using code components, it's useful to think about the positions of your code components and how they are executed relative to your other components. Since **Begin Routine** code tabs are executed at the same time as **set every repeat** component parameters in top to bottom order. Did you set the parameter before or after it was used? If you something to change during a routine, it needs to be in an **Each Frame** code tab or a **set every frame** component parameter.

8.1.3 Getting Help

Once you have identified the error message or behaviour you are trying to fix, search the PsychoPy forum for other threads discussing the same issue, using keywords from your error message or issue. Some threads are marked with a tick before the name to indicate that they contain a solution. You may also find the solution in Wakefield Morys-Carter's PsychoPy to JS crib sheet.

If your issue is solved thanks to a solution you found in a thread, we recommend adding a +1 or like reaction to the post that helped you (remember many of those who support our forum are volunteers! so it's useful to show appreciation and indicate to others seeking help which answer was used by others). If a post you create is solved by a suggestion please mark that response with as the "solution".

If you are unable to solve the problem with existing solutions already posted on the forum then either add a post to a thread which refers to the same issue and doesn't have a solution or start a new thread and include a link to the solution you tried or the most similar thread you have come across in your search.

Creating a New Topic on the forum

Select an appropriate *category*:

- **Online experiments** if you are planning to run your experiment online.
- **Builder** if you are using PsychoPy Builder for a local experiment.
- **Coding** if you are using PsychoPy Coder for a local experiment.
- **Other** if you are having issues that aren't related to a particular experiment.

Give your new topic a useful *title* such as the text of the error message and/or a short clear description of what is going wrong.

Include the *version of PsychoPy* you are using and a usable link to your experiment.

If you have a Browser error near the beginning of your experiment, it is helpful to allow people to try it for themselves. Since Pilot tokens expire, the easiest way to allow others to view your experiment is to set it to RUNNING and allocate it a small number of credits. Add a final routine with a text component that doesn't end (possibly unless you press a key such as = which isn't typically used). You should also set your experiment not to save incomplete results using the Dashboard entry for your project so no credits are consumed during testing.

Since most of the JavaScript code is generated automatically, either from Builder components or by Auto translations in code components it is most useful to show screen shots from Builder (the flow and the relevant routine, plus the contents of the component with the issue). If the issue is with an Auto code component, then you should paste the contents of the Python side as preformatted text, as well as showing the screenshot. Only paste JavaScript from Both and JS only code components to clarify that these have been manually edited.

8.1.4 What next?

We will try to give as much support as possible for free in the public space. However if you are still stuck we can offer paid consultancy options to help debug. You can contact our team directly at consultancy@opensciencetools.org. Consultancy is part of our sustainable model for Open Source Tools and allows us to keep creating free and accessible tools (see *Overview* and read more on Open Science Tools). Our Science team will be happy to help via one-to-one technical support hours or larger consultancy projects.

contains a collection of experiments that you can use as a starting point for your own experiment. Below we explain how you can search for experiments in this collection and contribute your own experiment.

8.2 Searching for experiments on Pavlovia

You can search for experiments via the website and from within the PsychoPy Builder.

8.2.1 Via the website

From the home page, you can explore your own existing projects, or other users' public projects. To find a project, go to Pavlovia's Explore page (see Fig. **??**).

When exploring studies online, you are presented with a series of thumbnail images for all of the projects on . See Fig. **??**.

From the "Explore" page, you can filter projects by setting the filter buttons to a) Public or Private, b) Active or Inactive, and c) sort by number of forks, name, date and number of stars. The default sorting method is Stars. You can also search for projects using the search tool using keywords describing your area of interest, e.g., Stroop, or attention.

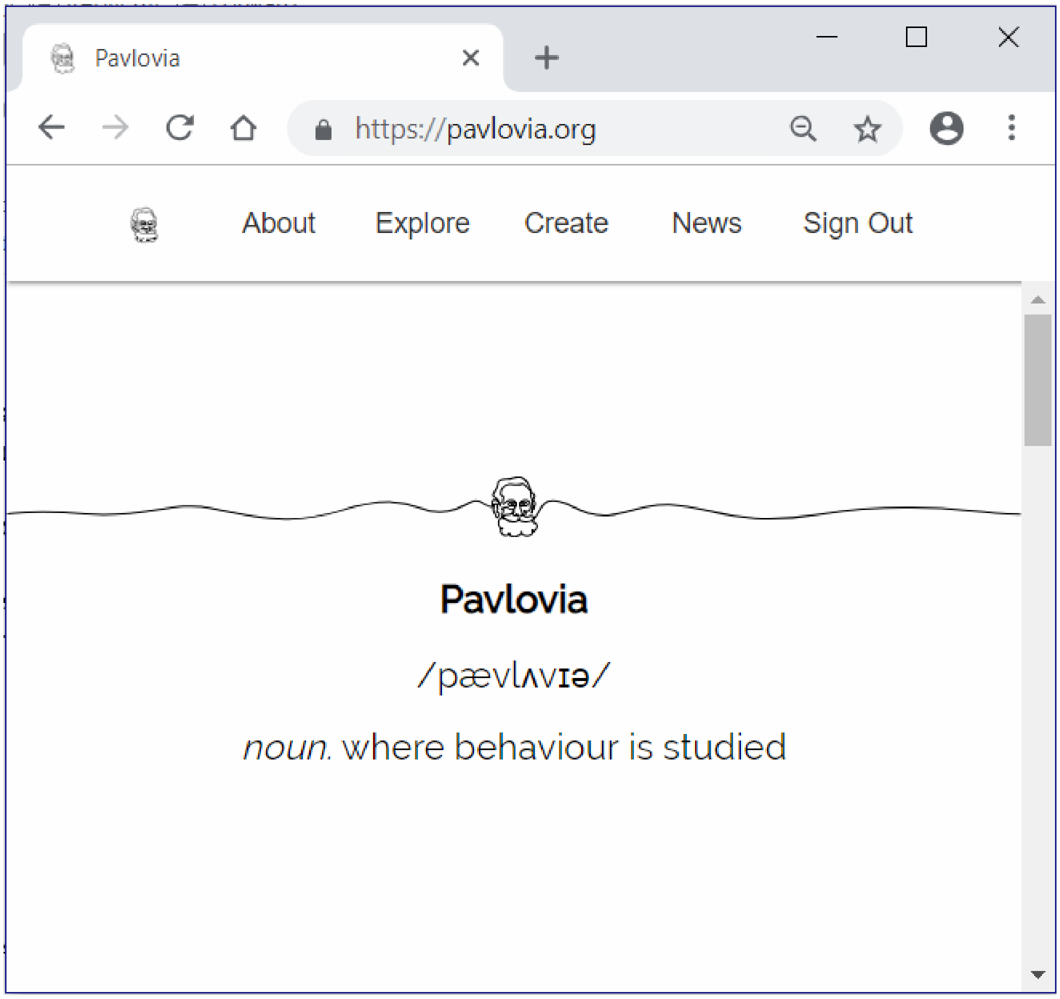
8.2.2 Via the PsychoPy Builder

If you wish to search for your own existing projects on , or other users' public projects, you can also do this via the Builder interface. To search for a project, click button (3) on the Builder Frame in Fig. **??**.

Following this, a search dialog will appear, see Fig. **??**. The search dialog presents several options that allow users to search, fork and synchronize projects.

To search for a project (see Fig. **??**, Box A), type in search terms in the text box and click the "Search" button to find related projects on . Use the search filters (e.g., "My group", "Public" etc) above the text box to filter the search output. The output of your search will be listed in the search panel below the search button, where you can select your project of interest.

To fork and sync a project is to take your own copy of a project from (*fork*) and copy a version to your local desktop or laptop computer (*sync*). To fork a project, select the local folder to download the project using the "Browse" button, and then click "Sync" when you are ready - (see Fig. **??**, Box B). You should now have a local copy of the project from ready to run in PsychoPy!

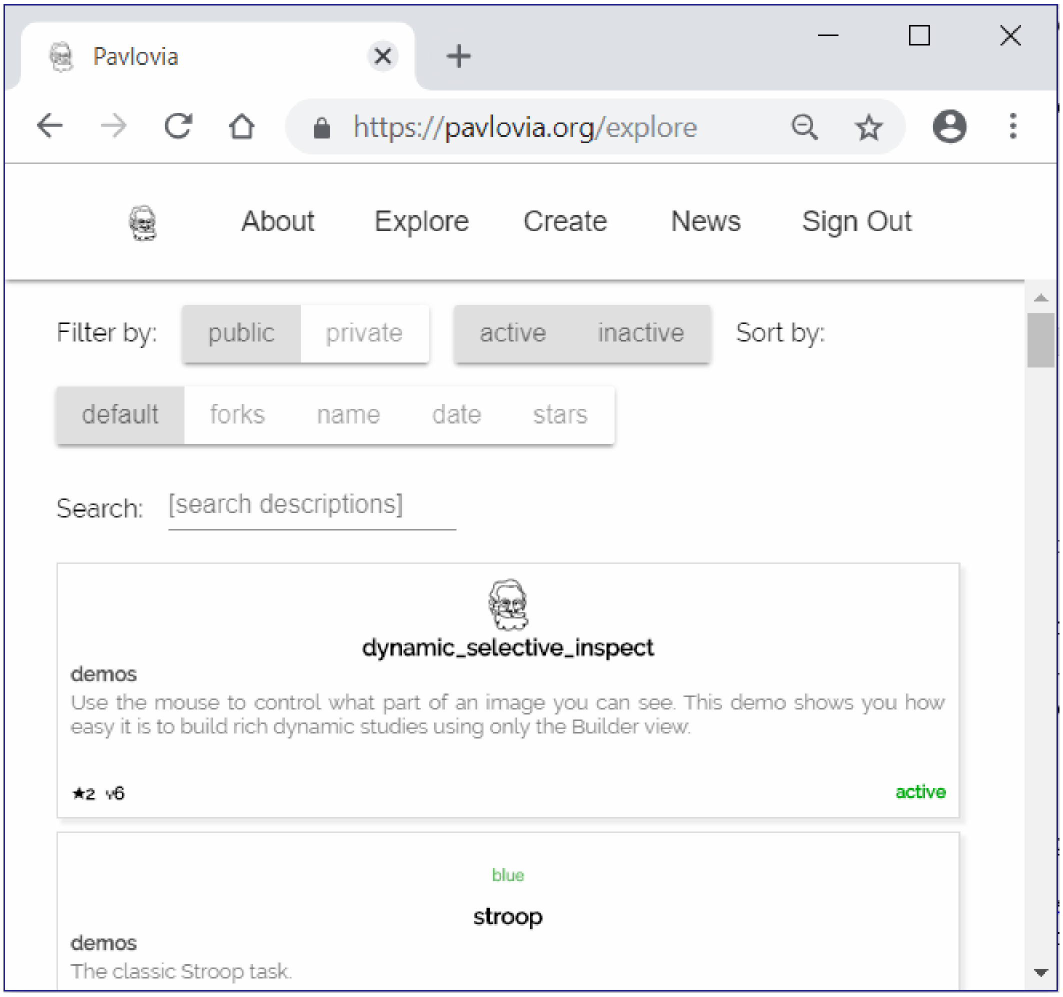


Pavlovia

/pævlʌvɪə/

noun. where behaviour is studied

Fig. 8.7: The home page



dynamic_selective_inspect

demos

Use the mouse to control what part of an image you can see. This demo shows you how easy it is to build rich dynamic studies using only the Builder view.

stroop

demos

The classic Stroop task.

Fig. 8.8: Exploring projects on

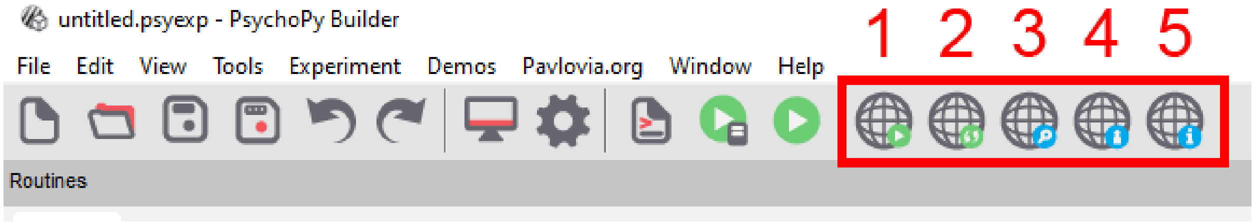
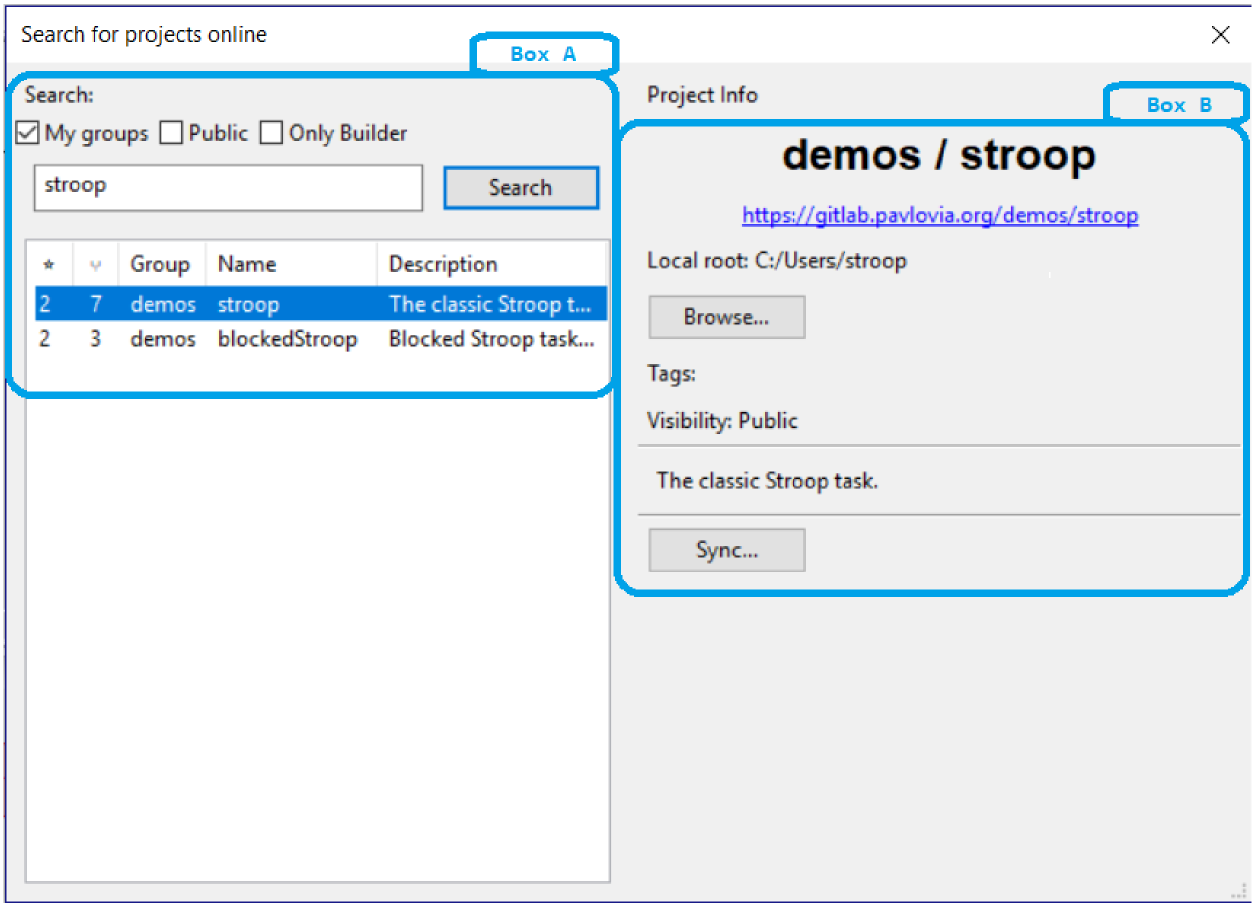


Fig. 8.9: Buttons for running an online study from the PsychoPy Builder.



Search:

☑ My groups ☐ Public ☐ Only Builder

stroop

★	⑂	Group	Name	Description
2	7	demos	stroop	The classic Stroop t...
2	3	demos	blockedStroop	Blocked Stroop task...

Project Info

demos / stroop

https://gitlab.pavlovia.org/demos/stroop

Local root: C:/Users/stroop

Tags:

Visibility: Public

The classic Stroop task.

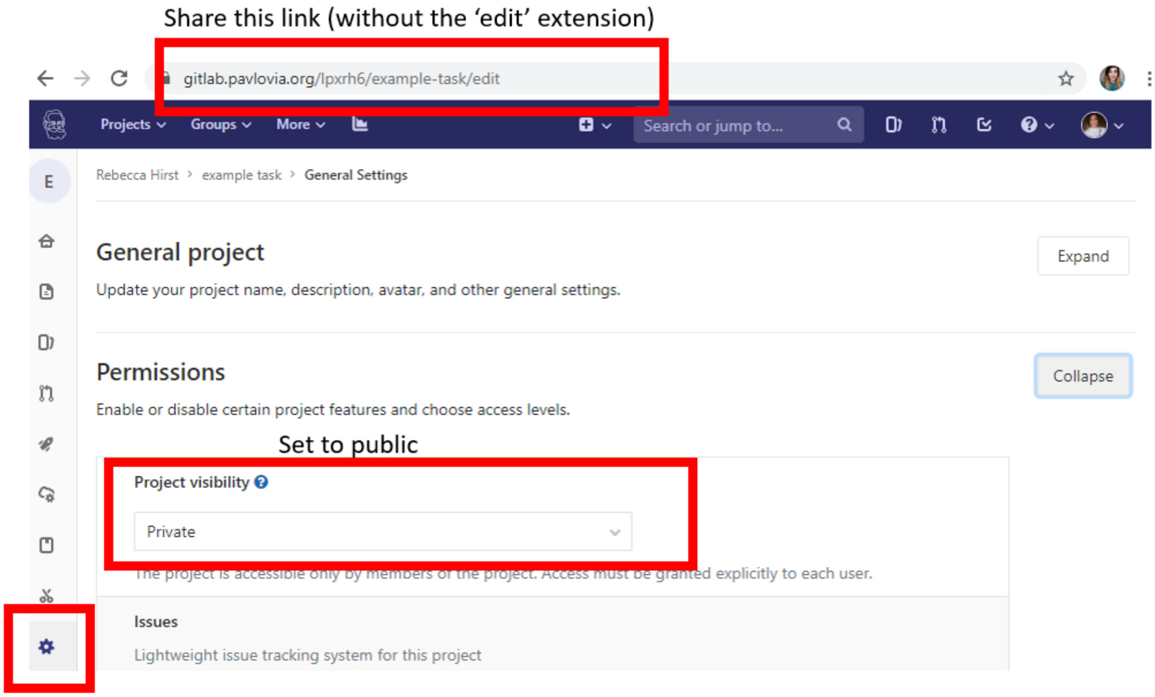
Fig. 8.10: The search dialog in Builder

8.3 Contributing an experiment to Pavlovia

If you contribute an experiment to Pavlovia, other researchers can access it. Besides contributing to open science, this can be handy if you've got an issue with your experiment and would like other researchers to take a look.

8.3.1 Making an experiment public

A public experiment is visible to anyone to clone and fork. To make your experiment public navigate to your experiments' GitLab page, then select > View code <> > Settings > Permissions (set to public). See Fig. **??**.



Share this link (without the 'edit' extension)

gitlab.pavlovia.org/lpxrh6/example-task/edit

Rebecca Hirst > example task > General Settings

General project

Update your project name, description, avatar, and other general settings.

Permissions

Enable or disable certain project features and choose access levels.

Set to public

Project visibility

Private

The project is accessible only by members of the project. Access must be granted explicitly to each user.

Issues

Lightweight issue tracking system for this project

Fig. 8.11: Setting a GitLab project to public access

8.3.2 Adding a team member

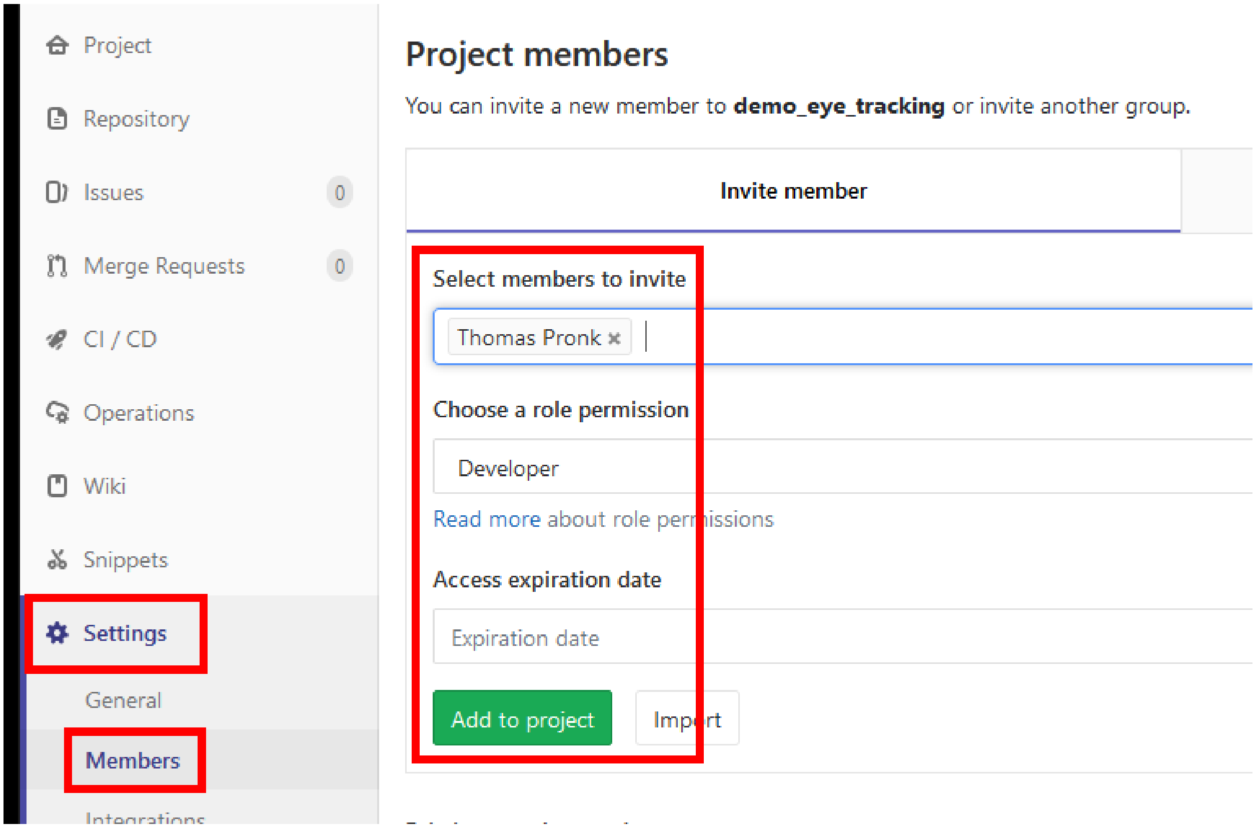
If you'd like to share your experiment only with specific researchers, navigate to your experiment, then select > View code <> > settings > Members. At this page: select a member, give them a role (to be able to fork your experiment, they should at least be Developer), optionally an access expiration date, and then add them. See Fig. **??**.

8.4 Recruiting participants and daisy-chaining with other platforms

We have now moved our updated documentation on daisy-chaining participants to Pavlovia. Click here to find out more.

8.5 Counterbalancing online

If you are manually recruiting your participants (i.e. sending our your experiment URL to a unique population or group) the methods described in *Blocks of trials and counterbalancing* will also work online, and can be used in the same way. However, if you have your experiment URL advertised on a recruitment website, it could be that 10s or



Project members

You can invite a new member to **demo_eye_tracking** or invite another group.

Select members to invite: Thomas Pronk

Choose a role permission: Developer

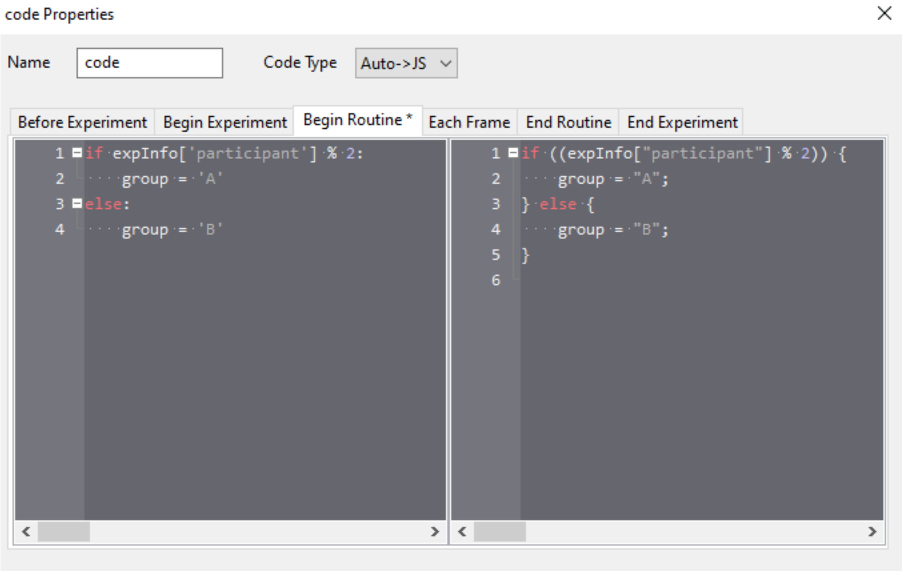
Access expiration date: Expiration date

Fig. 8.12: Adding a user to a GitLab project

hundreds of participants click your link. Manually assigning participants and keeping track of participant groupings in these situations is going to be difficult. So, what do we do?

At the moment and don't have an internal method for keeping track of how many participants have already completed your task (and this is needed for counterbalancing). However, some core contributors have developed some excellent tools to help out, in particular this tool developed by Wakefield Morys Carter that generates sequential participant IDs for your task. Although not a counterbalance tool per se, we can use this to assign out participants to specific groups.

Add a code component to the beginning of your task that looks something like this:



Name: code

Code Type: Auto->JS

```
if expInfo['participant'] % 2:
    group = 'A'
else:
    group = 'B'
```

```
if ((expInfo["participant"] % 2)) {
    group = "A";
} else {
    group = "B";
}
```

Here we are checking if your participant ID is divisible by 2 (i.e. odd of even) and creating the variable 'group' using that. We would then use the same methods outlined previously in *Blocks of trials and counterbalancing* except this time we replace any instance of: `` `expInfo['group']` `` with `` `group` ``

So the conditions files used is selected based on the participant ID!

8.6 How does it work?

The first stage of this is that there is now a JavaScript library, PsychoJS, that mirrors the PsychoPy Python library classes and functions.

PsychoPy Builder is effectively just writing a script for you based on the visual representation of your study so the new feature is for it simply to write a html/JavaScript/PsychoJS page instead.

Modern browsers are remarkably powerful. Most browsers released since 2011 have allowed HTML5 which supports more flexible rendering of web pages (images and text can be positioned precisely enough to run "proper" behavioural experiments). Since 2013 most have supported WebGL. That allows graphics to be rendered really quickly using "hardware acceleration" on your graphics card. The result is rich pages that can be updated very rapidly and can be forced to sync to screen refresh, which is critical for stimulus timing.

All this means we can do great things with online experiments that actually have good temporal precision!

The way it works is that you have a web page containing JavaScript (generated by PsychoPy Builder). You upload that to a web server. The participant of your study uses their web browser to visit the page you've created with a standard URL you send them.

Now, JavaScript executes on their computer (as opposed to scripts like PHP that operate on the server and aren't directly visible to the viewer/browser). In this case the PsychoJS script will present a dialog box at the start of the study to get the participant ID and any other basic information you need. While that dialog box is presented the script will be downloading all your stimuli and files to the local computer and storing them in memory. When all the necessary files are downloaded the participant can press "OK" and the experiment will start.

The experiment supports all the standard timing aspects of any PsychoPy Builder experiment; you can specify your stimuli in terms of time presented or number of screen refreshes etc (and the actual refresh rate of your participant's computer will be stored in your data file). When it's finished it saves the data into a comma-separated-value (CSV) file in the "data" folder on the web server. This looks very much like the standard CSV outputs of your same PsychoPy experiment run locally.

Not all components are currently supported. Keep an eye on the *Status of online options* page to see what objects you can use already.

8.6.1 How does this compare with jsPsych?

In jsPsych you use one of the pre-programmed "types" of trial (like single stimulus or 2-alternative-forced-choice) and you have rather little flexibility over how that gets conducted. If you wanted to alter the positioning of the stimuli, for instance, in a 2-alternative-force-choice task or you wanted a stimulus to change in time (appear gradually or move location) then you would need to write a new trial "type" using raw javascript.

PsychoPy, by comparison, is designed to give you total flexibility. You decide what constitutes a "trial" and how things should operate in time. We think that control is very important to creating a wide range of studies.

8.7 Manual coding of PsychoJS studies

Note that PsychoJS is very much under development and all parts of the API are subject to change

Some people may want to write a JS script from scratch or convert their PsychoPy Python script into **|PsychoJS|**. However, supporting this approach is beyond the scope of our documentation and our forum.

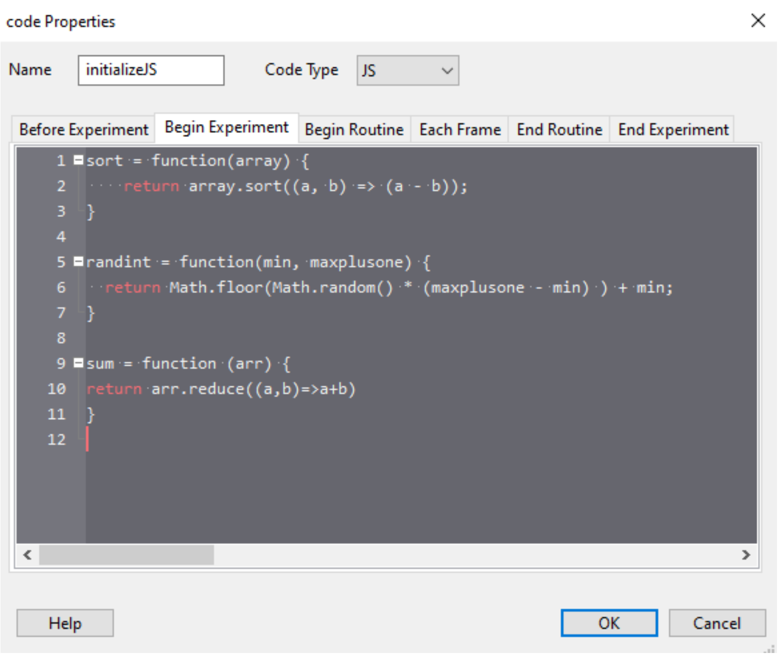
8.7.1 Working with JS Code Components

Code components can automatically convert Python to JavaScript. However, this doesn't always work. Below are some pointers to help you out:

- For common JS functions, see the PsychoPy to JS crib sheet by Wakefield Morys-Carter
- For finding out how to manipulate PsychoJS components via code, see the PsychoJS API. The tutorial_js_expose_psychojs experiment shows how to expose PsychoJS objects to the web browser, so that you can access them via the browser console, and try things out in order to see what works (or not).
- If you're looking for a JS equivalent of a Python function, try searching 'JS equivalent/version of function X' on stack overflow or Google
- Still stuck? Try asking for help on the forum. For giving researchers access to the repository of your experiment, see *Contributing an experiment to Pavlovia*

8.7.2 Adding JS functions

If you have a function you want to use, and you find the equivalent on the crib sheet or stack overflow, add an 'initialization' code component to the start of your experiment. Set code type to be 'JS' and copy and paste the function(s) you want there in the 'Begin experiment' tab. These functions will then be available to be called throughout the rest of the task.



Name: initializeJS

Code Type: JS

```
sort = function(array) {
    return array.sort((a, b) => (a - b));
}

randint = function(min, maxplusone) {
  return Math.floor(Math.random() * (maxplusone - min) ) + min;
}

sum = function (arr) {
return arr.reduce((a,b)=>a+b)
}
```

8.7.3 Don’t change the generated JS file

When you export an experiment to HTML from the PsychoPy builder, it generates a JS file. We recommend *not* to edit this JS file, for the reasons below:

- Changes you make in your .js file will not be reflected back in your builder file; it is a one way street.
- It becomes more difficult to sync your experiment with from the builder
- Researchers that would like to replicate your experiment but aren’t very JavaScript-savvy might be better off using the PsychoPy Builder

8.8 Usage statistics

The first generation of PsychoJS was realized by a Wellcome Trust grant, awarded in January 2018. to make online studies possible from . This is what we call PsychoPy3 - the 3rd major phase of PsychoPy’s development.

CHAPTER

NINE

COMMUNICATING WITH EXTERNAL HARDWARE USING PSYCHOPY

PsychoPy is able to communicate with a range of external hardware, like EEG recording devices and eye trackers.

This page provides step-by-step instructions on how to communicate with some of the more commonly used hardware. The page is being updated regularly so if you don't see your device listed here please do post in the forum as we keep an eye on commonly-faced issues (and solutions!) there.

9.1 Communicating with EEG

Before getting started with an EEG study in PsychoPy, we **highly** recommend reading relevant information on how to measure and understand *Timing Issues and synchronisation*. Although these guides will talk you through how to communicate with EEG hardware, they can really be used to communicate with any device that is connected via the same method:

9.1.1 Communicating via a Parallel Port

Step one: Set up your Parallel Port component in Builder

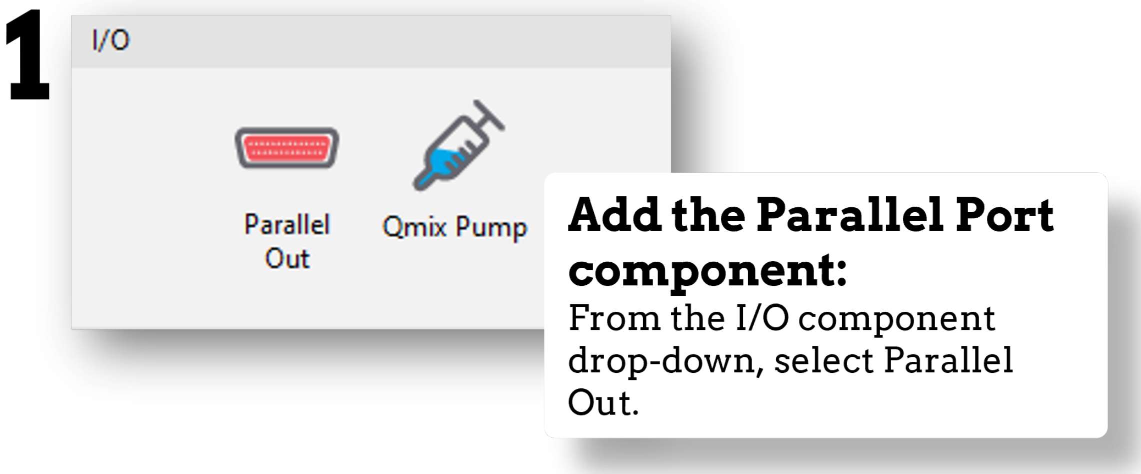
PsychoPy has a Parallel Port component in Builder view. This can be found in the I/O component drop down. This component supports both traditional parallel ports and USB devices.

If you'd like to use a *Parallel Port* to **record** responses (for example from a button box) please read this excellent thread from our Discourse Forum user jtseng.

- Add your Parallel Port component to your routine in the same way that you would with any other component:
- Now, imagine we want our trigger sent to indicate stimulus onset. We *could* do this by simply setting the onset time of the trigger to match that of our stimulus. But this is not the **most** precise way to do this. Also, this doesn't help us if we want to send our trigger to indicate something with variable timing, such as when a response is made.
- For maximum precision, we'll set the trigger to be sent when the status of our stimulus is set to *started*:
- Now we set that condition by inserting the following code:

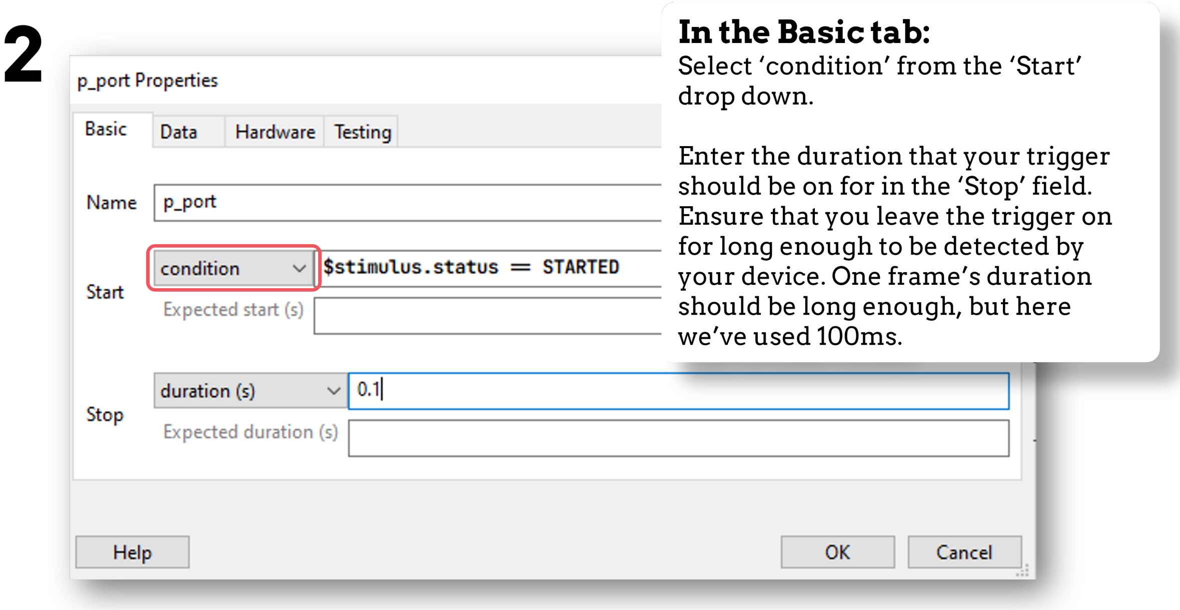
```
stimulus.status == STARTED #Change 'stimulus' here to match the name of your own component
```

- Now, in the *Data* tab, we set the data we want the trigger to actually send:
- So our component is added and we've set it up the way we want. We now need to make sure that the trigger is going to be sent to the right place!
- To do this, we're firstly going to check our port address.



Add the Parallel Port component:
From the I/O component drop-down, select Parallel Out.

Fig. 9.1: Select the *Parallel Port* component from the *I/O* or *EEG* component drop-down menus.



Name: p_port

Start: condition — $stimulus.status == STARTED

Expected start (s):

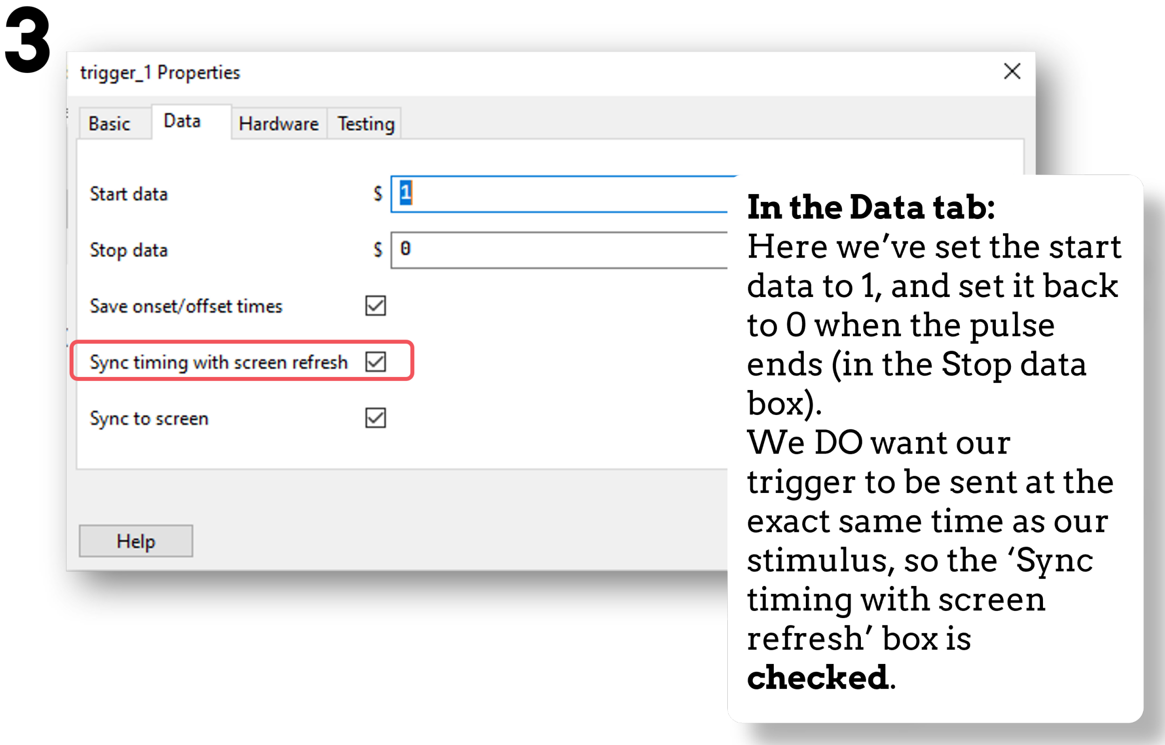
Stop: duration (s) — 0.1

Expected duration (s):

In the Basic tab:
Select 'condition' from the 'Start' drop down.

Enter the duration that your trigger should be on for in the 'Stop' field. Ensure that you leave the trigger on for long enough to be detected by your device. One frame's duration should be long enough, but here we've used 100ms.

Fig. 9.2: In the *Basic* tab, we'll choose to start our trigger when a condition is met by selecting *condition* from the *Start* drop down.



3

trigger_1 Properties

Start data $ 1

Stop data $ 0

Save onset/offset times ☑

Sync timing with screen refresh ☑

Sync to screen ☑

In the Data tab:
Here we've set the start data to 1, and set it back to 0 when the pulse ends (in the Stop data box).
We DO want our trigger to be sent at the exact same time as our stimulus, so the 'Sync timing with screen refresh' box is **checked**.

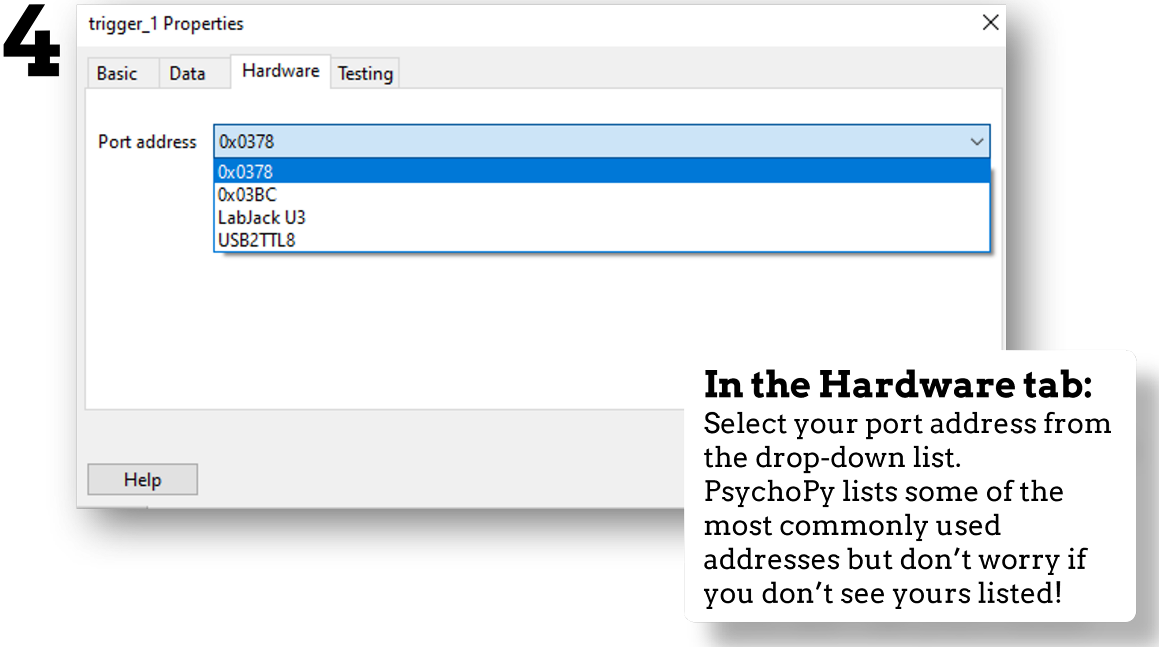
Fig. 9.3: Set the data to be sent by the trigger

- We can check this on our computer (not within PsychoPy itself) by navigating to: *Device Manager > Ports > Find the parallel port that you are using from the drop down > Right-click Properties > Resources tab >* The port's address is under the Settings header.

Note

The **address** of the port is not the same as the **name** of the port. For instance, the name of the port could be "LPT 1" but the address might be "0378".

- Now, in the *Hardware* tab of the parallel port component in PsychoPy, select the correct parallel port address:



In the Hardware tab: Select your port address from the drop-down list. PsychoPy lists some of the most commonly used addresses but don't worry if you don't see yours listed!

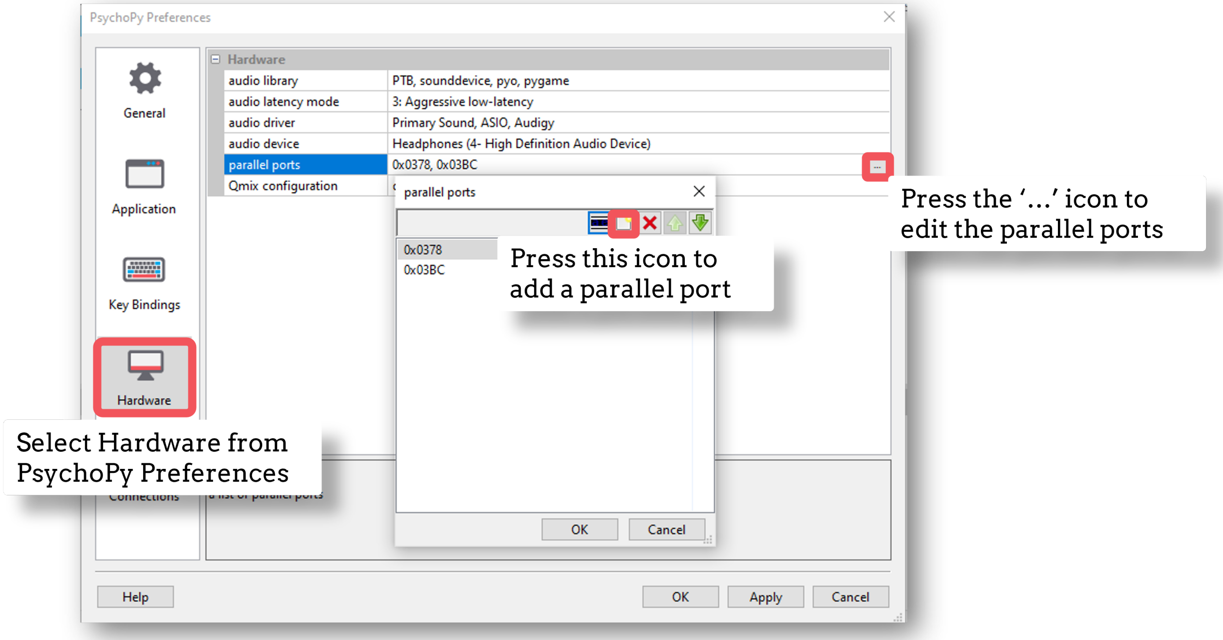
Fig. 9.4: Select your port from the drop down, if you don't see it listed just follow the next step.

- If you do not see the correct address in the drop down, in PsychoPy navigate to: *File > Preferences > Hardware > Parallel Ports > Click the "…" icon > Click the New Item icon > Enter the parallel port address > OK > Apply*:

Note

The parallel port address is usually a hexadecimal address. We tell PsychoPy to read it as such by prefixing with "0x". So if your port address appears in Device Manager as "0378-037F" for example, in PsychoPy this would be written as "0x0378".

- The correct port address will now appear in the drop down menu in the *Hardware* tab of the *Parallel Port* component.



Hardware

audio library	PTB, sounddevice, pyo, pygame
audio latency mode	3: Aggressive low-latency
audio driver	Primary Sound, ASIO, Audigy
audio device	Headphones (4- High Definition Audio Device)
parallel ports	0x0378, 0x03BC
Qmix configuration	

0x0378
0x03BC

Press the '...' icon to edit the parallel ports

Press this icon to add a parallel port

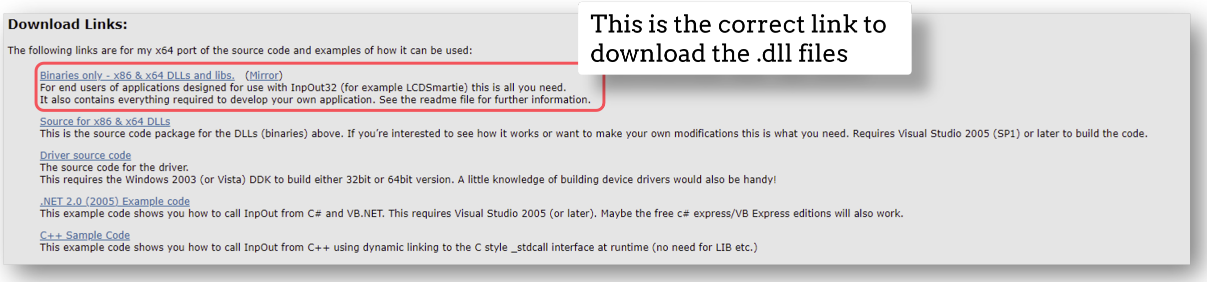
Select Hardware from PsychoPy Preferences

Fig. 9.5: Follow these steps to add your port address, only if it was not already in the drop-down menu.

Step two: Make sure you have the correct drivers installed

If you're using a Mac, it's recommended that you skip this step. For Windows users, a common error when trying to communicate via a Parallel Port component is that certain drivers are not found. We're going to pre-empt that error by downloading and installing the correct drivers now.

- Download the InpOutx64.dll and InpOutx32.dll files from here. You need to use the "Binaries only - x86 & x64 DLLs and libs" option under the *Download Links* subheading near the bottom of the page:



Download Links:

The following links are for my x64 port of the source code and examples of how it can be used:

Binaries only - x86 & x64 DLLs and libs. (Mirror)
For end users of applications designed for use with InpOut32 (for example LCDSmartie) this is all you need.
It also contains everything required to develop your own application. See the readme file for further information.

Source for x86 & x64 DLLs
This is the source code package for the DLLs (binaries) above. If you're interested to see how it works or want to make your own modifications this is what you need. Requires Visual Studio 2005 (SP1) or later to build the code.

Driver source code
The source code for the driver.
This requires the Windows 2003 (or Vista) DDK to build either 32bit or 64bit version. A little knowledge of building device drivers would also be handy!

.NET 2.0 (2005) Example code
This example code shows you how to call InpOut from C# and VB.NET. This requires Visual Studio 2005 (or later). Maybe the free c# express/VB Express editions will also work.

C++ Sample Code
This example code shows you how to call InpOut from C++ using dynamic linking to the C style _stdcall interface at runtime (no need for LIB etc.)

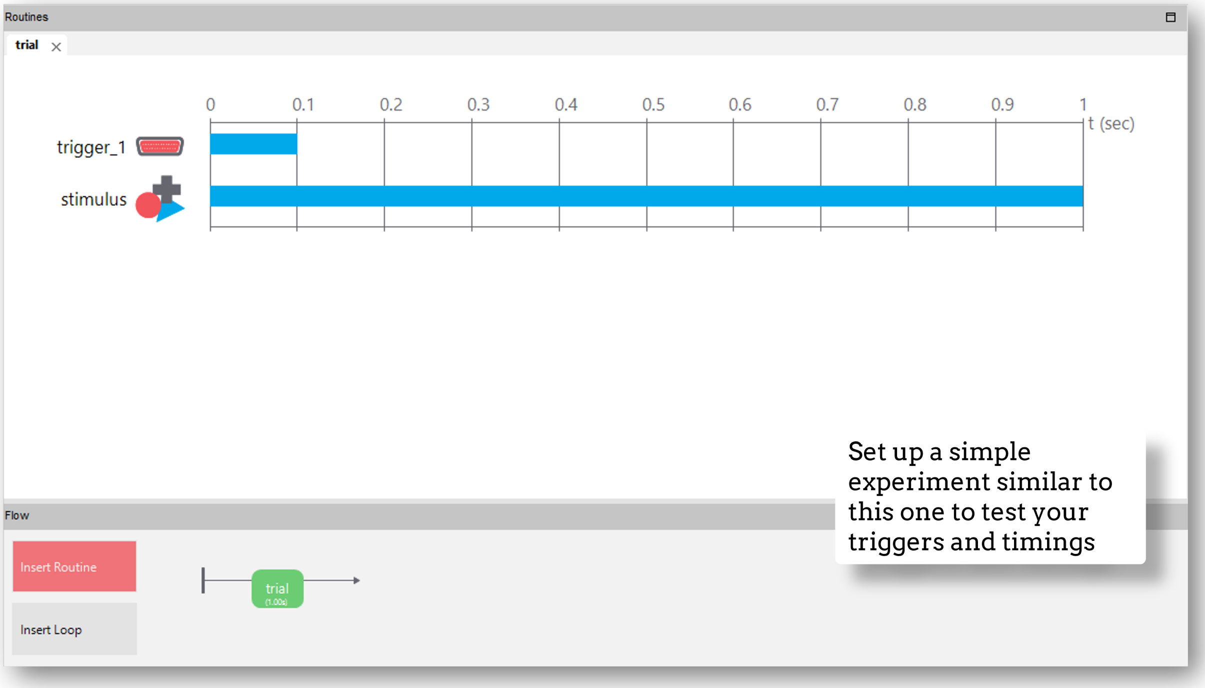
This is the correct link to download the .dll files

Fig. 9.6: The correct folder to select is shown here.

- When downloaded, find and extract the .zip folder. This will be called something like "InpOutBinaries_1501.zip".
- In the unzipped folder, find and copy the files "inpoutx64.dll" and "inpoutx64.lib" from the x64 folder, and then the file "inpout32.h" from the Win32 folder. Place a copy of all of these in the **same folder as your PsychoPy experiment file (the one with the .psyexp filetype)**.
- Restart PsychoPy (save your experiment first!)

Step three: Test your triggers

- To check that everything works, we recommend that you set up a very basic experiment that looks similar to this:



Set up a simple experiment similar to this one to test your triggers and timings

- Turn on your EEG recording device and start recording as you would in your actual experiment, and just check that you see triggers coming through.
- It's a good idea at this point to also check the timing of your stimulus presentation and your triggers using, for example, a photodiode for visual stimuli.
- Doing these checks with a very basic experiment just means that you don't accidentally change something on your real experiment file that you don't want to, and also means you don't have to disable components or sit through lots of instructions etc!

If there is a problem - We want to know!

If you have followed the steps above and are having an issue with triggers, please post details of this on the PsychoPy Forum.

We are constantly looking to update our documentation so that it's easy for you to use PsychoPy in the way that you want to. Posting in our forum allows us to see what issues users are having, offer solutions, and to update our documentation to hopefully prevent those issues from occurring again!

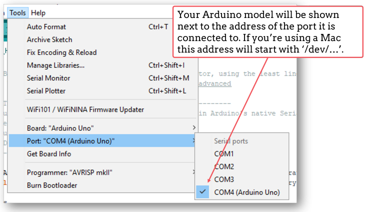
9.1.2 Recording information from an Arduino via serial port

Arduino microcontrollers are a relatively cost-effective way to record biophysical responses to stimuli, such as galvanic skin response (GSR) or heart rate. This page will guide you through how to record information from an Arduino via a serial port connection.

This guide will cover how to set up your PsychoPy experiment only - for lots of tutorials on using your Arduino, and also how to download the open-source Arduino software, take a look at the Arduino website.

Step one: Find out the address of your serial port

You can quickly find out the address of the serial port that your Arduino is connected to by opening the Arduino IDE and clicking on *Tools* at the top of the window, then down to *Port*. Here, the port that your Arduino is connected to will show the model of your Arduino next to it.



Your Arduino model will be shown next to the address of the port it is connected to. If you're using a Mac this address will start with '/dev/…'.

Step two: Add code components to your Builder experiment

Let's assume for this tutorial that we have a basic experiment set up where we are presenting an image stimulus to a participant, and we want to record their heart rate, via a module connected to an Arduino, during viewing.

- The first thing we'll need to do is initiate the communication between PsychoPy and the Arduino. We do this by adding in a code component to a routine at the start of the experiment (such as an instructions routine).
- In the *Begin Experiment* tab of that code component, add the following code to import the necessary libraries:

```
import serial
import time
```

- Then in that same code component, in the *End Routine* tab, we're going to add in code to start the communication between PsychoPy and Arduino. This will also initialise the Arduino:

```
port = serial.Serial('COM4', 9600) #Change 'COM4' here to the address of the serial port your Arduino is connected to. '9600' is the Baudrate, and this should be set to the same rate as that of your Arduino.
time.sleep(1) #Give the Arduino some time to wake up!
```

- Next, we'll add a code component to our trial routine. This component will record the information that the Arduino is sending over the serial port. We'll add it here to record information on every frame when the stimulus is presented, as we want to know how the participant's heart rate changes over the course of the stimulus.
- In the *Begin Routine* tab of this code component, add the following code to set up a list in which you'll record your data:

```
res = []
```

- Then in the *Each Frame* tab of that same code component, add the following to get PsychoPy to read the information sent over the serial port by Arduino:

```
res.append(port.readline())
```

- Now in the *End Routine* tab, we're going to ask PsychoPy to save the data to our .csv data file. But in this case we want **only the numbers** that are sent. You might have noticed that Arduino sends things like '\n' along with its data. This isn't always helpful for analysis, so we'll ask PsychoPy to ignore those values and save only a list of integers in our data file:

```
numbers = [] #Make a list to put the numbers only in
```

(continues on next page)

(continued from previous page)

```
for i, string in enumerate(res):
    for word in string.split():
        if word.isdigit():
            numbers.append(int(word))
thisExp.addData('heart_rate', numbers) #Add the list to our data file - 'heart_rate' will be the name of this column in our .csv file.
```

- Finally, we're going to close the port when the experiment ends. To do this, add the following to the *End Experiment* tab of any code component:

```
port.close()
```

- You should now have an experiment that reads and records the information being sent by an Arduino. Here we used heart rate as an example, but this code can easily be adapted to record any information that your Arduino is sending.

If there is a problem - We want to know!

If you have followed the steps above and are having an issue, please post details of this on the PsychoPy Forum.

We are constantly looking to update our documentation so that it's easy for you to use PsychoPy in the way that you want to. Posting in our forum allows us to see what issues users are having, offer solutions, and to update our documentation to hopefully prevent those issues from occurring again!

9.1.3 Sending triggers via a Serial Port

Note that if you are using PsychoPy version 2022.2 onwards, you may use the *serial port component*. If you are using an earlier version you will need to use :ref: *code components <serial_code>*. For both use cases you will need to know your serial port address.

Find out the address of your serial port

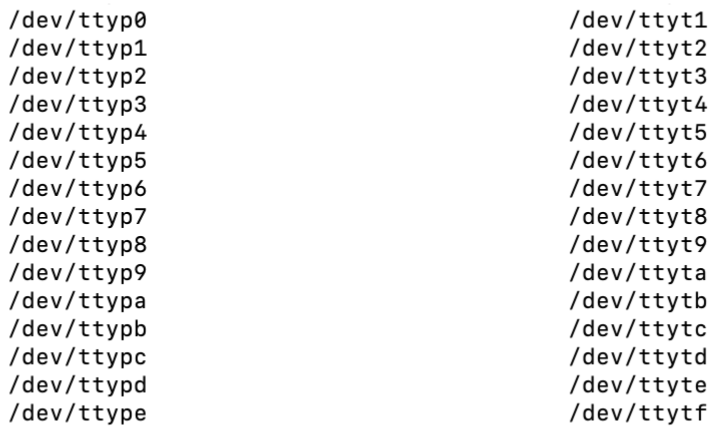
Serial port addresses are different depending on whether you're using a Mac or a Windows device:

If you're using a Mac

- Open a *Terminal* window and type:

```
ls dev/tty*
```

- In the terminal window, you'll see a long list of port names like in the screenshot below:

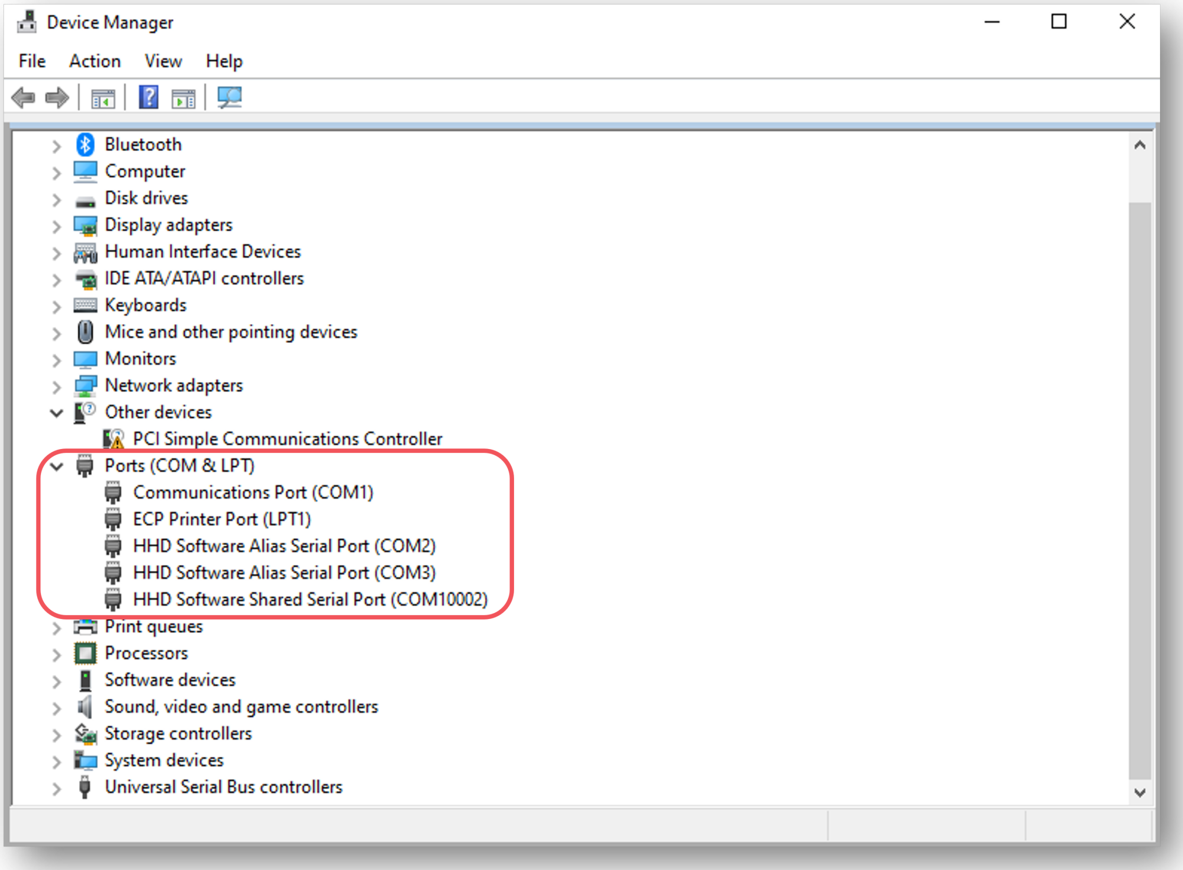


```
/dev/ttyp0                              /dev/ttyt1
/dev/ttyp1                              /dev/ttyt2
/dev/ttyp2                              /dev/ttyt3
/dev/ttyp3                              /dev/ttyt4
/dev/ttyp4                              /dev/ttyt5
/dev/ttyp5                              /dev/ttyt6
/dev/ttyp6                              /dev/ttyt7
/dev/ttyp7                              /dev/ttyt8
/dev/ttyp8                              /dev/ttyt9
/dev/ttyp9                              /dev/ttyta
/dev/ttypa                              /dev/ttytb
/dev/ttypb                              /dev/ttytc
/dev/ttypc                              /dev/ttytd
/dev/ttypd                              /dev/ttyte
/dev/ttype                              /dev/ttytf
```

- To find out which one your device is connected to, you can remove and replace your device to see which port name is changing.

If you're using Windows

- Open the *Device Manager* and click on the *Ports* drop down to show available ports like in the screenshot below:



Bluetooth
Computer
Disk drives
Display adapters
Human Interface Devices
IDE ATA/ATAPI controllers
Keyboards
Mice and other pointing devices
Monitors
Network adapters
Other devices
PCI Simple Communications Controller
Ports (COM & LPT)
Communications Port (COM1)
ECP Printer Port (LPT1)
HHD Software Alias Serial Port (COM2)
HHD Software Alias Serial Port (COM3)
HHD Software Shared Serial Port (COM10002)
Print queues
Processors
Software devices
Sound, video and game controllers
Storage controllers
System devices
Universal Serial Bus controllers

- If it's not obvious which port your device is connected to, remove and replace your device to see which port name changes.

Using a Serial Port Component to communicate via Serial Port

If you're using PsychoPy version 2022.2 or later, you can use the serial port component. If you're running an earlier version, you'll need to use a code component (see :ref: *this section <serial_code>*).

- The serial port component can be found in both the I/O and EEG component drop down menus. Add in a serial port component to the routine that you'd like triggers to be sent from by selecting it from the menu:
- Now, imagine we want our trigger sent to indicate stimulus onset. We *could* do this by simply setting the onset time of the trigger to match that of our stimulus. But this is not the **most** precise way to do this. Also, this doesn't help us if we want to send our trigger to indicate something with variable timing, such as when a response is made.
- For maximum precision, we'll set the trigger to be sent when the status of our stimulus is set to *started*:
- Now we set that condition by inserting the following code:

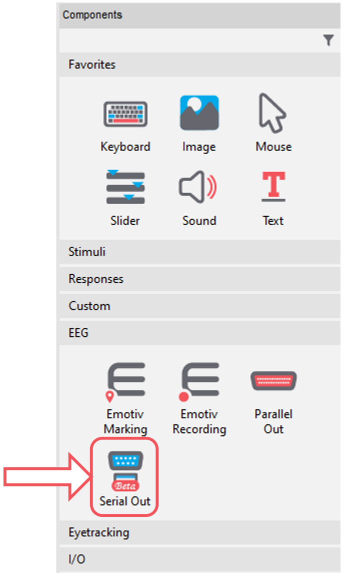
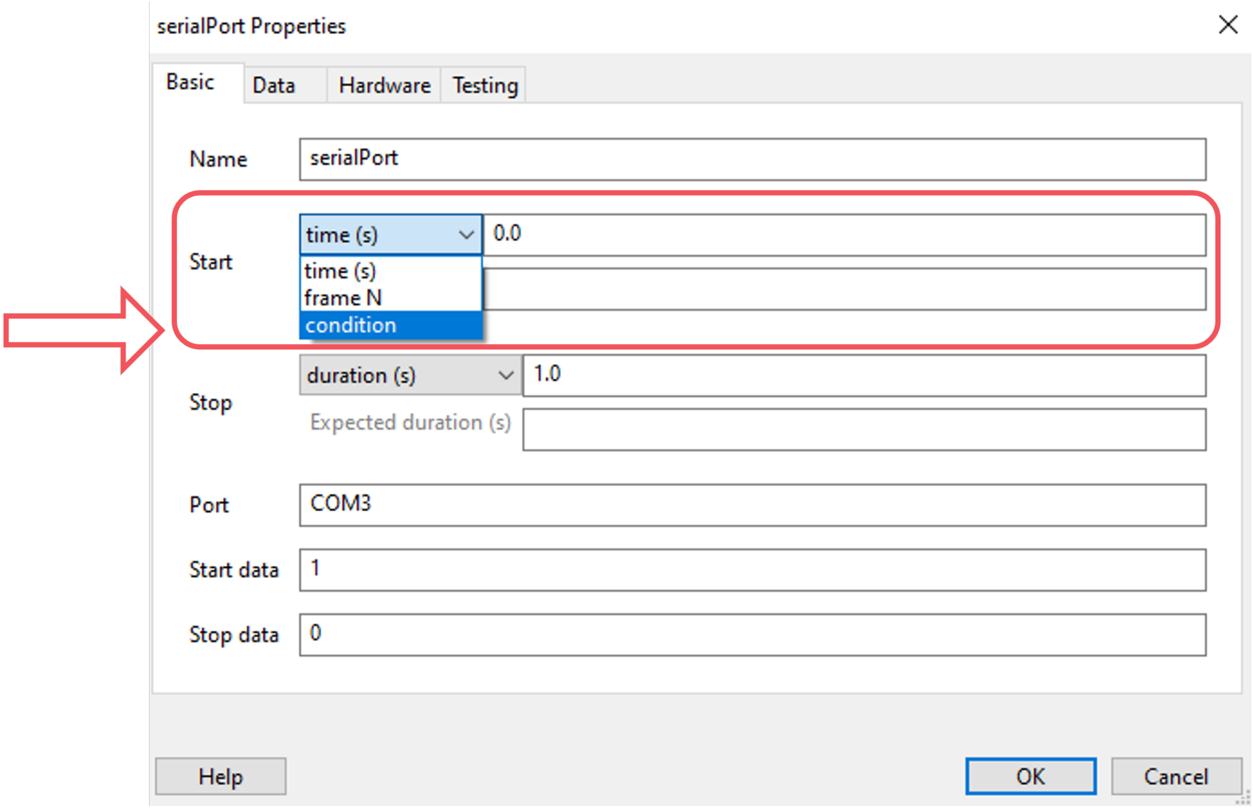


Fig. 9.7: Select the *SerialPort* component from the *I/O* or *EEG* component drop-down menus.



Name: serialPort

Start: time (s) 0.0

Stop: duration (s) 1.0

Expected duration (s)

Port: COM3

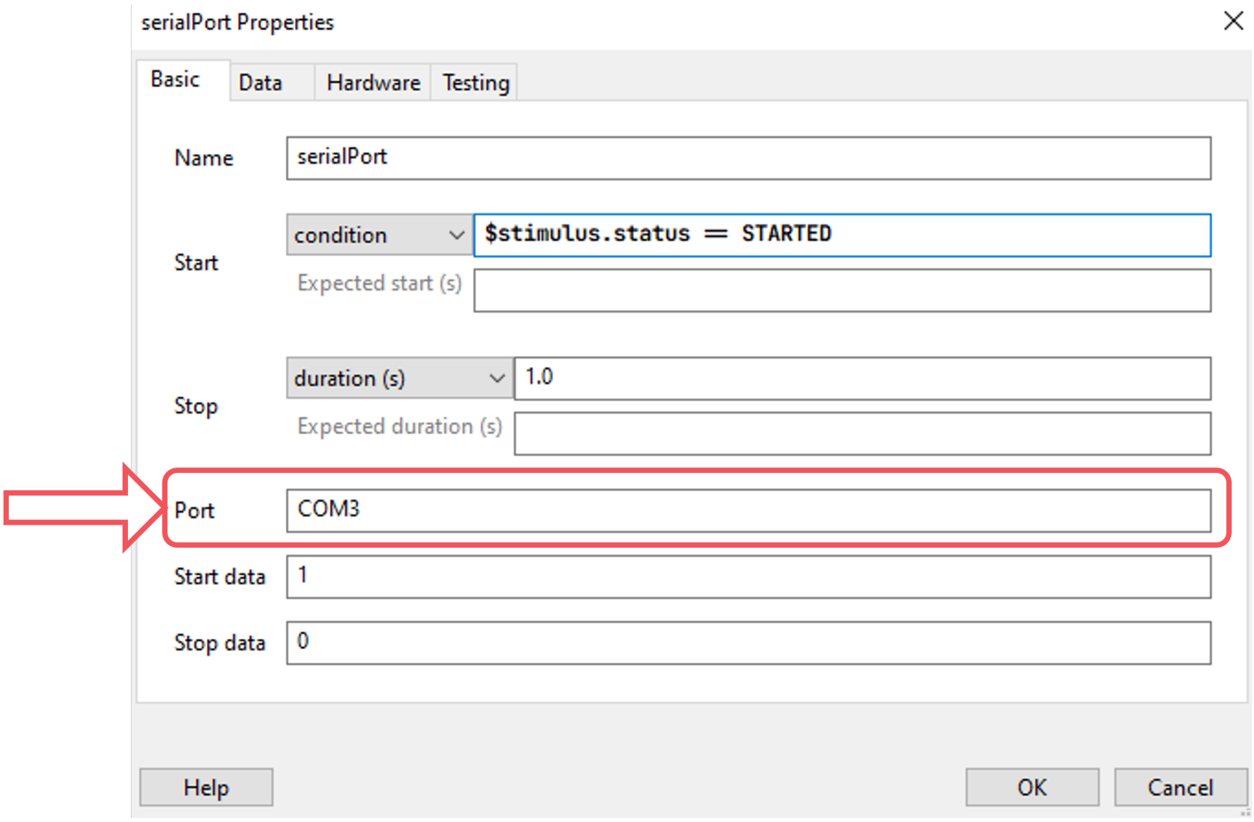
Start data: 1

Stop data: 0

Fig. 9.8: In the *Basic* tab, we'll choose to start our trigger when a condition is met by selecting *condition* from the *Start* drop down.

```
stimulus.status == STARTED #Change 'stimulus' here to match the name of your own␣
↪component
```

- Next, we need to set the address of the serial port that we want to use. To do this, write the address of the port in the *Port* field:



serialPort Properties

Name: serialPort

Start: condition — $stimulus.status == STARTED

Expected start (s):

Stop: duration (s) — 1.0

Expected duration (s):

Port: COM3

Start data: 1

Stop data: 0

Fig. 9.9: Type in the address of your serial port.

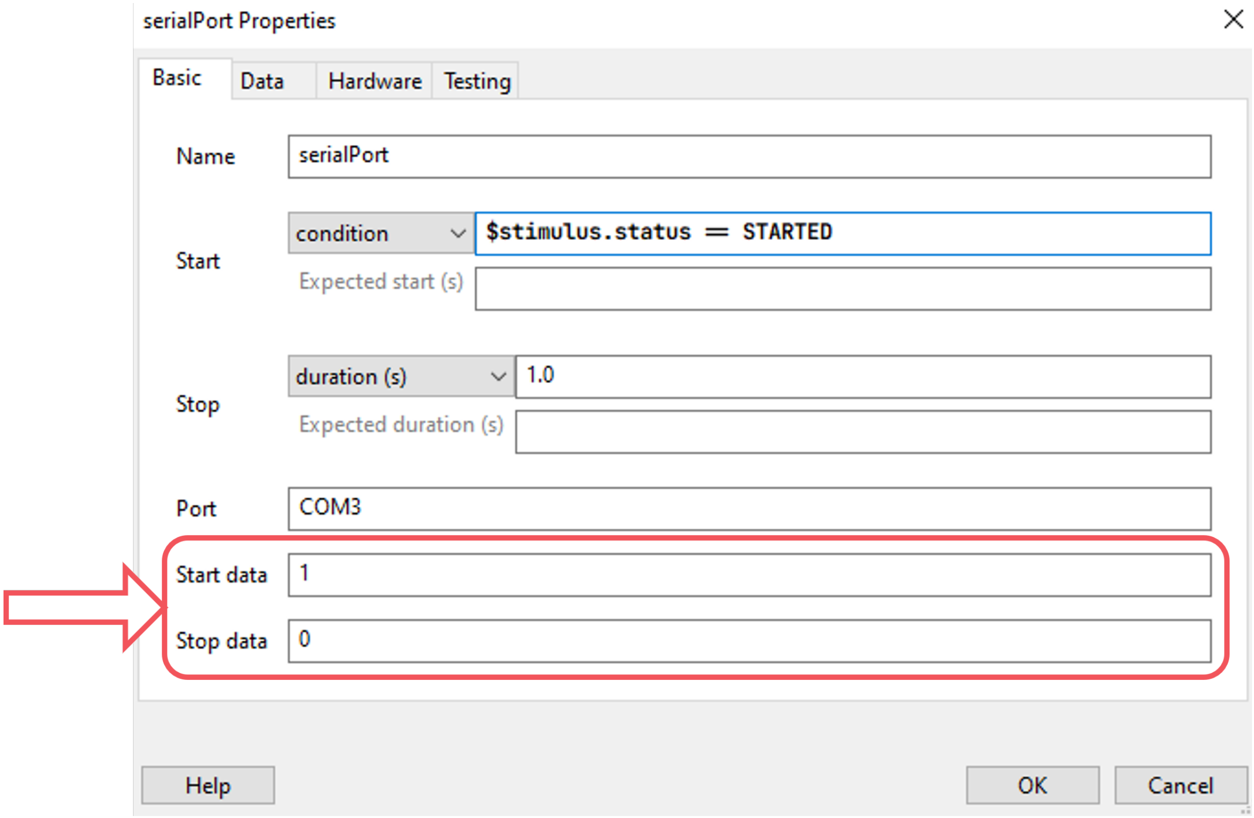
- Next, we'll set the data that we'd like to send to the device at the start of the pulse, and what we want it to be reset to at the end of the pulse. Do this by completing the *Start data* and *Stop data* fields:
- By default, any integers that you type in these fields will be converted to characters. So the integer 1 will be converted to the character "1". If you want to send the **number** 1, enter the following into the *Start/Stop data* fields:

```
chr(1) # Where 1 is the integer you want to send
```

- You can also reference a variable from your conditions file in the *Start/Stop data* fields using `$`, as long as those variables are strings.
- Now that your serial port component is set up, we now recommend that you :ref: *test your triggers <trigger_test>*.

Using a Code Component to communicate via Serial Port

- First, add in a code component to your *Instructions* routine (or something similar, at the start of your experiment):
- In the *Begin Experiment* tab, copy and paste the following code which will import the serial library and initiate PsychoPy's communication with your serial port - be sure to change `COM3` to the correct serial port address for your device:



serialPort Properties

Name: serialPort

Start: condition $stimulus.status == STARTED

Expected start (s):

Stop: duration (s) 1.0

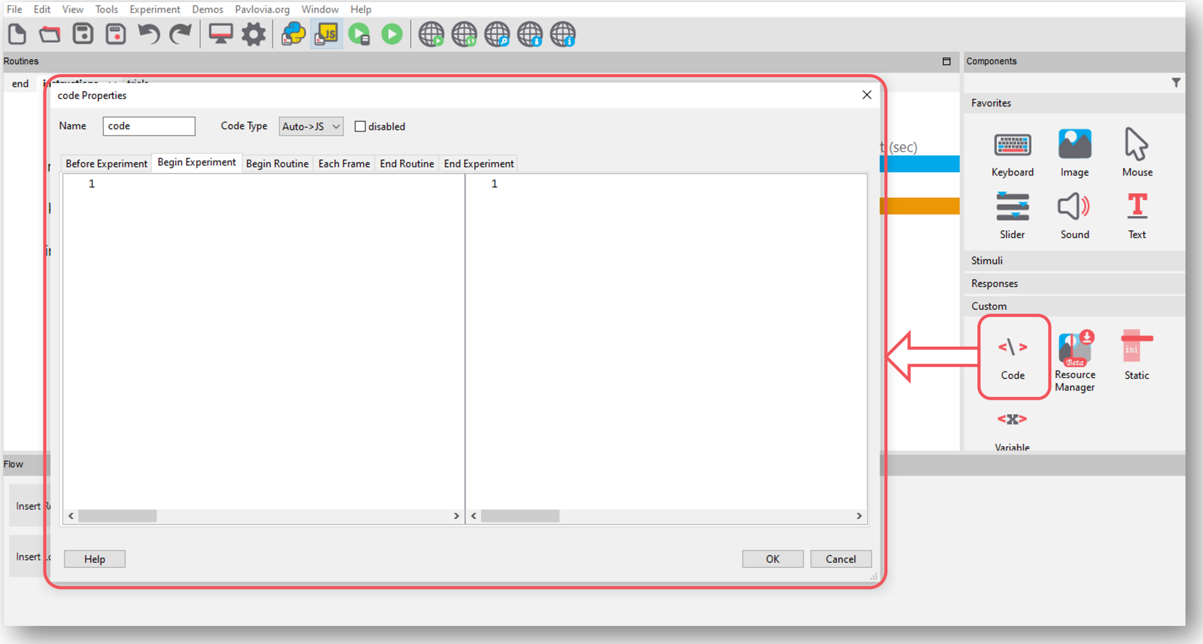
Expected duration (s):

Port: COM3

Start data: 1

Stop data: 0

Fig. 9.10: What do you want PsychoPy to send at the start of your trigger pulse, and what do you want it to be reset to at the end of the pulse?



code Properties

Name: code

Code Type: Auto->JS

☐ disabled

Fig. 9.11: Select the *Code component* from the *Custom* component drop-down

```
import serial #Import the serial library
port = serial.Serial('COM3') #Change 'COM3' here to your serial port address
```

- Now, copy and paste the following code component to your trials routine in the *Begin Routine* tab (or whichever routine you want to send triggers from):

```
stimulus_pulse_started = False
stimulus_pulse_ended = False
```

- In the same routine, copy and paste the following code in the *Each Frame* tab - be sure to change *stimulus* in line 1 to match the name of the component that you want to send the triggers for:

```
if stimulus.status == STARTED and not stimulus_pulse_started: #Change 'stimulus' to match the name of the component that you want to send the trigger for
    win.callOnFlip(port.write, str.encode('1'))
    stimulus_pulse_start_time = globalClock.getTime()
    stimulus_pulse_started  = True

if stimulus_pulse_started and not stimulus_pulse_ended:
        if globalClock.getTime() - stimulus_pulse_start_time >= 0.005:
            win.callOnFlip(port.write,  str.encode('0'))
            stimulus_pulse_ended = True
```

- This code will send a '1' to your device at the onset of the stimulus component, and then reset back to '0'. You can change these values to whatever is meaningful to your data, including asking PsychoPy to pull the value from your conditions file.
- Finally, in a routine at the end of your experiment (the *Thanks for participating* screen for example) copy and paste the following:

```
port.close()
```

- We now recommend that you :ref: *test your triggers <trigger_test>*.

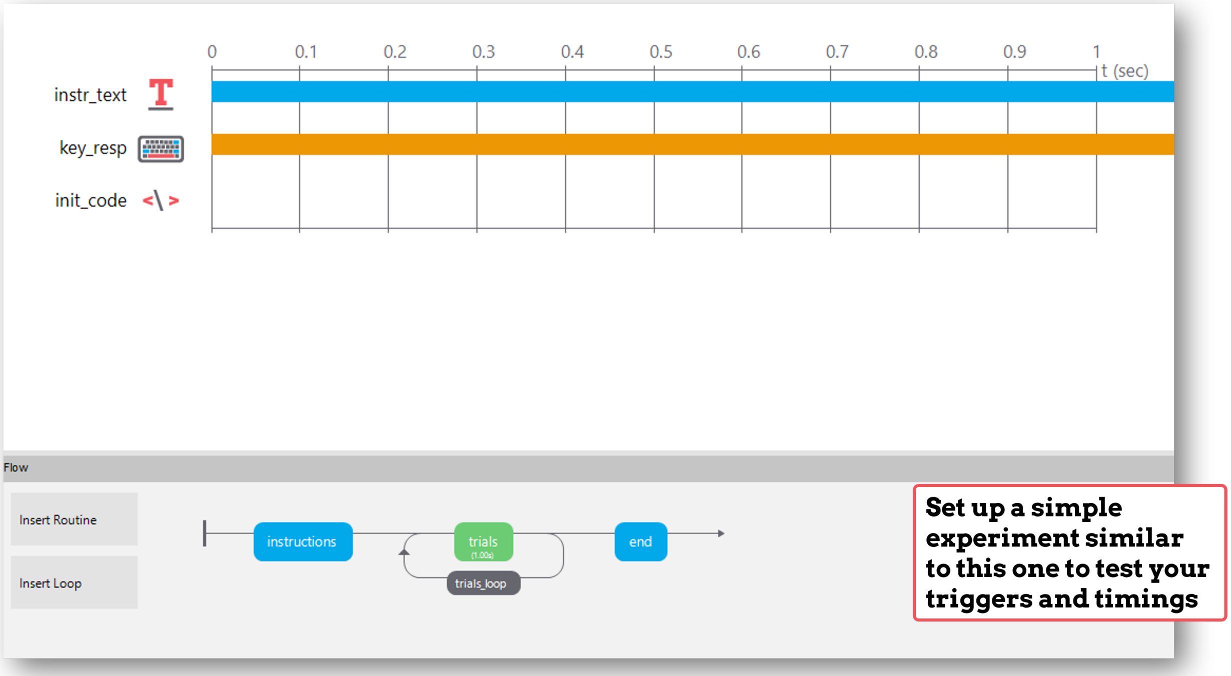
Test your triggers

- To check that everything works, we recommend that you set up a very basic experiment that looks similar to this:
- Turn on your EEG recording device and start recording as you would in your actual experiment, and just check that you see triggers coming through.
- It's a good idea at this point to also check the timing of your stimulus presentation and your triggers using, for example, a photodiode for visual stimuli.
- Doing these checks with a very basic experiment just means that you don't accidentally change something on your real experiment file that you don't want to, and also means you don't have to disable components or sit through lots of instructions etc!

If there is a problem - We want to know!

If you have followed the steps above and are having an issue with triggers, please post details of this on the PsychoPy Forum.

We are constantly looking to update our documentation so that it's easy for you to use PsychoPy in the way that you want to. Posting in our forum allows us to see what issues users are having, offer solutions, and to update our documentation to hopefully prevent those issues from occurring again!



Set up a simple experiment similar to this one to test your triggers and timings

9.1.4 Sending triggers via EGI NetStation

Communicating via EGI NetStation is very similar to communicating via a serial port, in that you'll need to add some code components into your experiment.

The egi-pynetstation package allows communication using an NTP protocol. It is important to first verify your hardware setup. The code is compatible with EGI (also known as Philips EGI and most recently MagStim-EGI) amplifiers 300 and 400 series. While 400 series amplifiers serve as their own NTP server so are able to work with newer macOS versions (10.14.x as of April 2023). If you are using a 300-series amplifier you must be using macOS 10.12. This is because EGI has configured its own NTP server for use with 300-series amps; users who wish to alter this configuration with a 300-series amp should continue to use PsychoPy2's EGI package or investigate ports of the old package to Python3.

The old EGI "pynetstation" package uses a "polling" method of asking the EGI system what time it is before sending events. While generally accurate, delays in drawing to the screen can cause (usually minor) inconsistencies in timing. This pakcage's implementation of NTP timing should be superior to the older method.

Step one: Verify your Amplifier and NTP server are active

Users of 300 series amplifiers should open a terminal and run the following command: `sntp -d localhost`

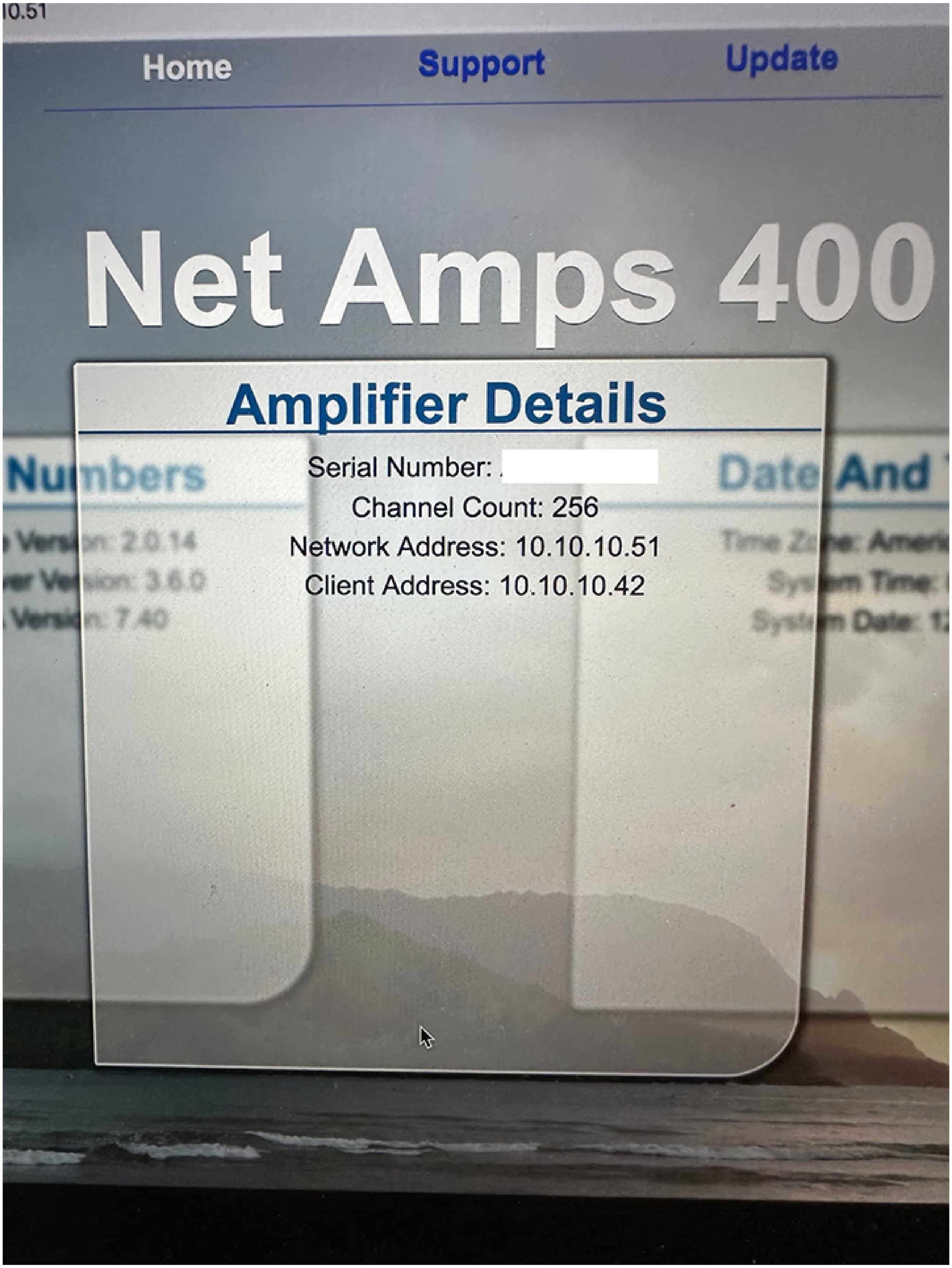
Users of 400 series amplifiers may input the IP of your amplifier (usually 10.10.10.51) in the command above or open the webpage associated with the amplifier on the EGI laptop/desktop.

Notice that this page gives information about your amplifier address (10.10.10.51) and the Net Station computer (10.10.10.42).

Step two: Install EGI NetStation Python Library on older versions of PsychoPy

If you're using PsychoPy version 2022.1.3 or older, you'll need to install the EGI NetStation library using the Command Prompt in Windows. You will only need to do this once.

- To access the Command Prompt, just type *Command Prompt `into the search bar next to your `Start Menu* icon and select it.



Net Amps 400

Amplifier Details

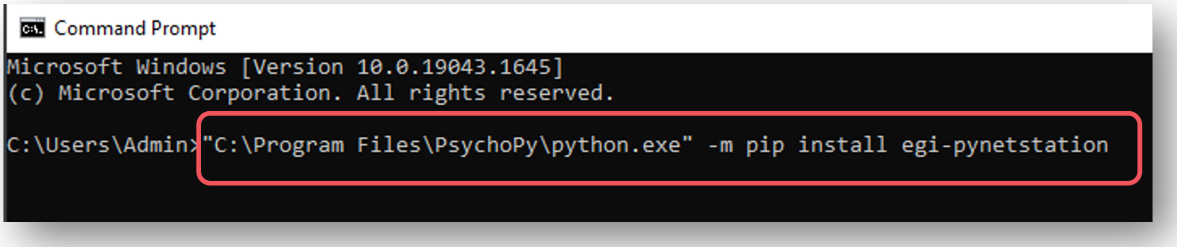
Serial Number: [redacted]

Channel Count: 256

Network Address: 10.10.10.51

Client Address: 10.10.10.42

- You now need to copy the file path to the file *python.exe* that is **inside** your PsychoPy folder (usually this is installed in `C:\Program Files\PsychoPy`).
- When you've found the PsychoPy folder, copy the file path and paste it into the Command Prompt, surrounded by quotation marks (" ").
- Now, add `\python.exe` to the line, so that the line reads: `"C:\Program Files\PsychoPy\python.exe"` (or similar, depending on where your PsychoPy is saved).
- Finally, add `-m pip install egi-pynetstation` to the line.
- Your line should now look similar to this: `"C:\Program Files\PsychoPy\python.exe" -m pip install egi-pynetstation` as shown in the following screenshot:



```
Microsoft Windows [Version 10.0.19043.1645]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Admin>"C:\Program Files\PsychoPy\python.exe" -m pip install egi-pynetstation
```

You're now ready to go!

Step three: Verify / Update the egi-pynetstation package to 1.0.1

If you are using Psychopy versions from 2023 or later, please verify that you are using version 1.0.1 of the package the PsychoPy package manager update the package.

- From the menu system, select Tools, Plugin/Package manager
- Select the "Packages" tab
- Search for "egi_pynetstation"
- Verify the installed version is 1.0.1
- If necessary, click the "Install" button

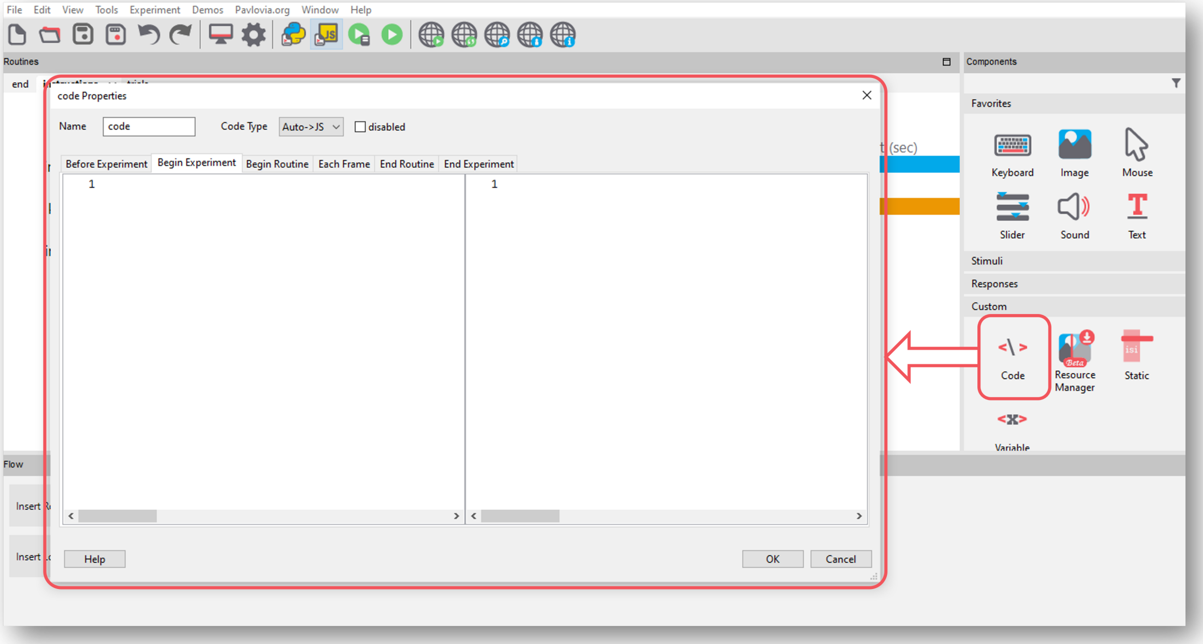
Step four: Add code components into your Builder experiment

To communicate with your NetStation EEG hardware, you'll need to add in some Python code components to your experiment.

- First, add in a code component to your *Instructions* routine (or something similar, at the start of your experiment):
- In the *Begin Experiment* tab, copy and paste the following code which will import the relevant libraries and set up the communication with your NetStation - be sure to change the IP address of the NetStation so that it matches that of your own NetStation:

```
#Import Netstation library
from egi_pynetstation.NetStation import NetStation

#IP address of NetStation - CHANGE THIS TO MATCH THE IP ADDRESS OF YOUR NETSTATION
IP_ns = '10.10.10.42'

#IP address of amplifier (if using 300
```

(continues on next page)



Name: code
Code Type: Auto->JS
☐ disabled

Fig. 9.12: Select the *Code component* from the *Custom* component drop-down

(continued from previous page)

```
#series, this is the same as the IP address of
#NetStation. If using newer series, the amplifier
#has its own IP address)
IP_amp = '10.10.10.51'

#Port configured for ECI in NetStation - CHANGE THIS IF NEEDED
port_ns = 55513

#Start recording and send trigger to show this
eci_client = NetStation(IP_ns, port_ns)
eci_client.connect(ntp_ip = IP_amp)
eci_client.begin_rec()
eci_client.send_event(event_type = 'STRT', start = 0.0)
```

- Now, copy and paste the following code component to your trials routine in the *Begin Routine* tab, this just (re)sets a value at the start of the routine to indicate that no trigger has yet been sent:

```
triggerSent = False
eci_client.resync()
```

- Now, in the *Each Frame* tab of that same code component, add the following code to send a trigger OF NO MORE THAN FOUR CHARACTERS when your stimulus is presented. The `.status` attribute here is checking whether the our stimulus has started, and if it has, PsychoPy sends the trigger to EGI NetStation. Note that most components in PsychoPy have the `.status` attribute, so you could easily adapt this code to, for example, send a trigger when a response key is pressed:

```
#Send trigger to NetStation - Change 'stim' to
```

(continues on next page)

(continued from previous page)

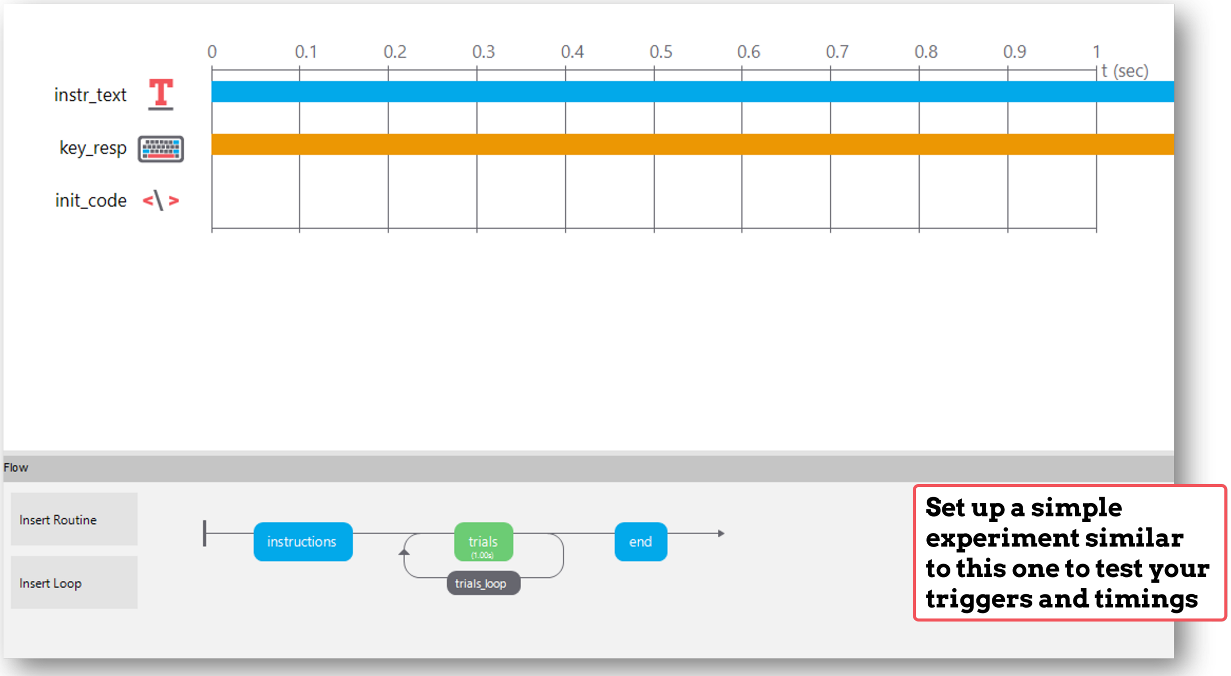
```
#a meaningful trigger for your experiment OF NO MORE THAN FOUR CHARACTERS. You can
#also set the trigger in a conditions file.

if stimulus.status == STARTED and not triggerSent: #If the stimulus component has␣
↪started and the trigger has not yet been sent. Change 'stimulus' to match the name␣
↪of the component you want the trigger to be sent at the same time as
    win.callOnFlip(eci_client.send_event, event_type = 'stim', label='stim') #Send␣
↪the trigger, synced to the screen refresh
    triggerSent = True #The trigger has now been sent, so we set this to true to␣
↪avoid a trigger being sent on each frame
```

- Finally, in a routine at the end of your experiment (the *Thanks for participating* screen for example) copy and paste the following:

```
#Stop recording and disconnect
eci_client.end_rec()
eci_client.disconnect()
```

Step five: Test your triggers

- To check that everything works, we recommend that you set up a very basic experiment that looks similar to this:

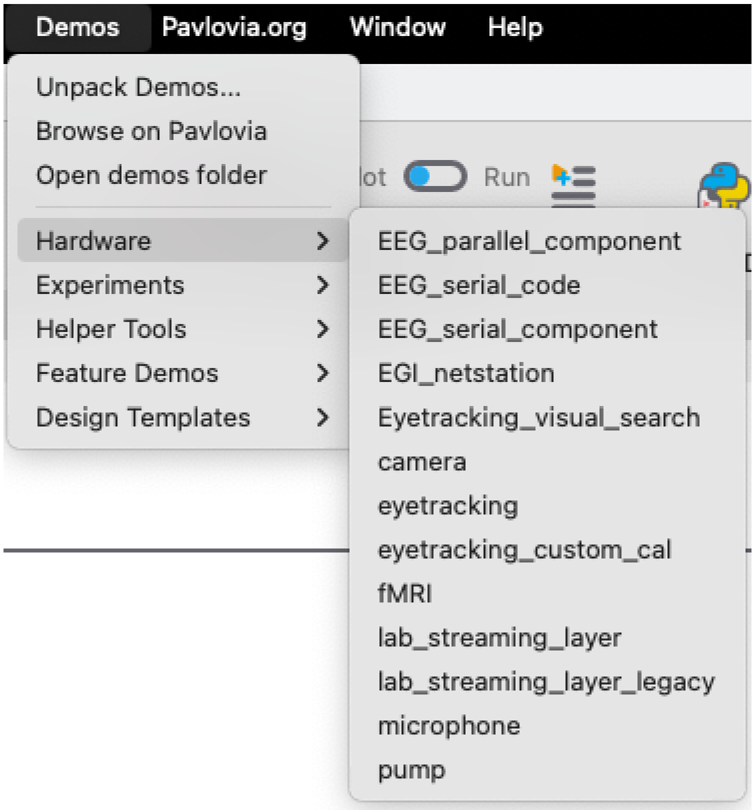


Set up a simple experiment similar to this one to test your triggers and timings

- Turn on your EEG recording device and start recording as you would in your actual experiment, and just check that you see triggers coming through.
- It's a good idea at this point to also check the timing of your stimulus presentation and your triggers using, for example, a photodiode for visual stimuli.
- Doing these checks with a very basic experiment just means that you don't accidentally change something on your real experiment file that you don't want to, and also means you don't have to disable components or sit through lots of instructions etc!

See Built-in Example: Stroop Task

There is a complete experiment built into PsychoPy demonstrating EEG triggers to the EGI amplifier.



To access the demo:

- Select "Demos" menu
- If not previously done, select "Unpack Demos"
- Select the "Demos" menu again, click "Hardware", select "EGI_netstation"
- This built-in demo should run and send appropriate triggers to the EGI amplifier/computer

If there is a problem - We want to know!

If you have followed the steps above and are having an issue with triggers, please post details of this on the PsychoPy Forum.

Further documentation can be found on the egi-pynetstation RTD as well as their github project .

We are constantly looking to update our documentation so that it's easy for you to use PsychoPy in the way that you want to. Posting in our forum allows us to see what issues users are having, offer solutions, and to update our documentation to hopefully prevent those issues from occurring again!

9.1.5 Communicating with Brain Products Devices

This guide will talk you through how to use send triggers (event markers) from PsychoPy, via serial communication, to Brain Products' BrainVision Analyzer. We're assuming here that you're sending a trigger to mark the onset of a visual stimulus, but you can easily adapt the logic of when the trigger is sent, to mark other events too.

If you'd like to interface Brain Products' Remote Control Server 2 (RCS) from PsychoPy you can find out how to do that in this article

You can also find guidance on how to send LabStreaming Layer markers in this article from the BCI blog

Step one: Add code components into your Builder experiment

To communicate with Brain Products devices, you'll need to add in some Python code components to your experiment.

- First, add in a code component to your *Instructions* routine (or something similar, at the start of your experiment):

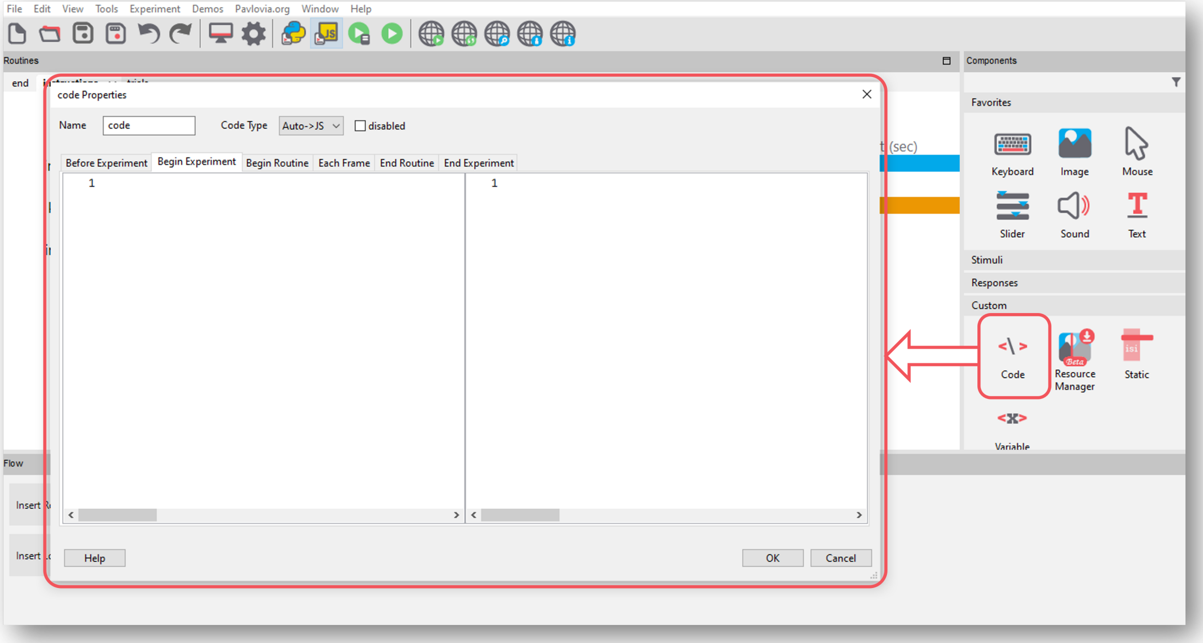


Fig. 9.13: Select the *Code component* from the *Custom* component drop-down

- In the *Begin Experiment* tab, copy and paste the following code which will import the relevant libraries and set up the communication with your Brain Products device:

```
# Import modules needed
import serial

# Define the send_triggers function to send trigger
# to serial port
def send_triggers(value):
    """Send value as hardware trigger"""
    port.write(value)

# Define serial port to be used
port = serial.Serial("COM3") ##CHANGE THIS TO THE ADDRESS OF YOUR SERIAL PORT
```

- Now, copy and paste the following code component to your trial routine in the *Begin Routine* tab, this just (re)sets a value at the start of the routine to indicate that no trigger has yet been sent:

```
#Mark the stimulus onset triggers as "not sent"
#at the start of the trial
stimulus_pulse_started = False
stimulus_pulse_ended = False
```

- Now, in the *Each Frame* tab of that same code component, add the following code to send a trigger when your stimulus is presented. The `.status` attribute here is checking whether the our stimulus has started, and if it has, PsychoPy sends the trigger. Note that most components in PsychoPy have the `.status` attribute, so you could easily adapt this code to send triggers on the onset of other components.:

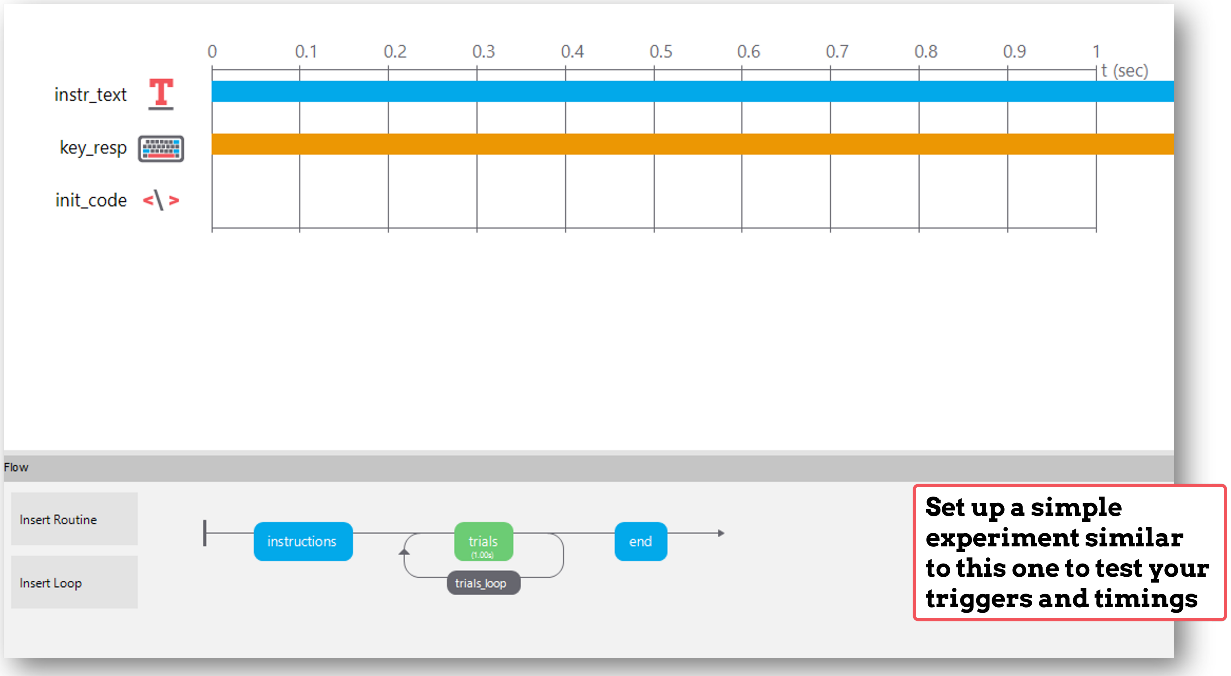
```
##STIMULUS TRIGGERS##
#Check to see if the stimulus is presented this frame
#and send the trigger if it is
if stimulus.status == STARTED and not stimulus_pulse_started: #If the stimulus␣
↪component has started and the trigger has not yet been sent. Change 'stimulus' to␣
↪match the name of the component you want the trigger to be sent at the same time␣
↪as
    win.callOnFlip(send_triggers, [0,0,0,0,0,0,0,1])#Send the trigger, synced to␣
↪the screen refresh
    stimulus_pulse_start_time = globalClock.getTime()
    stimulus_pulse_started  = True #The trigger has now been sent, so we set this␣
↪to true to avoid a trigger being sent on each frame

#If it's time to end the pulse, reset the value to "0"
#so that we don't continue sending triggers on every frame
if stimulus_pulse_started and not stimulus_pulse_ended:
    if globalClock.getTime() - stimulus_pulse_start_time >= 0.005:
        win.callOnFlip(send_triggers, [0,0,0,0,0,0,0,0])
        stimulus_pulse_ended = True
```

- Finally, in a routine at the end of your experiment (the *Thanks for participating* screen for example) copy and paste the following:

```
#Close the connection to the serial port
port.close()
```

Step four: Test your triggers

- To check that everything works, we recommend that you set up a very basic experiment that looks similar to this:
- Turn on your EEG recording device and start recording as you would in your actual experiment, and just check that you see triggers coming through.
- It's a good idea at this point to also check the timing of your stimulus presentation and your triggers using, for example, a photodiode for visual stimuli.
- Doing these checks with a very basic experiment just means that you don't accidentally change something on your real experiment file that you don't want to, and also means you don't have to disable components or sit through lots of instructions etc!



Set up a simple experiment similar to this one to test your triggers and timings

If there is a problem - We want to know!

If you have followed the steps above and are having an issue with triggers, please post details of this on the PsychoPy Forum.

We are constantly looking to update our documentation so that it's easy for you to use PsychoPy in the way that you want to. Posting in our forum allows us to see what issues users are having, offer solutions, and to update our documentation to hopefully prevent those issues from occurring again!

9.1.6 Communicating with fMRI

Due to the haemodynamic response being comparatively sluggish relative to scalp voltage changes, fMRI studies don't typically require sub-millisecond timing precision *within* a trial like EEG studies do.

However it **is** important that an fMRI study has consistent timing *across* trials so that the scanner sequence remains in sync with an experiment.

Step one: Know your Scanner!

Rather than programming your PsychoPy experiment to send triggers *to* some hardware in the same way as EEG, with fMRI you would want to set up your experiment so that it waits until it has detected when the scanner has sent out a trigger before moving on to present trials.

Before doing anything else, it's important that you know **how** the scanner you'll be using will emit these triggers, and whether these are converted to some other signal such as characters on a serial port or a simulated keypress. In general, there are at least 3 ways a scanner might send a trigger to your experiment:

1. Emmulate a keypress.
2. Via parallel port
3. Via serial port

Step two: Create a Routine to wait for scanner triggers

A Routine to detect fMRI triggers is really simple to set up. Regardless of the method your scanner uses to send the triggers, you'll just need a Routine that waits until it's detected the trigger before moving on. Create a new Routine and insert a Text component that says 'Waiting for Scanner'.

- **If your scanner emulates key presses:** ***This is the simplest of all communication methods!***
  - Insert a Keyboard component to your 'Waiting for Scanner' Routine. In 'allowed keys' use the key that the scanner will send e.g. if the scanner sends '5' allowed keys will be '5'.
  - Now, when the keypress is detected, the 'Waiting for Scanner' screen will end. Although, be careful! PsychoPy doesn't know the difference between the emulated key presses sent from the scanner and key presses made by a human being! So take care not to type on the keyboard connected to the PsychoPy computer whilst your experiment runs to avoid your key presses being mistaken for triggers.
- **If your scanner communicates via a Parallel Port:**
  - Insert a code component to your 'Waiting for Scanner' Routine
  - In the *Begin Experiment* tab of the code component, add the following code to set up the Parallel Port:

```
from psychopy.hardware.parallel import ParallelPort
triggers = ParallelPort(address = 0x0378) #Change this address to match the␣
↪address of the Parallel Port that the device is connected to
pinNumber = 4 #Change to match the pin that is receiving the pulse value␣
↪sent by your scanner. Set this to None to scan all pins
```

  - In the *Each Frame* tab of the same code component, add the following code to check for triggers:

```
if frameN > 1: #To allow the 'Waiting for Scanner' screen to display
    trig = triggers.waitTriggers(triggers = [pinNumber], direction = 1,␣
↪maxWait = 30)
    if trig is not None:
        continueRoutine = False #A trigger was detected, so move on
```

  - The 'Waiting for Scanner' message will now remain on the screen until the trigger is received from the scanner.
- **If your scanner communicates via a Serial Port:**
  - Insert a code component to your 'Waiting for Scanner' Routine
  - In the *Begin Experiment* tab of the code component, add the following code to set up the Serial Port:

```
from psychopy.hardware.serial import SerialPort
triggers = SerialPort('COM3', baudrate = 9600) #Change to match the address␣
↪of your Serial Port
trigger = '1' #Change to match the expected character sent from your␣
↪scanner, or set to None for any character
```

  - In the *Each Frame* tab of the same code component, add the following code to check for triggers:

```
if thisTrigger in self.read(self.inWaiting()):
    continueRoutine = False #Our trigger was detected, so move on
```

  - The 'Waiting for Scanner' message will now remain on the screen until the trigger is received from the scanner.

Timing in fMRI

In fMRI studies, it’s important that the scanner runs remain in sync with the experiment, especially if you are only waiting for the scanner to send a trigger once at the start of the experiment.

PsychoPy implements a feature called non-slip timing to help with this (you can find out more about what this is and why it’s important here.

If you set your trial Routines to have a definite end-point (e.g. all components within a trial Routine will end after 5 seconds), you’ll notice that the colour of the Routine in your Flow changes from blue to green. This is your indication that the Routine is making use of non-slip timing.

If you can’t set your Routines to have a fixed duration (for example if a trial ends when a participant makes a response), it’s a good idea to insert a ‘Waiting for Scanner’ Routine at the start of every trial so that you know that each trial has been synced with your scanner’s trigger.

If there is a problem - We want to know!

If you have followed the steps above and are having an issue with triggers, please post details of this on the PsychoPy Forum.

We are constantly looking to update our documentation so that it’s easy for you to use PsychoPy in the way that you want to. Posting in our forum allows us to see what issues users are having, offer solutions, and to update our documentation to hopefully prevent those issues from occurring again!

9.1.7 Communicating with an Eyetracker

PsychoPy has components that allow you to connect and communicate with eyetrackers directly from Builder - without any code! These steps will guide you through how to set up, calibrate, and record from your eyetracker.

Step one: Select your plugin

If you are using a version of PsychoPy from 2022 or earlier, skip this step and go straight to step two

To use your eye tracker, you will need to install a plugin. PsychoPy supports many of the commonly used eye trackers. To find out whether yours is supported, follow these steps:

- Under the *Tools* tab, select *Plugin/packages Manager*
- Find your eye tracker in the plugins list and select *install*

If you would like to install a plugin from a file (e.g. a .whl file):

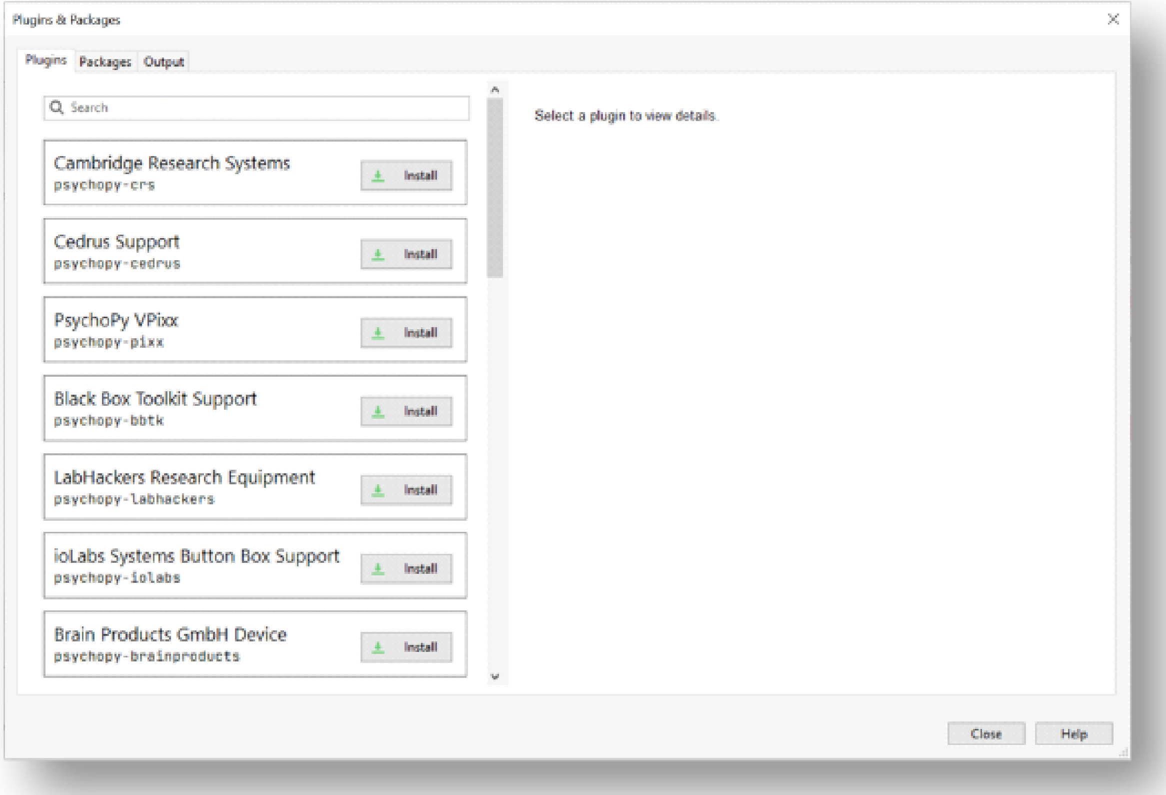
- In the Plugin and Packages Manager, select the *Packages* tab
- At the bottom left of the window, select whether you would like to install from a file or via the PIP terminal
- Select the relevant file or run a PIP install
- Restart PsychoPy to use the newly installed plugin

To find out more about plugins, visit our plugins website

Step two: Know Your Eyetracker

PsychoPy supports many of the commonly used eyetrackers, you can find out if yours is supported by following these steps:

- Click on the *Experiment Settings* icon (the one that looks like a cog, near the top left-hand side of the Builder window).
- Click on the *Eyetracking* tab:



Cambridge Research Systems
psychopy-crs

Cedrus Support
psychopy-cedrus

PsychoPy VPixx
psychopy-pixx

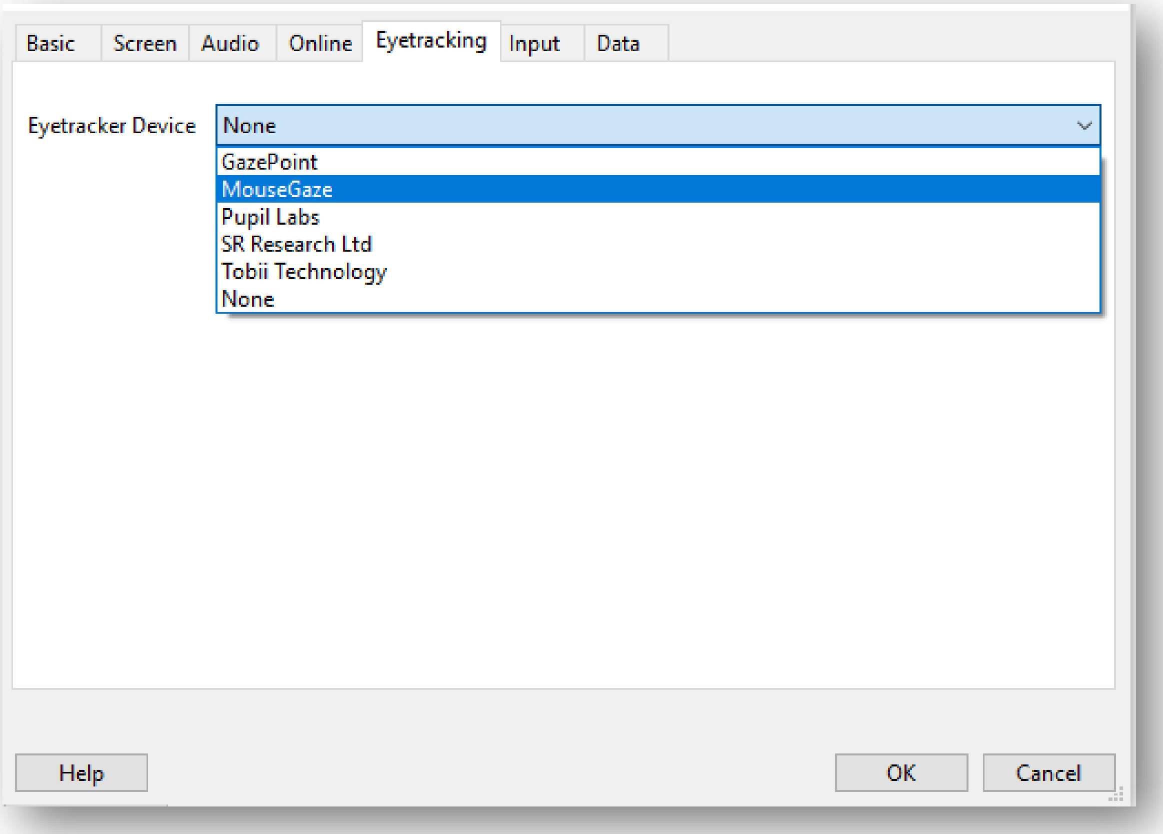
Black Box Toolkit Support
psychopy-bbtk

LabHackers Research Equipment
psychopy-labhackers

ioLabs Systems Button Box Support
psychopy-iolabs

Brain Products GmbH Device
psychopy-brainproducts

Select a plugin to view details.

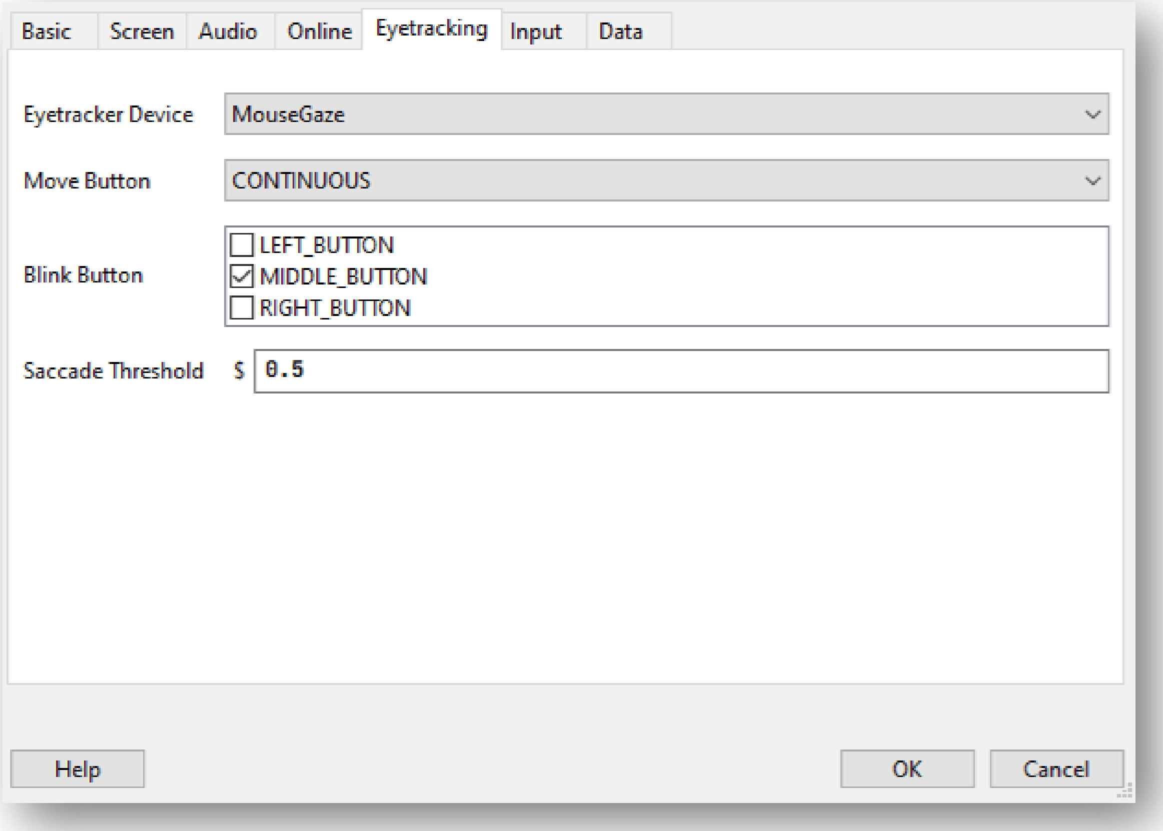


Eyetracker Device: None

- The *SR Research* option is also known as *Eyelink*, so if you have an Eyelink device this is the option to choose.
- When you've found your eyetracker, just select it and click *OK*.
- If you want to test out your eyetracking experiment but don't have an eyetracker with you, you can select *MouseGaze*. This will allow your mouse cursor to act as a gaze point on your screen, and so allow you to simulate eye movements without using an eyetracker. Then, when you're ready to use your eyetracker, you can just select it from the Experiment Settings and run your experiment in the same way.

Step three: Set up your Eyetracker

When you've selected your eyetracker from the drop-down menu, a set of options that are specific to that device will appear, such as the model and serial number of your device. Here we will follow through with the MouseGaze options:



Eyetracker Device: MouseGaze

Move Button: CONTINUOUS

Blink Button: □ LEFT_BUTTON ☑ MIDDLE_BUTTON □ RIGHT_BUTTON

Saccade Threshold: $ 0.5

- Choose which mouse button you'd like to use to simulate blinks by clicking on the boxes.
- The *Move Button* option allows you to select whether PsychoPy monitors your mouse movement continuously, or just when you press and hold one of the mouse buttons.
- The *Saccade Threshold* is the threshold, in degrees of visual angle, before a saccade is recorded.

EyeLink

When setting up your EyeLink you will first need to make sure you have the following set up:

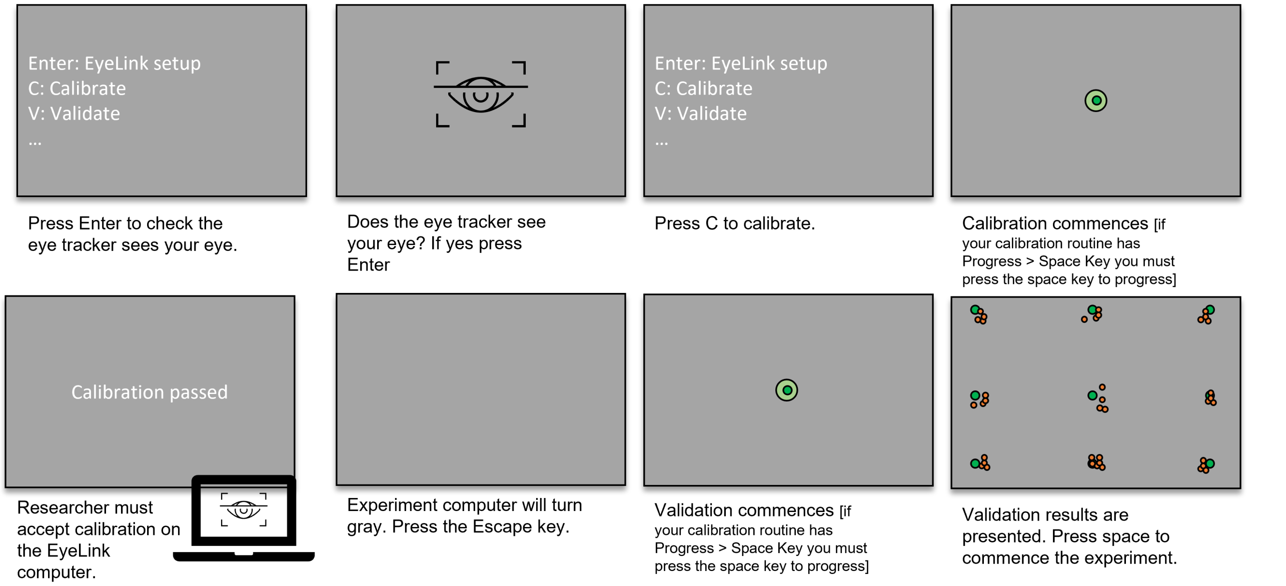
1. An "Experiment" computer (this is the computer the experiment is run on) - set the IP address of this computer to 100.1.1.2

2. A “Host” computer (this is the computer where the EyeLink software runs) - set the IP address of this computer to 100.1.1.1
3. In your PsychoPy Experiment Settings > Eyetracking ensure you have SR Research selected, in the IP address use 100.1.1.1 (the IP of the host computer).

Before any communication can happen between the eyetracker and your experiment, the two computers must be connected via an ethernet cable and you need to check the two devices can communicate with one another. You can check the connection by opening the command prompt/terminal on the experiment computer and typing `ping 100.1.1.1` if the connection is successful you will see that the pings are successfully returned. If you have trouble connecting at this phase you will want to trouble shoot by searching the returned error message.

Sometimes different eyetracking systems will have their own set of “screens” or “protocols” that they present. These are independant of what we can currently control from PsychoPy, which means that if you have made your experiment using MouseGaze, then move to the lab with the EyeLink and change the eyetracker to SR Research the instructions that you see at the start of the calibration may appear a little different to what you were expecting!

The general protocol you will see is shown below.



Enter: EyeLink setup
C: Calibrate
V: Validate
...

Press Enter to check the eye tracker sees your eye.

Does the eye tracker see your eye? If yes press Enter

Enter: EyeLink setup
C: Calibrate
V: Validate
...

Press C to calibrate.

Calibration commences [if your calibration routine has Progress > Space Key you must press the space key to progress]

Calibration passed

Researcher must accept calibration on the EyeLink computer.

Experiment computer will turn gray. Press the Escape key.

Validation commences [if your calibration routine has Progress > Space Key you must press the space key to progress]

Validation results are presented. Press space to commence the experiment.

Fig. 9.14: The set of screens that will appear on your experiment presentation screen during calibration/validation, and what to press when. If pressing the Escape key does not work, try pressing the ‘O’ key.

For further information about EyeLink installation and use, look at SR Research’s learning resources

Tobii

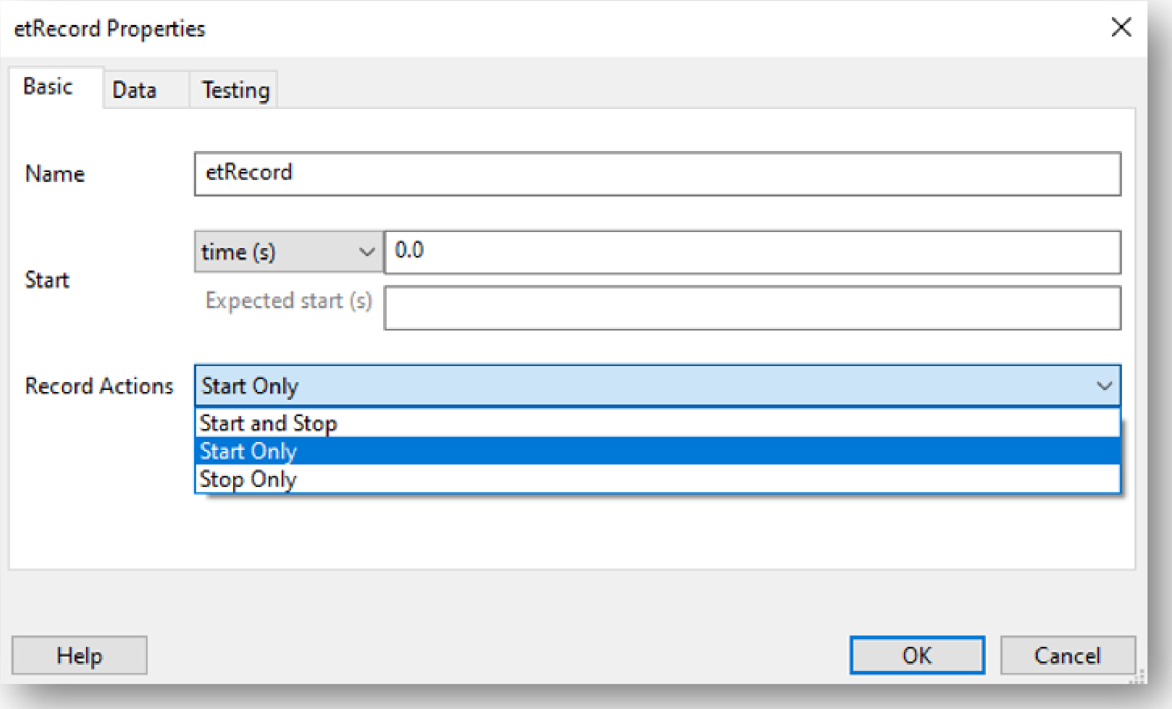
When plugging in your Tobii you will first want to download and install the free Eyetracker Manager software. We’de recommend conducting a quick calibration using that software so that you arre confident that your Tobii is connected and working, independantly of your PsychoPy experiment.

In PsychoPy, then click “Experiment Settings” > “Eyetracking”. Select “Tobii Technology” in the Eyetracker device drop down and write the name of your device (e.g. “Tobii Pro Nano”, “Tobii Pro Fusion”, “Tobii Pro Spectrum”, “Tobii TX300”). The other settings in this section are optional.

Step three: Add Eyetracker components to your Builder experiment

You can find the eyetracker components in the eyetracker component drop-down on the right-hand side of the Builder window.

- The first component to add is the **'Eyetracker Record'** component as this starts and stops the eyetracker recording. Usually, you would add this component to your instructions routine or something similar, so that your eyetracker is set off recording before your trials start, but you can add them in wherever makes sense for your experiment:



etRecord Properties

Name: etRecord

Start: time (s) 0.0

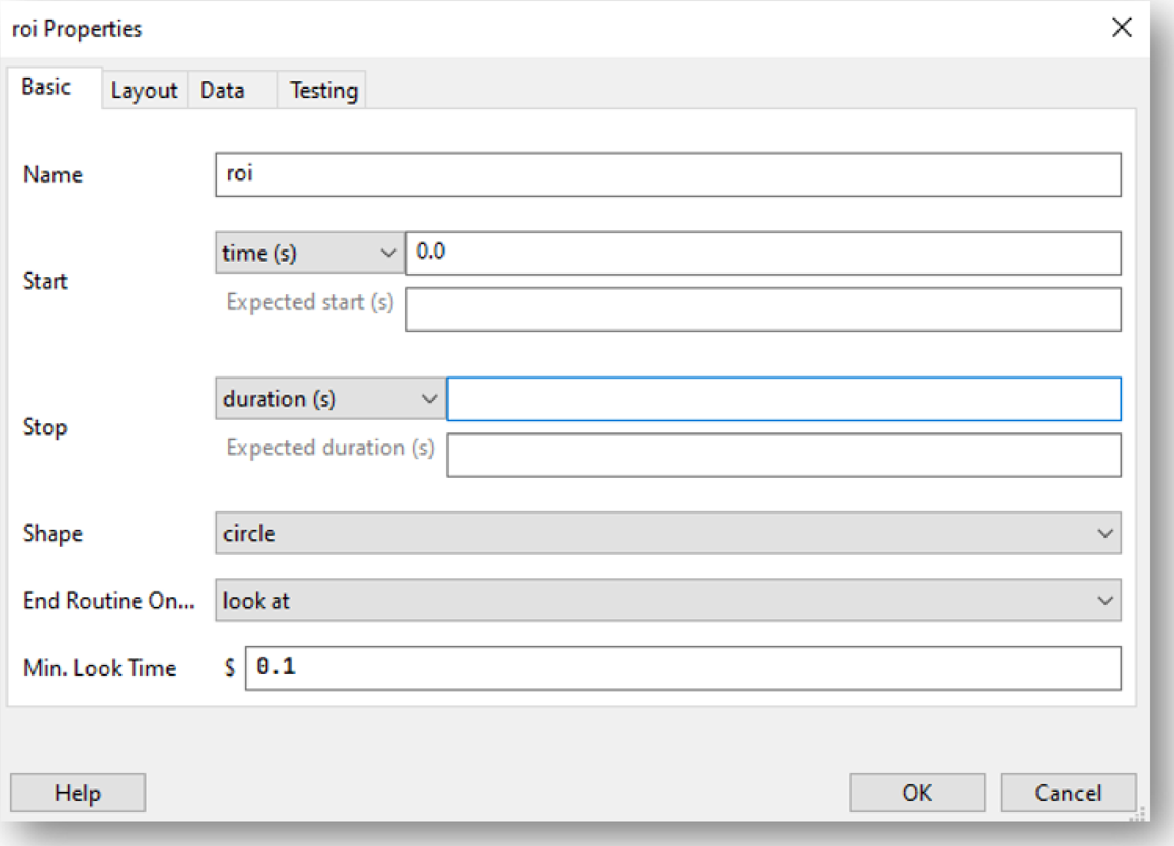
Expected start (s)

Record Actions: Start Only

Fig. 9.15: You can choose whether you want this component to just start your eyetracker recording, just stop the recording, or whether you want the component to start the recording and then stop it after a certain duration.

> **Note**
>
> If you've started the eyetracker recording at the start of your experiment, be sure to add in another eyetracker record component at the end of your experiment to stop the recording too!

- If you want to record information on gaze position, or you want your trial to move on when your participant has looked at or away from a target, you'll need to add in an **ROI component**. The ROI component has lots of options - you can choose what you want to happen when the participant looks at or away from a certain part of the screen, what shape your ROI is etc. All of which can also be defined in your conditions file, just like any other component. Choose the options that fit the needs of your experiment. Here, the component is set such that when a participant looks at a circular target for at least 0.1s (set by the min look time), the trial will end:
- On the *layout* tab of the ROI component, you set the position and size of the ROI in the same way as you would set the position of any visual component:
- It's also vitally important that you calibrate and validate your eyetracker. To do this, you will use two standalone



Name: roi

Start: time (s) 0.0

Expected start (s)

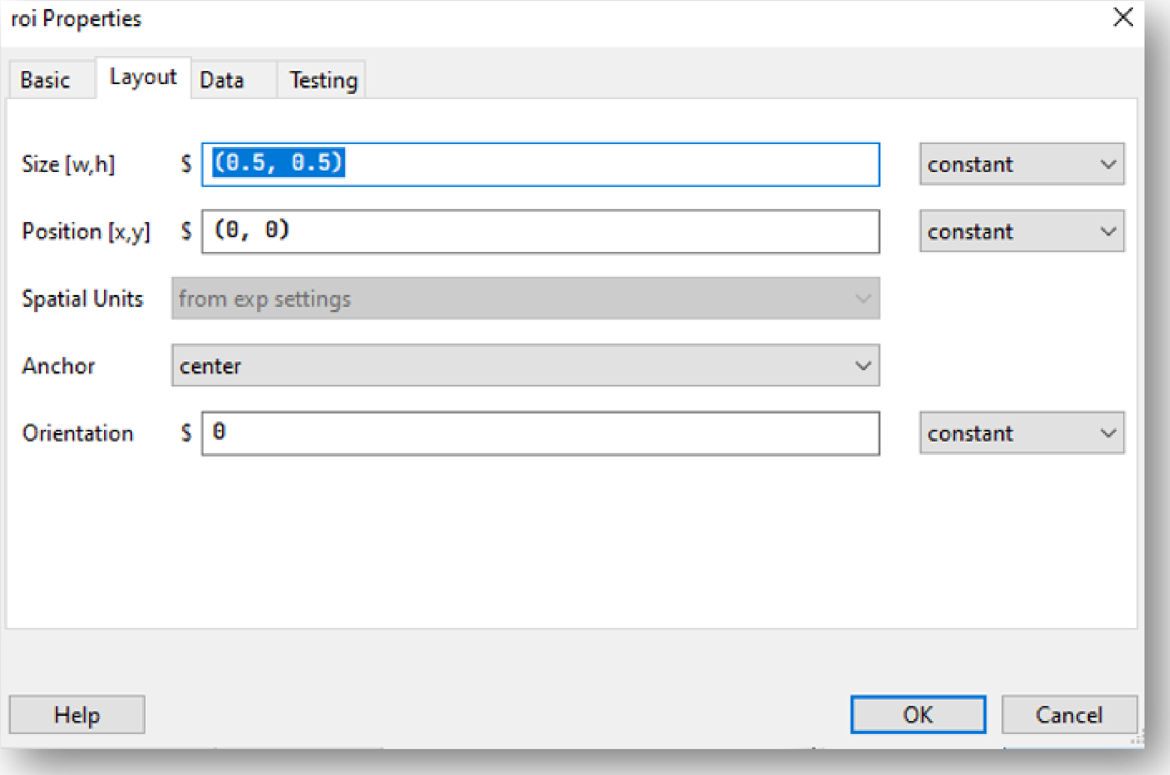
Stop: duration (s)

Expected duration (s)

Shape: circle

End Routine On...: look at

Min. Look Time: $ 0.1



Size [w,h] $ (0.5, 0.5) constant

Position [x,y] $ (0, 0) constant

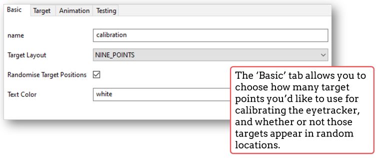
Spatial Units from exp settings

Anchor center

Orientation $ 0 constant

components: **Eyetracker calibrate** and **Eyetracker validate**.

- These are a little different from other components in that they form a routine all on their own. You'll need to add them in right at the start of your experiment Flow.
- The **Eyetracker calibrate** component has all of the options you would expect from an eyetracker calibration:



name calibration

Target Layout NINE_POINTS

Randomise Target Positions ☑

Text Color white

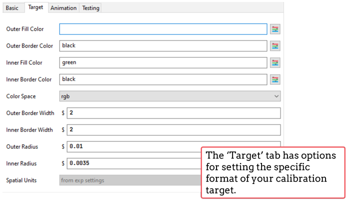
The 'Basic' tab allows you to choose how many target points you'd like to use for calibrating the eyetracker, and whether or not those targets appear in random locations.

Fig. 9.16: Set the basic properties of the calibration routine here.

- The **Eyetracker validate** component, you'll notice, is pretty much identical to the calibration component - that's because it will use the calibration information to present the same screen to the participant to cross-check the recorded gaze position with the calibrated gaze position.
- The Eyetracker validate component will then show the offset between the recorded and calibrated gaze positions. You'll want these to be as close as possible to ensure that your eyetracker is recording gaze accurately.

What about the data?

- The eyetracking data from the ROI will be saved in your usual data file. Extra columns are created and populated by PsychoPy, depending on what you've asked to record.



Outer Fill Color

Outer Border Color: black

Inner Fill Color: green

Inner Border Color: black

Color Space: rgb

Outer Border Width: $ 2

Inner Border Width: $ 2

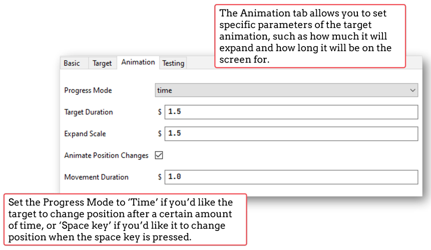
Outer Radius: $ 0.01

Inner Radius: $ 0.0035

Spatial Units: from exp settings

The 'Target' tab has options for setting the specific format of your calibration target.

Fig. 9.17: Set the properties of the target on this tab.



The Animation tab allows you to set specific parameters of the target animation, such as how much it will expand and how long it will be on the screen for.

Progress Mode: time

Target Duration: $ 1.5

Expand Scale: $ 1.5

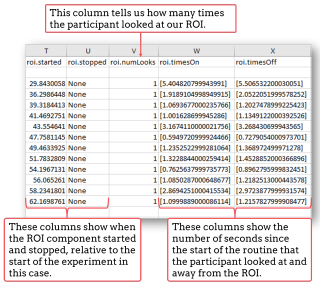
Animate Position Changes ☑

Movement Duration: $ 1.0

Set the Progress Mode to 'Time' if you'd like the target to change position after a certain amount of time, or 'Space key' if you'd like it to change position when the space key is pressed.

Fig. 9.18: This tab allows you to set the properties of the target animation.

- In the example below, the trial ended when a participant looked at a target on the screen. You can see what each column represents in the figure below:



This column tells us how many times the participant looked at our ROI.

T	U	V	W	X
roi.started	roi.stopped	roi.numLooks	roi.timesOn	roi.timesOff
29.8430058	None	1	[5.404820799943991]	[5.506532200030051]
36.2986448	None	1	[1.9189104998949915]	[2.0522051999578252]
39.3184413	None	1	[1.0693677000235766]	[1.2027478999225423]
41.4692751	None	1	[1.001628699945286]	[1.1349122000392526]
43.554641	None	1	[3.1674110000021756]	[3.268430699943565]
47.7581145	None	1	[0.5949720999924466]	[0.7279054000973701]
49.4633925	None	1	[1.2352522999281064]	[1.368972499971278]
51.7832809	None	1	[1.3228844000259414]	[1.4528852000366896]
54.1967131	None	1	[0.7625637999735773]	[0.8962795999832451]
56.065261	None	1	[1.0850287000648677]	[1.2182513000443578]
58.2341801	None	1	[2.8694251000415534]	[2.9723877999931574]
62.1698761	None	1	[1.0999889000086114]	[1.2157827999908477]

These columns show when the ROI component started and stopped, relative to the start of the experiment in this case.

These columns show the number of seconds since the start of the routine that the participant looked at and away from the ROI.

Fig. 9.19: The data output will vary according to what you've asked PsychoPy to record about gaze.

- PsychoPy also provides the option to save your eyetracking data as a hdf5 file, which is particularly useful if you are recording a large amount of eyetracking data, such as gaze position on every frame for example.
- To save eyetracking data as a hdf5 file, just click on the Experiment Settings icon, and in the 'Data' tab check the box next to 'Save hdf5 file'. Hdf5 files can be inspected using a free software such as hdfView or, alternatively, you can extract data from your hdf5 files using the python h5py library. For example, the code below could be used to write data stored in a hdf5 file to a csv using a combination of h5py and pandas:

```
import h5py
import pandas as pd

filename = "data/becca2_becca_track_2022-12-12_17h36.27.977.hdf5"
id = filename.split("/")[1].split("_")[0]
with h5py.File(filename, "r") as f:

    # get the list of eyetracker measures available in the hdf5
    eyetracker_measures = list(f['data_collection']['events']['eyetracker'])

    for measure in eyetracker_measures:
```

(continues on next page)

(continued from previous page)

```
        print('Extracting events of type: ', measure)
        data_collection = list(f['data_collection']['events']['eyetracker
↪'][measure])
        if len(data_collection)>0:
            column_headers = data_collection[0].dtype.descr
            cols = []
            data_dict = {}
            for ch in column_headers:
                cols.append(ch[0])
                data_dict[ch[0]] = []

            for row in data_collection:
                for i, col in enumerate(cols):
                    data_dict[col].append(row[i])
            pd_data = pd.DataFrame.from_dict(data_dict)
            pd_data.to_csv(id+'_'+measure+'.csv', index = False)
        else:
            print('No data for type', measure, ' moving on')
```

Finally, you could plot data from the above, for instance, as a heatmap:

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

filename = '2_BinocularEyeSampleEvent.csv'

# read as pandas dataframe
data = pd.read_csv(filename)

# convert pandas arrays to no arrays
x = data['left_gaze_x'].to_numpy()
y = data['left_gaze_y'].to_numpy()

# remove nan values
x = x[~np.isnan(x)]
y = y[~np.isnan(y)]

# plot x and y values as a heat map
heatmap, xedges, yedges = np.histogram2d(x, y, bins=50)
extent = [xedges[0], xedges[-1], yedges[0], yedges[-1]]

# show the plot
plt.clf()
plt.imshow(heatmap.T, extent=extent, origin='lower')
plt.show()
```

If there is a problem - We want to know!

If you have followed the steps above and are having an issue, please post details of this on the PsychoPy Forum.

We are constantly looking to update our documentation so that it's easy for you to use PsychoPy in the way that you want to. Posting in our forum allows us to see what issues users are having, offer solutions, and to update our documentation to hopefully prevent those issues from occurring again!

please also see this video tutorial.

Note

If you'd like to use a *Parallel Port* to **record** responses (for example from a button box) please read this excellent thread from our Discourse Forum user jtseng.

9.2 Communicating with an eye-tracker

- *Communicating with an Eyetracker*

9.3 Communicating with other devices

- *Communicating with fMRI*
- *Recording information from an Arduino via serial port*
- To communicate with fNIRS, please watch this super-clear video tutorial from NIRx.

CHAPTER

TEN

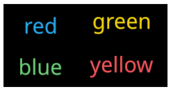
HOW-TO GUIDES

If you're wondering how to do something you're in the right place!

Need more help? See our Resources page!

10.1 How do I build experiments?

See the guides below to learn how to make some classic experiments.



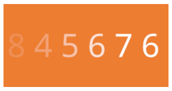
Stroop Task   Report the color of words while ignoring their content.

A great starting point for learning to use PsychoPy!

stroop

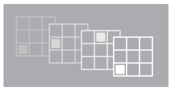
Lexical Decision Task   Decide if a word is a "real" word or not.

lexical_decision



Digit Span Task   A basic test of working memory.

digit_span



N-Back Task   Decide if a stimulus is the same as the stimulus n trials back.

n_back

10.2 How do I extend experiments with code?

10.2.1 Builder - providing feedback

If you're using the Builder then the way to provide feedback is with a *Code Component* to generate an appropriate message (and then a *Text Component* to present that message). PsychoPy will be keeping track of various aspects of the stimuli and responses for you throughout the experiment and the key is knowing where to find those.

The following examples assume you have a *Loop* called *trials*, containing a *Routine* with a *Keyboard Component* called *key_resp*. Obviously these need to be adapted in the code below to fit your experiment.

Note

The following generate strings use python 'formatted strings'. These are very powerful and flexible but a little strange when you aren't used to them (they contain odd characters like %.2f). See *Generating formatted strings* for more info.

Feedback after a trial

This is actually demonstrated in the demo, *ExtendedStroop* (in the Builder>demos menu, unpack the demos and then look in the menu again. tada!)

If you have a Keyboard Component called *key_resp* then, after every trial you will have the following variables:

```
key_resp.keys # A python list of keys pressed
key_resp.rt # The time to the first key press
key_resp.corr # None, 0 or 1, if you are using 'store correct'
```

To create your *msg*, insert the following into the 'start experiment` section of the *Code Component*:

```
msg='doh!'# If this comes up we forgot to update the msg!
```

and then insert the following into the *Begin Routine* section (this will get run every repeat of the routine):

```
if not key_resp.keys :
    msg = "Failed to respond"
elif key_resp.corr: # Stored on last run routine
    msg = "Correct! RT=%.3f" %(key_resp.rt)
else:
    msg="Oops! That was wrong"
```

Feedback after a block

In this case the feedback routine would need to come after the loop (the block of trials) and the message needs to use the stored data from the loop rather than the *key_resp* directly. Accessing the data from a loop is not well documented but totally possible.

In this case, to get all the keys pressed in a numpy array:

```
trials.data['key_resp.keys'] # numpy array with size=[ntrials,ntypes]
```

If you used the 'Store Correct' feature of the Keyboard Component (and told psychopy what the correct answer was) you will also have a variable:

```
# numpy array storing whether each response was correct (1) or not (0)
trials.data['key_resp.corr']
```

So, to create your *msg*, insert the following into the 'start experiment` section of the *Code Component*:

```
msg='doh!'# If this comes up we forgot to update the msg!
```

and then insert the following into the *Begin Routine* section (this will get run every repeat of the routine):

```
nCorr = trials.data['key_resp.corr'].sum() # .std(), .mean() also available
meanRt = trials.data['key_resp.rt'].mean()
msg = "You got %i trials correct (rt=%.2f)" %(nCorr,meanRt)
```

Draw your message to the screen

Using one of the above methods to generate your *msg* in a *Code Component*, you then need to present it to the participant by adding a *Text Component* to your *feedback* Routine and setting its text to *$msg*.

> ⚠ **Warning**
>
> The Text Component needs to be below the Code Component in the Routine (because it needs to be updated after the code has been run) and it needs to *set every repeat*.

Youtube tutorial

- Trial by trial accuracy feedback
- Trial by trial reaction time feedback
- Feedback for typed responses

10.2.2 Builder - terminating a loop

People often want to terminate their *Loops* before they reach the designated number of trials based on subjects' responses. For example, you might want to use a Loop to repeat a sequence of images that you want to continue until a key is pressed, or use it to continue a training period, until a criterion performance is reached.

To do this you need a *Code Component* inserted into your *routine*. All loops have an attribute called *finished* which is set to *True* or *False* (in Python these are really just other names for *1* and *0*). This *finished* property gets checked on each pass through the loop. So the key piece of code to end a loop called *trials* is simply:

```
trials.finished=True #or trials.finished=1 if you prefer
```

Of course you need to check the condition for that with some form of *if* statement.

Example 1: You have a change-blindness study in which a pair of images flashes on and off, with intervening blanks, in a loop called *presentationLoop*. You record the key press of the subject with a *Keyboard Component* called *resp1*. Using the 'ForceEndTrial' parameter of *resp1* you can end the current cycle of the loop but to end the loop itself you would need a *Code Component*. Insert the following two lines in the *End Routine* parameter for the Code Component, which will test whether more than zero keys have been pressed:

```
if resp1.keys is not None and len(resp1.keys)>0 :
        trials.finished=1
```

or:

```
if resp1.keys :
        presentationLoop.finished=1
```

Example 2: Sometimes you may have more possible trials than you can actually display. By default, a loop will present all possible trials (nReps * length-of-list). If you only want to present the first 10 of all possible trials, you can use a code component to count how many have been shown, and then finish the loop after doing 10.

This example assumes that your loop is named 'trials'. You need to add two things, the first to initialize the count, and the second to update and check it.

Begin Experiment:

```
myCount = 0
```

Begin Routine:

```
myCount = myCount + 1
if myCount > 10:
    trials.finished = True
```

Note

In Python there is no *end* to finish an *if* statement. The content of the *if* or of a for-loop is determined by the indentation of the lines. In the above example only one line was indented so that one line will be executed if the statement evaluates to *True*.

10.3 How do I…

10.3.1 Animation

General question: How can I animate something?

Conceptually, animation just means that you vary some aspect of the stimulus over time. So the key idea is to draw something slightly different on each frame. This is how movies work, and the same principle can be used to create scrolling text, or fade-in / fade-out effects, and the like.

(copied & pasted from the email list; see the list for people's names and a working script.)

10.3.2 Scrolling text

Key idea: Vary the **position** of the stimulus across frames.

Question: How can I produce scrolling text (like html's <marquee behavior = "scroll" > directive)?

Answer: PsychoPy has animation capabilities built-in (it can even produce and export movies itself (e.g. if you want to show your stimuli in presentations)). But here you just want to animate stimuli directly.

e.g. create a text stimulus. In the 'pos' (position) field, type:

> [frameN, 0]

and select "set every frame" in the popup button next to that field.

Push the Run button and your text will move from left to right, at one pixel per screen refresh, but stay at a fixed y-coordinate. In essence, you can enter an arbitrary formula in the position field and the stimulus will be-redrawn at a new position on each frame. frameN here refers to the number of frames shown so far, and you can extend the formula to produce what you need.

You might find performance issues (jittering motion) if you try to render a lot of text in one go, in which case you may have to switch to using images of text.

I wanted my text to scroll from right to left. So if you keep your eyes in the middle of the screen the next word to read would come from the right (as if you were actually reading text). The original formula posted above scrolls the other way. So, you have to put a negative sign in front of the formula for it to scroll the other way. You have to change the units to pixel. Also, you have to make sure you have an end time set, otherwise it just flickers. I also set my letter height to 100 pixels. The other problem I had was that I wanted the text to start blank and scroll into the screen. So, I wrote

> [2000-frameN, 0]

and this worked really well.

10.3.3 Fade-in / fade-out effects

Key idea: vary the **opacity** of the stimulus over frames.

Question: I'd like to present an image with the image appearing progressively and disappearing progressively too. How to do that?

Answer: The Patch stimulus has an opacity field. Set the button next to it to be "set every frame" so that its value can be changed progressively, and enter an equation in the box that does what you want.

e.g. if your screen refresh rate is 60 Hz, then entering:

> frameN/120

would cycle the opacity linearly from 0 to 1.0 over 2s (it will then continue incrementing but it doesn't seem to matter if the value exceeds 1.0).

Using a code component might allow you to do more sophisticated things (e.g. fade in for a while, hold it, then fade out). Or more simply, you just create multiple successive Patch stimulus components, each with a different equation or value in the opacity field depending on their place in the timeline.

10.3.4 Typing effects

Key idea: vary the **onset/offset** of stimulus

Question: I'd like to present my text using a typing effect.

Answer: Please click here to watch youtube tutorial Typing effect without code

10.3.5 Generating formatted strings

A formatted string is a variable which has been converted into a string (text). In python the specifics of how this is done is determined by what kind of variable you want to print.

Example 1: You have an experiment which generates a string variable called *text*. You want to insert this variable into a string so you can print it. This would be achieved with the following code:

```
message = 'The result is %s' %(text)
```

This will produce a variable *message* which if used in a text object would print the phrase 'The result is' followed by the variable *text*. In this instance %s is used as the variable being entered is a string. This is a marker which tells the script where the variable should be entered. *%text* tells the script which variable should be entered there.

Multiple formatted strings (of potentially different types) can be entered into one string object:

```
longMessage = 'Well done %s that took %0.3f seconds' %(info['name'], time)
```

Some of the handy formatted string types:

```
>>> x=5
>>> x1=5124
>>> z='someText'
>>> 'show %s' %(z)
'show someText'
>>> '%0.1f' %(x)   #will show as a float to one decimal place
'5.0'
>>> '%3i' %(x) #an integer, at least 3 chars wide, padded with spaces
'  5'
>>> '%03i' %(x) #as above but pad with zeros (good for participant numbers)
'005'
```

See the python documentation for a more complete list.

10.3.6 How do I run experiments written with older versions of ?

If you perform experiments on computers shared by different research groups (e. g. at a shared experimental facility), it's possible that their experiments and yours are written using different versions of . Or maybe you yourself have some older and some newer experiments. In such a situation, it's important to ensure that the right version of is used for the right experiment.

In PsychoPy standalone, there is an easy-to-use system for controlling what version of is used.

Version control using Builder View

1. Open up the experiment file (a '.psyexp' file) that contains the experiment you want to use.
2. Go to experiment settings by clicking the icon with a cogwheel.
3. Under the "Basic" settings tab, there is an option named "Use PsychoPy version". Set it to the PsychoPy version you want to emulate.
4. Click "OK" to save the settings.
5. Run the experiment by clicking the green 'run' button.

Version control using Coder View

1. Open up the script file (a '.py' file) that contains the experiment you want to use.
2. Add `import psychopy` at the top of your script, **above** all other import statements.

3. Add the function call `psychopy.useVersion('<version_no>')` directly below `import psychopy`, but still **above** the other import statements. Here's an example:

```
import psychopy
psychopy.useVersion('2021.1.0')
from psychopy import visual, core, event
# the rest of your script follows
```

4. Run the script.

NOTE: Internet connection needed (the first time)

You need to have a working internet connection the first time that you run an experiment using a particular version (e. g. 1.90.2) on a computer, so that can download some info about the version for you. Once you've used a version once, your computer has saved the information it needs for emulating it. This means that after the first time, you don't need an internet connection if you run the same or another experiment using that version (e. g. 1.90.2).

Compatibility

Using either of the above methods, you should often only need the latest version of standalone to run older experiments. However, if you have an experiment designed with a very old version of (say, version 1.77.01) you might have to install an older version of standalone . Since these things change over time, you probably want to search for help in the PsychoPy forums.

10.3.7 Coder - interleave staircases

Often psychophysicists using staircase procedures want to interleave multiple staircases, either with different start points, or for different conditions.

There is now a class, `psychopy.data.MultiStairHandler` to allow simple access to interleaved staircases of either basic or QUEST types. That can also be used from the *Loops* in the *Builder*. The following method allows the same to be created in your own code, for greater options.

The method works by nesting a pair of loops, one to loop through the number of trials and another to loop across the staircases. The staircases can be shuffled between trials, so that they do not simply alternate.

Note

Note the need to create a *copy* of the info. If you simply do *thisInfo=info* then all your staircases will end up pointing to the same object, and when you change the info in the final one, you will be changing it for all.

```
from psychopy import visual, core, data, event
from numpy.random import shuffle
import copy, time #from the std python libs

#create some info to store with the data
info={}
info['startPoints']=[1.5,3,6]
info['nTrials']=10
info['observer']='jwp'

win=visual.Window([400,400])
#---------------------
#create the stimuli
#---------------------

#create staircases
stairs=[]
for thisStart in info['startPoints']:
    #we need a COPY of the info for each staircase
    #(or the changes here will be made to all the other staircases)
    thisInfo = copy.copy(info)
    #now add any specific info for this staircase
    thisInfo['thisStart']=thisStart #we might want to keep track of this
    thisStair = data.StairHandler(startVal=thisStart,
        extraInfo=thisInfo,
        nTrials=50, nUp=1, nDown=3,
        minVal = 0.5, maxVal=8,
        stepSizes=[4,4,2,2,1,1])
    stairs.append(thisStair)

for trialN in range(info['nTrials']):
    shuffle(stairs) #this shuffles 'in place' (ie stairs itself is changed, nothing␣
↪returned)
    #then loop through our randomised order of staircases for this repeat
    for thisStair in stairs:
        thisIntensity = next(thisStair)
        print('start=%.2f, current=%.4f' %(thisStair.extraInfo['thisStart'],␣
↪thisIntensity))

        #---------------------
```

(continues on next page)

(continued from previous page)

```
        #run your trial and get an input
        #----------------------
        keys = event.waitKeys() #(we can simulate by pushing left for 'correct')
        if 'left' in keys: wasCorrect=True
        else: wasCorrect = False

        thisStair.addData(wasCorrect) #so that the staircase adjusts itself

    #this trial (of all staircases) has finished
#all trials finished

#save data (separate pickle and txt files for each staircase)
dateStr = time.strftime("%b_%d_%H%M", time.localtime())#add the current time
for thisStair in stairs:
    #create a filename based on the subject and start value
    filename = "%s start%.2f %s" %(thisStair.extraInfo['observer'], thisStair.extraInfo[
↪'thisStart'], dateStr)
    thisStair.saveAsPickle(filename)
    thisStair.saveAsText(filename)
```

10.3.8 Coder - show images

Sometimes we want to use PsychoPy to show images — either read from an image file (such as *.png*) or from a Numpy array. We can use either the *psychopy.visual.ImageStim* or *psychopy.visual.GratingStim* to achieve this. However, some of the nuances of actually getting the correct image to screen can be difficult to figure out.

This recipe demonstrates (1) a way to use *psychopy.visual.ImageStim* to read an image from disc and show it, and (2) using *psychopy.visual.ImageStim* to show a numpy array as an image.

When showing and converting images, you need to be careful about data types and channels. The scikit-image docs on this are quite good.

```
from pathlib import Path

import numpy as np
from PIL import Image
from psychopy import core, event, visual

# ---------------------
# Setup window
# ---------------------
win = visual.Window(
    (900, 900),
    screen=0,
    units="pix",
    allowGUI=True,
    fullscr=False,
)
# ---------------------
# Example 1: load a stimulus from disk
# ---------------------

# assume we're running from the root psychopy repo.
```

(continues on next page)

(continued from previous page)

```python
# you could replace with path to any image:
path_to_image_file = Path() / "PsychoPy2_screenshot.png"

# simply pass the image path to ImageStim to load and display:
image_stim = visual.ImageStim(win, image=path_to_image_file)
text_stim = visual.TextStim(
    win,
    text="Showing image from file",
    pos=(0.0, 0.8),
    units="norm",
    height=0.05,
    wrapWidth=0.8,
)

image_stim.draw()
text_stim.draw()
win.flip()
event.waitKeys()  # press space to continue

# ---------------------
# Example 2: convert image to numpy array
#
# Perhaps you want to convert an image to numpy, do some things to it,
# and then display. Here I use the Python Imaging Library for image loading,
# and a conversion function from skimage. PsychoPy has an internal
# "image2array" function but this only handles single-layer (i.e. intensity) images.
#
# ---------------------

pil_image = Image.open(path_to_image_file)
image_np = np.array(
    pil_image, order="C"
)  # convert to numpy array with shape width, height, channels
image_np = (
    image_np.astype(np.float) / 255.0
)  # convert to float in 0--1 range, assuming image is 8-bit uint.

# Note this float conversion is "quick and dirty" and will not
# fix potential out-of-range problems if you're going
# straight from a numpy array. See the img_as_float
# function of scikit image for a more careful conversion.

# flip image (row-axis upside down so we need to reverse it):
image_np = np.flip(image_np, axis=0)
image_stim = visual.ImageStim(
    win,
    image=image_np,
    units="pix",
    size=(
        image_np.shape[1],
        image_np.shape[0],
    ),  # here's a gotcha: need to pass the size (x, y) explicitly.
```

(continues on next page)

(continued from previous page)

```
    colorSpace="rgb1",  # img_as_float converts to 0:1 range, whereas PsychoPy defaults␣
↪to -1:1.
)
text_stim.text = "Showing image from numpy array"
image_stim.draw()
text_stim.draw()
win.flip()
event.waitKeys()  # press space to continue

win.close()
core.quit()
```

10.3.9 Making isoluminant stimuli

From the mailing list (see there for names, etc):

Q1: How can I create colours (RGB) that are isoluminant?

A1: The easiest way to create isoluminant stimuli (or control the luminance content) is to create the stimuli in DKL space and then convert them into RGB space for presentation on the monitor.

More details on DKL space can be found in the section about *Color spaces* and conversions between DKL and RGB can be found in the API reference for `psychopy.misc`

Q2: There's a difference in luminance between my stimuli. How could I correct for that?

I'm running an experiment where I manipulate color chromatic saturation, keeping luminance constant. I've coded the colors (red and blue) in rgb255 for 6 saturation values (10%, 20%, 30%, 40%, 50%, 60%, 90%) using a conversion from HSL to RGB color space.

Note that we don't possess spectrophotometers such as PR650 in our lab to calibrate each color gun. I've calibrated the gamma of my monitor psychophysically. Gamma was set to 1.7 (threshold) for gamm(lum), gamma(R), gamma(G), gamma(B). Then I've measured the luminance of each stimuli with a Brontes colorimeter. But there's a difference in luminance between my stimuli. How could I correct for that?

A2: Without a spectroradiometer you won't be able to use the color spaces like DKL which are designed to help this sort of thing.

If you don't care about using a specific colour space though you should be able to deduce a series of isoluminant colors manually, because the luminance outputs from each gun should sum linearly. e.g. on my monitor:

```
maxR=46cd/m2
maxG=114
maxB=15
```

(note that green is nearly always brightest)

So I could make a 15cd/m2 stimulus using various appropriate fractions of those max values (requires that the screen is genuinely gamma-corrected):

```
R=0, G=0, B=255
R=255*15/46, G=0, B=0
R=255*7.5/46, G=255*15/114, B=0
```

Note that, if you want a pure fully-saturated blue, then you're limited by the monitor to how bright you can make your stimulus. If you want brighter colours your blue will need to include some of the other guns (similarly for green if you want to go above the max luminance for that gun).

A2.1. You should also consider that even if you set appropriate RGB values to display your pairs of chromatic stimuli at the same luminance that they might still appear different, particularly between observers (and even if your light measurement device says the luminance is the same, and regardless of the colour space you want to work in). To make the pairs perceptually isoluminant, each observer should really determine their own isoluminant point. You can do this with the minimum motion technique or with heterochromatic flicker photometry.

10.3.10 Installing PsychoPy in a classroom (administrators)

For running PsychoPy in a classroom environment it is probably preferable to have a 'partial' network installation. The PsychoPy library features frequent new releases, including bug fixes and you want to be able to update machines with these new releases. But PsychoPy depends on many other python libraries (over 200Mb in total) that tend not to change so rapidly, or at least not in ways critical to the running of experiments. If you install the whole PsychoPy application on the network then all of this data has to pass backwards and forwards, and starting the app will take even longer than normal.

The basic aim of this document is to get to a state whereby;

- Python and the major dependencies of PsychoPy are installed on the local machine (probably a disk image to be copied across your lab computers)
- PsychoPy itself (only ~2Mb) is installed in a network location where it can be updated easily by the administrator
- a file is created in the installation that provides the path to the network drive location
- Start-Menu shortcuts need to be set to point to the local Python but the remote PsychoPy application launcher

Once this is done, the vast majority of updates can be performed simply by replacing the PsychoPy library on the network drive.

1. Install dependencies locally

Download the latest version of the Standalone PsychoPy distribution, and run as administrator. This will install a copy of Python and many dependencies to a default location of

> *C:\Program Files\PsychoPy2\*

2. Move the PsychoPy to the network

You need a network location that is going to be available, with read-only access, to all users on your machines. You will find all the contents of PsychoPy itself at something like this (version dependent obviously):

> *C:\Program Files\PsychoPy2\Lib\site-packages\PsychoPy-1.70.00-py2.6.egg*

Move that entire folder to your network location and call it psychopyLib (or similar, getting rid of the version-specific part of the name). Now the following should be a valid path:

> *<NETWORK_LOC>\psychopyLib\psychopy*

3. Update the Python path

The Python installation (in C:\Program Files\PsychoPy2) needs to know about the network location. If Python finds a text file with extension *.pth* anywhere on its existing path then it will add to the path any valid paths it finds in the file. So create a text file that has one line in it:

> *<NETWORK_LOC>\psychopyLib*

You can test if this has worked. Go to *C:\Program Files\PsychoPy2* and double-click on python.exe. You should get a Python terminal window come up. Now try:

```
>>> import psychopy
```

If psychopy is not found on the path then there will be an import error. Try adjusting the .pth file, restarting python.exe and importing again.

4. Update the Start Menu

The shortcut in the Windows Start Menu will still be pointing to the local (now non-existent) PsychoPy library. Right-click it to change properties and set the shortcut to point to something like:

```
"C:\Program Files\PsychoPy2\pythonw.exe" "<NETWORK_LOC>\psychopyLib\psychopy\app\
↪psychopyApp.py"
```

You probably spotted from this that the PsychoPy app is simply a Python script. You may want to update the file associations too, so that *.psyexp* and *.py* are opened with:

```
"C:\Program Files\PsychoPy2\pythonw.exe" "<NETWORK_LOC>\psychopyLib\psychopy\app\
↪psychopyApp.py" "%1"
```

Lastly, to make the shortcut look pretty, you might want to update the icon too. Set the icon's location to:

```
"<NETWORK_LOC>\psychopyLib\psychopy\app\Resources\psychopy.ico"
```

5. Updating to a new version

Fetch the latest .zip release. Unpack it and replace the contents of *<NETWORK_LOC>\psychopyLib\* with the contents of the zip file.

10.3.11 Adding external modules to Standalone PsychoPy

You might find that you want to add some additional Python module/package to your Standalone version of PsychoPy. To do this you need to:

- download a copy of the package (make sure it's for Python 2.7 on your particular platform)
- unzip/open it into a folder
- add that folder to the path of PsychoPy by one of the methods below

Avoid adding the entire path (e.g. the site-packages folder) of separate installation of Python, because that may contain conflicting copies of modules that PsychoPy is also providing.

Using preferences

As of version 1.70.00 you can do this using the PsychoPy preferences/general. There you will find a preference for *paths* which can be set to a list of strings e.g. *['/Users/jwp/code', '~/code/thirdParty']*

These only get added to the Python path when you import psychopy (or one of the psychopy packages) in your script.

Adding a .pth file

An alternative is to add a file into the site-packages folder of your application. This file should be pure text and have the extension .pth to indicate to Python that it adds to the path.

On win32 the site-packages folder will be something like:

C:/Program Files/PsychoPy2/lib/site-packages

On macOS you need to right-click the application icon, select 'Show Package Contents' and then navigate down to Contents/Resources/lib/pythonX.X. Put your .pth file here, next to the various libraries.

The advantage of this method is that you don't need to do the import psychopy step. The downside is that when you update PsychoPy to a new major release you'll need to repeat this step (patch updates won't affect it though).

10.3.12 Building an application from your script

A lot of people ask how they can build a standalone application from their Python script. Usually this is because they have a collaborator and want to just send them the experiment.

In general this is not advisable - the resulting bundle of files (single file on macOS) will be on the order of 100Mb and will not provide the end user with any of the options that they might need to control the task (for example, Monitor Center won't be provided so they can't to calibrate their monitor). A better approach in general is to get your collaborator to install the Standalone PsychoPy on their own machine, open your script and press run. (You don't send a copy of Microsoft Word when you send someone a document - you expect the reader to install it themself and open the document).

Nonetheless, it is technically possible to create exe files on Windows, and Ricky Savjani (savjani at bcm.edu) has kindly provided the following instructions for how to do it. A similar process might be possible on macOS using py2app - if you've done that then feel free to contribute the necessary script or instructions.

Using py2exe to build an executable

Instructions:

1. Download and install py2exe (http://www.py2exe.org/)
2. Develop your PsychoPy script as normal
3. Copy this setup.py file into the same directory as your script
4. Change the Name of progName variable in this file to the Name of your desired executable program name
5. **Use cmd (or bash, terminal, etc.) and run the following in the directory of your the two files:**
   python setup.py py2exe
6. Open the 'dist' directory and run your executable

A example setup.py script:

```
#   Created 8-09-2011
#   Ricky Savjani
#   (savjani at bcm.edu)

#import necessary packages
from distutils.core import setup
import os, matplotlib
import py2exe

#the name of your .exe file
progName = 'MultipleSchizophrenia.py'

#Initialize Holder Files
preference_files = []
app_files = []
my_data_files=matplotlib.get_py2exe_datafiles()

#define which files you want to copy for data_files
for files in os.listdir('C:\\Program Files\\PsychoPy2\\Lib\\site-packages\\PsychoPy-1.65.
↪00-py2.6.egg\\psychopy\\preferences\\'):
```

(continues on next page)

(continued from previous page)

```
    f1 = 'C:\\Program Files\\PsychoPy2\\Lib\\site-packages\\PsychoPy-1.65.00-py2.6.egg\\psychopy\\preferences\\' + files
    preference_files.append(f1)

#if you might need to import the app files
#for files in os.listdir('C:\\Program Files\\PsychoPy2\\Lib\\site-packages\\PsychoPy-1.65.00-py2.6.egg\\psychopy\\app\\'):
#    f1 = 'C:\\Program Files\\PsychoPy2\\Lib\\site-packages\\PsychoPy-1.65.00-py2.6.egg\\psychopy\\app\\' + files
#    app_files.append(f1)

#all_files = [("psychopy\\preferences", preference_files),("psychopy\\app", app_files), my_data_files[0]]

#combine the files
all_files = [("psychopy\\preferences", preference_files), my_data_files[0]]

#define the setup
setup(
                console=[progName],
                data_files = all_files,
                options = {
                    "py2exe":{
                        "skip_archive": True,
                        "optimize": 2
                    }
                }
)
```

10.3.13 Adding a web-cam

From the mailing list (see there for names, etc):

"I spent some time today trying to get a webcam feed into my psychopy proj, inside my visual.window. The solution involved using the opencv module, capturing the image, converting that to PIL, and then feeding the PIL into a SimpleImageStim and looping and win.flipping. Also, to avoid looking like an Avatar in my case, you will have to change the default decoder used in PIL fromstring to utilize BGR instead of RGB in the decoding. I thought I would save some time for people in the future who might be interested in using a webcam feed for their psychopy project. All you need to do is import the opencv module into psychopy (importing modules was well documented by psychopy online) and integrate something like this into your psychopy script."

```
from psychopy import visual, event
import Image, pylab, cv

mywin = visual.Window(allowGUI=False, monitor='testMonitor', units='norm',
                      colorSpace='rgb', color=[-1, -1, -1], fullscr=True)
mywin.setMouseVisible(False)

capture = cv.CaptureFromCAM(0)
img = cv.QueryFrame(capture)
pi = Image.fromstring(
        "RGB", cv.GetSize(img), img.tostring(), "raw", "BGR", 0, 1)
```

(continues on next page)

(continued from previous page)

```
print(pi.size)
myStim = visual.GratingStim(win=mywin, tex=pi, pos=[0, 0.5], size=[0.6, 0.6],
                            opacity=1.0, units='norm')
myStim.setAutoDraw(True)

while True:
    img = cv.QueryFrame(capture)
    pi = Image.fromstring(
            "RGB", cv.GetSize(img), img.tostring(), "raw", "BGR", 0, 1)
    myStim.setTex(pi)
    mywin.flip()
    theKey = event.getKeys()
    if len(theKey) != 0:
        break
```

CHAPTER

ELEVEN

REFERENCE MANUAL (API)

Contents:

11.1 psychopy.core - basic functions (clocks etc.)

Basic functions, including timing, rush (imported), quit

class psychopy.core.**Clock**(*format=<class 'float'>*)

A convenient class to keep track of time in your experiments. You can have as many independent clocks as you like (e.g. one to time responses, one to keep track of stimuli …)

This clock is identical to the *MonotonicClock* except that it can also be reset to 0 or another value at any point.

add(*t*)

DEPRECATED: use .addTime() instead

This function adds time TO THE BASE (t0) which, counterintuitively, reduces the apparent time on the clock

addTime(*t*)

Add more time to the Clock/Timer

e.g.:

```
timer = core.Clock()
timer.addTime(5)
while timer.getTime() > 0:
    # do something
```

reset(*newT=0.0*)

Reset the time on the clock. With no args time will be set to zero. If a float is received this will be the new time on the clock

class psychopy.core.**CountdownTimer**(*start=0*)

Similar to a *Clock* except that time counts down from the time of last reset.

Parameters

start (*float or int*) – Starting time in seconds to countdown on.

Examples

Create a countdown clock with a 5 second duration:

```
timer = core.CountdownTimer(5)
while timer.getTime() > 0:  # after 5s will become negative
    # do stuff
```

addTime(*t*)

Add more time to the CountdownTimer

e.g.:

countdownTimer = core.CountdownTimer() countdownTimer.addTime(1)

while countdownTimer.getTime() > 0:

# do something

getTime()

Returns the current time left on this timer in seconds with sub-ms precision (*float*).

reset(*t=None*)

Reset the time on the clock.

Parameters

t (*float*, *int or None*) – With no args (*None*), time will be set to the time used for last reset (or start time if no previous resets). If a number is received, this will be the new time on the clock.

class psychopy.core.**MonotonicClock**(*start_time=None*, *format=<class 'float'>*)

A convenient class to keep track of time in your experiments using a sub-millisecond timer.

Unlike the *Clock* this cannot be reset to arbitrary times. For this clock t=0 always represents the time that the clock was created.

Don't confuse this *class* with *core.monotonicClock* which is an *instance* of it that got created when PsychoPy.core was imported. That clock instance is deliberately designed always to return the time since the start of the study.

Version Notes: This class was added in PsychoPy 1.77.00

getLastResetTime()

Returns the current offset being applied to the high resolution timebase used by Clock.

getTime(*applyZero=True*, *format=None*)

Returns the current time on this clock in secs (sub-ms precision).

Parameters

- **applyZero** (*bool*) – If applying zero then this will be the time since the clock was created (typically the beginning of the script). If not applying zero then it is whatever the underlying clock uses as its base time but that is system dependent. e.g. can be time since reboot, time since Unix Epoch etc.

  Only applies when format is *float*.

- **format** (*type*, *str or None*) – Format in which to show timestamp when converting to a string. Can be either: - time format codes: Time will return as a string in that format, as in time.strftime - *str*: Time will return as a string in ISO 8601 (YYYY-MM-DD_HH:MM:SS.mmmmmmZZZZ) - *None*: Will use this clock's *format* attribute

Returns

Time with format requested.

Return type
Timestamp

class psychopy.core.**StaticPeriod**(*screenHz=None*, *win=None*, *name='StaticPeriod'*)

A class to help insert a timing period that includes code to be run.

Parameters

- **screenHz** (*int or None*)
- **you** (*the frame rate of the monitor (leave as None if*) – don't want this accounted for)
- **win** (*Window*) – If a *Window* is given then *StaticPeriod* will also pause/restart frame interval recording.
- **name** (*str*) – Give this StaticPeriod a name for more informative logging messages.

Examples

Typical usage for the static period:

```
fixation.draw()
win.flip()
ISI = StaticPeriod(screenHz=60)
ISI.start(0.5)  # start a period of 0.5s
stim.image = 'largeFile.bmp'  # could take some time
ISI.complete()  # finish the 0.5s, taking into account one 60Hz frame

stim.draw()
win.flip()  # the period takes into account the next frame flip
# time should now be at exactly 0.5s later than when ISI.start()
# was called
```

complete()

Completes the period, using up whatever time is remaining with a call to *wait()*.

Returns
1 for success, *0* for fail (the period overran).

Return type
float

start(*duration*)

Start the period. If this is called a second time, the timer will be reset and starts again

Parameters
duration (*float or int*) – The duration of the period, in seconds.

psychopy.core.**getAbsTime**()

Get the absolute time.

This uses the same clock-base as the other timing features, like *getTime()*. The time (in seconds) ignores the time-zone (like *time.time()* on linux). To take the timezone into account, use *int(time.mktime(time.gmtime()))*.

Absolute times in seconds are especially useful to add to generated file names for being unique, informative (= a meaningful time stamp), and because the resulting files will always sort as expected when sorted in chronological, alphabetical, or numerical order, regardless of locale and so on.

Version Notes: This method was added in PsychoPy 1.77.00

Returns
Absolute Unix time (i.e., whole seconds elapsed since Jan 1, 1970).

Return type
float

`psychopy.core.`**`getTime`**(*applyZero=True*)

Get the current time since psychopy.core was loaded.

Version Notes: Note that prior to PsychoPy 1.77.00 the behaviour of getTime() was platform dependent (on OSX and linux it was equivalent to *`psychopy.core.getAbsTime()`* whereas on windows it returned time since loading of the module, as now)

`psychopy.core.`**`wait`**(*secs*, *hogCPUperiod=0.2*)

Wait for a given time period.

This function halts execution of the program for the specified duration.

Precision of this function is usually within 1 millisecond of the specified time, this may vary depending on factors such as system load and the Python version in use. Window events are periodically dispatched during the wait to keep the application responsive, to avoid the OS complaining that the process is unresponsive.

If *secs=10* and *hogCPU=0.2* then for 9.8s Python's *time.sleep* function will be used, which is not especially precise, but allows the cpu to perform housekeeping. In the final *hogCPUperiod* the more precise method of constantly polling the clock is used for greater precision.

If you want to obtain key-presses during the wait, be sure to use pyglet and then call *`psychopy.event.getKeys()`* after calling *`wait()`*

If you want to suppress checking for pyglet events during the wait, do this once:

```
core.checkPygletDuringWait = False
```

and from then on you can do:

```
core.wait(sec)
```

This will preserve terminal-window focus during command line usage.

Parameters

- **secs** (*float or int*) – Number of seconds to wait before continuing the program.
- **hogCPUperiod** (*float or int*) – Number of seconds to hog the CPU. This causes the thread to enter a 'tight' loop when the remaining wait time is less than the specified interval. This is set to 200ms (0.2s) by default. It is recommended that this interval is kept short to avoid stalling the processor for too long which may result in poorer timing.

11.2 `psychopy.clock` - Clocks and timers

Created on Tue Apr 23 11:28:32 2013

Provides the high resolution timebase used by psychopy, and defines some time related utility Classes.

Moved functionality from core.py so a common code base could be used in core.py and logging.py; vs. duplicating the getTime and Clock logic.

@author: Sol @author: Jon

class psychopy.clock.Clock(*format=<class 'float'>*)

A convenient class to keep track of time in your experiments. You can have as many independent clocks as you like (e.g. one to time responses, one to keep track of stimuli . . .)

This clock is identical to the *MonotonicClock* except that it can also be reset to 0 or another value at any point.

add(*t*)

DEPRECATED: use .addTime() instead

This function adds time TO THE BASE (t0) which, counterintuitively, reduces the apparent time on the clock

addTime(*t*)

Add more time to the Clock/Timer

e.g.:

```
timer = core.Clock()
timer.addTime(5)
while timer.getTime() > 0:
    # do something
```

reset(*newT=0.0*)

Reset the time on the clock. With no args time will be set to zero. If a float is received this will be the new time on the clock

class psychopy.clock.CountdownTimer(*start=0*)

Similar to a *Clock* except that time counts down from the time of last reset.

Parameters

start (*float or int*) – Starting time in seconds to countdown on.

Examples

Create a countdown clock with a 5 second duration:

```
timer = core.CountdownTimer(5)
while timer.getTime() > 0:  # after 5s will become negative
    # do stuff
```

addTime(*t*)

Add more time to the CountdownTimer

e.g.:

countdownTimer = core.CountdownTimer() countdownTimer.addTime(1)

while countdownTimer.getTime() > 0:

# do something

getTime()

Returns the current time left on this timer in seconds with sub-ms precision (*float*).

reset(*t=None*)

Reset the time on the clock.

Parameters

t (*float, int or None*) – With no args (*None*), time will be set to the time used for last reset (or start time if no previous resets). If a number is received, this will be the new time on the clock.

class psychopy.clock.**MonotonicClock**(*start_time=None*, *format=<class 'float'>*)

A convenient class to keep track of time in your experiments using a sub-millisecond timer.

Unlike the *Clock* this cannot be reset to arbitrary times. For this clock t=0 always represents the time that the clock was created.

Don't confuse this *class* with *core.monotonicClock* which is an *instance* of it that got created when PsychoPy.core was imported. That clock instance is deliberately designed always to return the time since the start of the study.

Version Notes: This class was added in PsychoPy 1.77.00

getLastResetTime()

Returns the current offset being applied to the high resolution timebase used by Clock.

getTime(*applyZero=True*, *format=None*)

Returns the current time on this clock in secs (sub-ms precision).

Parameters

- **applyZero** (*bool*) – If applying zero then this will be the time since the clock was created (typically the beginning of the script). If not applying zero then it is whatever the underlying clock uses as its base time but that is system dependent. e.g. can be time since reboot, time since Unix Epoch etc.

  Only applies when format is *float*.

- **format** (*type, str or None*) – Format in which to show timestamp when converting to a string. Can be either: - time format codes: Time will return as a string in that format, as in time.strftime - *str*: Time will return as a string in ISO 8601 (YYYY-MM-DD_HH:MM:SS.mmmmmmZZZZ) - *None*: Will use this clock's *format* attribute

Returns

Time with format requested.

Return type

Timestamp

class psychopy.clock.**StaticPeriod**(*screenHz=None*, *win=None*, *name='StaticPeriod'*)

A class to help insert a timing period that includes code to be run.

Parameters

- **screenHz** (*int or None*)
- **you** (*the frame rate of the monitor (leave as None if*) – don't want this accounted for)
- **win** (*Window*) – If a *Window* is given then *StaticPeriod* will also pause/restart frame interval recording.
- **name** (*str*) – Give this StaticPeriod a name for more informative logging messages.

Examples

Typical usage for the static period:

```
fixation.draw()
win.flip()
ISI = StaticPeriod(screenHz=60)
```

(continues on next page)

(continued from previous page)

```
ISI.start(0.5)  # start a period of 0.5s
stim.image = 'largeFile.bmp'  # could take some time
ISI.complete()  # finish the 0.5s, taking into account one 60Hz frame

stim.draw()
win.flip()  # the period takes into account the next frame flip
# time should now be at exactly 0.5s later than when ISI.start()
# was called
```

complete()

Completes the period, using up whatever time is remaining with a call to *wait()*.

Returns

1 for success, *0* for fail (the period overran).

Return type

float

start(*duration*)

Start the period. If this is called a second time, the timer will be reset and starts again

Parameters

duration (*float or int*) – The duration of the period, in seconds.

psychopy.clock.**getAbsTime**()

Get the absolute time.

This uses the same clock-base as the other timing features, like *getTime()*. The time (in seconds) ignores the time-zone (like *time.time()* on linux). To take the timezone into account, use *int(time.mktime(time.gmtime()))*.

Absolute times in seconds are especially useful to add to generated file names for being unique, informative (= a meaningful time stamp), and because the resulting files will always sort as expected when sorted in chronological, alphabetical, or numerical order, regardless of locale and so on.

Version Notes: This method was added in PsychoPy 1.77.00

Returns

Absolute Unix time (i.e., whole seconds elapsed since Jan 1, 1970).

Return type

float

psychopy.clock.**getTime**()

Copyright (c) 2018 Mario Kleiner. Licensed under MIT license.

For detailed help on a subfunction SUBFUNCTIONNAME, type GetSecs('SUBFUNCTIONNAME?') ie. the name with a question mark appended. E.g., for detailed help on the subfunction called Version, type this: GetSecs('Version?')

[GetSecsTime, WallTime, syncErrorSecs, MonotonicTime] = GetSecs('AllClocks' [, maxError=0.000020]);

psychopy.clock.**wait**(*secs*, *hogCPUperiod=0.2*)

Wait for a given time period.

This function halts execution of the program for the specified duration.

Precision of this function is usually within 1 millisecond of the specified time, this may vary depending on factors such as system load and the Python version in use. Window events are periodically dispatched during the wait to keep the application responsive, to avoid the OS complaining that the process is unresponsive.

If *secs=10* and *hogCPU=0.2* then for 9.8s Python's *time.sleep* function will be used, which is not especially precise, but allows the cpu to perform housekeeping. In the final *hogCPUperiod* the more precise method of constantly polling the clock is used for greater precision.

If you want to obtain key-presses during the wait, be sure to use pyglet and then call `psychopy.event.getKeys()` after calling `wait()`

If you want to suppress checking for pyglet events during the wait, do this once:

```
core.checkPygletDuringWait = False
```

and from then on you can do:

```
core.wait(sec)
```

This will preserve terminal-window focus during command line usage.

Parameters

- **secs** (*float or int*) – Number of seconds to wait before continuing the program.
- **hogCPUperiod** (*float or int*) – Number of seconds to hog the CPU. This causes the thread to enter a 'tight' loop when the remaining wait time is less than the specified interval. This is set to 200ms (0.2s) by default. It is recommended that this interval is kept short to avoid stalling the processor for too long which may result in poorer timing.

11.3 psychopy.session - for running a session with multiple experiments

Session

class psychopy.session.**Session**(*root*, *dataDir=None*, *clock='iso'*, *win=None*, *experiments=None*, *loggingLevel='info'*, *priorityThreshold=-9*, *params=None*, *liaison=None*, *restMsg='Rest'*)

A Session is from which you can run multiple PsychoPy experiments, so long as they are stored within the same folder. Session uses a persistent Window and inputs across experiments, meaning that you don't have to keep closing and reopening windows to run multiple experiments.

Through the use of multithreading, an experiment running via a Session can be sent commands and have variables changed while running. Methods of Session can be called from a second thread, meaning they don't have to wait for *runExperiment* to return on the main thread. For example, you could pause an experiment after 10s like so:

``` # define a function to run in a second thread def stopAfter10s(thisSession):

# wait 10s time.sleep(10) # pause thisSession.pauseExperiment()

# create a second thread thread = threading.Thread(

target=stopAfter10s, args=(thisSession,)

) # start the second thread thread.start() # run the experiment (in main thread) thisSession.runExperiment("testExperiment") ```

When calling methods of Session which have the parameter *blocking* from outside of the main thread, you can use *blocking=False* to force them to return immediately and, instead of executing, add themselves to a queue to be executed in the main thread by a while loop within the *start* function. This is important for methods like *runExperiment* or *setupWindowFromParams* which use OpenGL and so need to be run in the main thread. For example, you could alternatively run the code above like this:

``` # define a function to run in a second thread def stopAfter10s(thisSession):

# start the experiment in the main thread thisSession.runExperiment("testExperiment", blocking=False) # wait 10s time.sleep(10) # pause thisSession.pauseExperiment()

# create a second thread thread = threading.Thread(

target=stopAfter10s, args=(thisSession,)

) # start the second thread thread.start() # start the Session so that non-blocking methods are executed thisSession.start() ```

Parameters

- **root** (*str or pathlib.Path*) – Root folder for this session - should contain all of the experiments to be run.
- **liaison** (*liaison.WebSocketServer*) – Liaison server from which to receive run commands, if running via a liaison setup.
- **loggingLevel** (*str*) –

  How much output do you want in the log files? Should be one of the following:

  - 'error'
  - 'warning'
  - 'data'
  - 'exp'
  - 'info'
  - 'debug'

  ('error' is fewest messages, 'debug' is most)
- **inputs** (*dict, str or None*) – Dictionary of input objects for this session. Leave as None for a blank dict, or supply the name of an experiment to use the *setupInputs* method from that experiment.
- **win** (psychopy.visual.Window, *str or None*) – Window in which to run experiments this session. Supply a dict of parameters to make a Window from them, or supply the name of an experiment to use the *setupWindow* method from that experiment.
- **experiments** (*dict or None*) – Dict of name:experiment pairs which this Session can run. Each should be the file path of a .psyexp file, contained somewhere within the folder supplied for *root*. Paths can be absolute or relative to the root folder. Leave as None for a blank dict, experiments can be added later on via *addExperiment()*.
- **restMsg** (*str*) – Message to display inbetween experiments.

addAnnotation(*value*)

Add an annotation in the data file at the current point in the experiment and to the log.

Parameters
value (*str*) – Value of the annotation

Returns
True if completed successfully

Return type
bool

`addData`(*name*, *value*, *row=None*, *priority=None*)

Add data in the data file at the current point in the experiment, and to the log.

Parameters

- **name** (*str*) – Name of the column to add data as.
- **value** (*any*) – Value to add
- **row** (*int or None*) – Row in which to add this data. Leave as None to add to the current entry.
- **priority** (*int*) – Priority value to set the column to - higher priority columns appear nearer to the start of the data file. Use values from *constants.priority* as landmark values: - CRITICAL: Always at the start of the data file, generally reserved for Routine start times - HIGH: Important columns which are near the front of the data file - MEDIUM: Possibly important columns which are around the middle of the data file - LOW: Columns unlikely to be important which are at the end of the data file - EXCLUDE: Always at the end of the data file, actively marked as unimportant

Returns

True if completed successfully

Return type

bool

`addExperiment`(*file*, *key=None*, *folder=None*)

Register an experiment with this Session object, to be referred to later by a given key.

Parameters

- **file** (*str, Path*) – Path to the experiment (psyexp) file or script (py) of a Python experiment.
- **key** (*str*) – Key to refer to this experiment by once added. Leave as None to use file path relative to session root.
- **folder** (*str, Path*) – Folder for this project, if adding from outside of the root folder this entire folder will be moved. Leave as None to use the parent folder of *file*.

Returns

True if the operation completed successfully

Return type

bool or None

`addKeyboardFromParams`(*name*, *params*, *blocking=True*)

Add a keyboard to this session's inputs dict from a dict of params.

Parameters

- **name** (*str*) – Name of this input, what to store it under in the inputs dict.
- **params** (*dict*) – Dict of parameters to create the keyboard from, keys should be from the *addKeyboard* function in hardware.DeviceManager
- **blocking** (*bool*) – Should calling this method block the current thread?

  If True (default), the method runs as normal and won't return until completed. If False, the method is added to a *queue* and will be run by the while loop within *Session.start*. This will block the main thread, but won't block the thread this method was called from.

  If not using multithreading, this value is ignored. If you don't know what multithreading is, you probably aren't using it - it's difficult to do by accident!

Returns
True if the operation completed/queued successfully

Return type
bool or None

`close(blocking=True)`

Safely close and delete the current session.

Parameters
blocking (*bool*) – Should calling this method block the current thread?

If True (default), the method runs as normal and won't return until completed. If False, the method is added to a *queue* and will be run by the while loop within *Session.start*. This will block the main thread, but won't block the thread this method was called from.

If not using multithreading, this value is ignored. If you don't know what multithreading is, you probably aren't using it - it's difficult to do by accident!

`getAllTrials()`

Returns all trials (elapsed, current and upcoming) with an index indicating which trial is the current trial.

Returns

- *list[Trial]* – List of trials, in order (oldest to newest)
- *int* – Index of the current trial in this list

`getCurrentExpInfo()`

Get the *expInfo* dict for the currently running experiment.

Returns
The *expInfo* for the currently running experiment, or False if no experiment is running.

Return type
dict or False

`getCurrentExpInfoItem(key)`

Get the value of a key (or set of keys) from the current expInfo dict.

Parameters
key (*str or Iterable[str]*) – Key or keys to get values of fro expInfo dict

Returns
object, dict{str – If key was a string, the value of this key in expInfo. If key was a list of strings, a dict of key:value pairs for each key in the list. If no experiment is running or the process can't complete, False.

Return type
object} or False

`getCurrentTrial(asDict=False)`

Returns the current trial (*.thisTrial*)

Returns
The current trial

Return type
Trial

`getExpInfoFromExperiment(key, sessionParams=True)`

Get the global-level expInfo object from one of this Session's experiments. This will contain all of the keys needed for this experiment, alongside their default values.

Parameters

- **key** (*str*) – Key by which the experiment is stored (see *.addExperiment*).
- **sessionParams** (*bool*) – Should expInfo be extended with params from the Session, overriding experiment params where relevant (True, default)? Or return expInfo as it is in the experiment (False)?

Returns
Experiment info dict

Return type
dict

`getFrameRate`(*retest=False*)

Get the frame rate from the window.

Parameters
retest (*bool*) – If True, then will always run the frame rate test again, even if measured frame rate is already available.

Returns
Frame rate retrieved from Session window.

Return type
float

`getFutureTrial`(*n=1*, *asDict=False*)

Returns the condition for n trials into the future, without advancing the trials. Returns 'None' if attempting to go beyond the last trial in the current loop, if there is no current loop or if there is no current experiment.

Parameters

- **n** (*int*) – Number of places into the future to look
- **asDict** (*bool*) – If True, convert Trial object to a dict before returning (useful for Liaison)

`getFutureTrials`(*n=1*, *start=0*, *asDict=False*)

Returns Trial objects for a given range in the future. Will start looking at *start* trials in the future and will return n trials from then, so e.g. to get all trials from 2 in the future to 5 in the future you would use *start=2* and *n=3*.

Parameters

- **n** (*int, optional*) – How many trials into the future to look, by default 1
- **start** (*int, optional*) – How many trials into the future to start looking at, by default 0
- **asDict** (*bool*) – If True, convert Trial objects to a dict before returning (useful for Liaison)

Returns
List of Trial objects n long. Any trials beyond the last trial are None.

Return type
list[Trial or dict or None]

`getRequiredDeviceNamesFromExperiment`(*key*)

Get a list of device names referenced in a given experiment.

Parameters
key (*str*) – Key by which the experiment is stored (see *.addExperiment*).

Returns
List of device names

Return type
list[str]

`getStatus()`

Get an overall status flag for this Session. Will be one of either:

Returns
A value *psychopy.constants*, either: - NOT_STARTED: If no experiment is running - STARTED: If an experiment is running - PAUSED: If an experiment is paused - FINISHED: If an experiment is in the process of terminating

Return type
int

`getTime`(*format=<class 'str'>*)

Get time from this Session's clock object.

Parameters
`format` (*type, str or None*) – Can be either: - *float*: Time will return as a float as number of seconds - time format codes: Time will return as a string in that format, as in time.strftime - *str*: Time will return as a string in ISO 8601 (YYYY-MM-DD_HH:MM:SS.mmmmmmZZZZ) - *None*: Will use the Session clock object's *defaultStyle* attribute

Returns
Time in format requested.

Return type
str or float

`onIdle()`

Function to be called continuously while a SessionQueue is idle.

Returns
True if this Session was stopped safely.

Return type
bool

`pauseExperiment()`

Pause the currently running experiment.

Returns
True if the operation completed successfully

Return type
bool or None

`pauseLoop()`

Pause the current loop of the current experiment. Note that this will not take effect until the loop would next iterate.

Returns
True if the operation completed successfully

Return type
bool or None

`resumeExperiment()`

Resume the currently paused experiment.

Returns
True if the operation completed successfully

Return type
bool or None

`rewindTrials(`*n=1*`)`

Skip ahead n trials - the trials inbetween will be marked as "skipped". If you try to skip past the last trial, will log a warning and skip *to* the last trial.

Parameters
n (*int*) – Number of trials to skip ahead

Returns
True if the operation completed/queued successfully

Return type
bool or None

`runExperiment(`*key*, *expInfo=None*, *blocking=True*`)`

Run the *setupData* and *run* methods from one of this Session's experiments.

Parameters

- **key** (*str*) – Key by which the experiment is stored (see *.addExperiment*).
- **expInfo** (*dict*) – Information about the experiment, created by the *setupExpInfo* function.
- **blocking** (*bool*) – Should calling this method block the current thread?

  If True (default), the method runs as normal and won't return until completed. If False, the method is added to a *queue* and will be run by the while loop within *Session.start*. This will block the main thread, but won't block the thread this method was called from.

  If not using multithreading, this value is ignored. If you don't know what multithreading is, you probably aren't using it - it's difficult to do by accident!

Returns
True if the operation completed/queued successfully

Return type
bool or None

`saveCurrentExperimentData(`*blocking=True*`)`

Call *.saveExperimentData* on the currently running experiment - if there is one.

Parameters
blocking (*bool*) – Should calling this method block the current thread?

If True (default), the method runs as normal and won't return until completed. If False, the method is added to a *queue* and will be run by the while loop within *Session.start*. This will block the main thread, but won't block the thread this method was called from.

If not using multithreading, this value is ignored. If you don't know what multithreading is, you probably aren't using it - it's difficult to do by accident!

Returns
True if the operation completed/queued successfully, False if there was no current experiment running

Return type
bool or None

`saveExperimentData`(*key*, *thisExp=None*, *blocking=True*)

Run the *saveData* method from one of this Session's experiments, on a given ExperimentHandler.

Parameters

- **key** (*str*) – Key by which the experiment is stored (see *.addExperiment*).
- **thisExp** (`psychopy.data.ExperimentHandler`) – ExperimentHandler object to save the data from. If None, save the last run of the given experiment.
- **blocking** (*bool*) – Should calling this method block the current thread?

  If True (default), the method runs as normal and won't return until completed. If False, the method is added to a *queue* and will be run by the while loop within *Session.start*. This will block the main thread, but won't block the thread this method was called from.

  If not using multithreading, this value is ignored. If you don't know what multithreading is, you probably aren't using it - it's difficult to do by accident!

Returns
True if the operation completed/queued successfully

Return type
bool or None

`sendExperimentData`(*key=None*)

Send last ExperimentHandler for an experiment to liaison. If no experiment is given, sends the currently running experiment.

Parameters
key (*str or None*) – Name of the experiment whose data to send, or None to send the current experiment's data.

Returns
True if data was sent, otherwise False

Return type
bool

`sendToLiaison`(*value*)

Send data to this Session's *Liaison* object.

Parameters
value (*str*, *dict*, `psychopy.data.ExperimentHandler`) – Data to send - this can either be a single string, a dict of strings, or an ExperimentHandler (whose data will be sent)

Returns
True if the operation completed successfully

Return type
bool or None

`setCurrentExpInfoItem`(*key*, *value*)

Set the value of a key (or set of keys) from the current expInfo dict.

Parameters

- **key** (*str or Iterable[str]*) – Key or list of keys whose value or values to set.

- **value** (*object or Iterable[str]*) – Value or values to set the key to. If one value is given along with multiple keys, all keys will be set to that value. Otherwise, the number of values should match the number of keys.

Returns
True if operation completed successfully

Return type
bool

`setupDevicesFromExperiment`(*key*, *expInfo=None*, *thisExp=None*, *blocking=True*)

Setup inputs for this Session via the 'setupInputs` method from one of this Session's experiments.

Parameters

- **key** (*str*) – Key by which the experiment is stored (see *.addExperiment*).
- **expInfo** (*dict*) – Information about the experiment, created by the *setupExpInfo* function.
- **thisExp** (`psychopy.data.ExperimentHandler`) – Handler object for this experiment, contains the data to save and information about where to save it to.
- **blocking** (*bool*) – Should calling this method block the current thread?

  If True (default), the method runs as normal and won't return until completed. If False, the method is added to a *queue* and will be run by the while loop within *Session.start*. This will block the main thread, but won't block the thread this method was called from.

  If not using multithreading, this value is ignored. If you don't know what multithreading is, you probably aren't using it - it's difficult to do by accident!

Returns
True if the operation completed/queued successfully

Return type
bool or None

`setupInputsFromExperiment`(*key*, *expInfo=None*, *thisExp=None*, *blocking=True*)

Deprecated: legacy alias of setupDevicesFromExperiment

`setupWindowFromExperiment`(*key*, *expInfo=None*, *blocking=True*)

Setup the window for this Session via the 'setupWindow` method from one of this Session's experiments.

Parameters

- **key** (*str*) – Key by which the experiment is stored (see *.addExperiment*).
- **expInfo** (*dict*) – Information about the experiment, created by the *setupExpInfo* function.
- **blocking** (*bool*) – Should calling this method block the current thread?

  If True (default), the method runs as normal and won't return until completed. If False, the method is added to a *queue* and will be run by the while loop within *Session.start*. This will block the main thread, but won't block the thread this method was called from.

  If not using multithreading, this value is ignored. If you don't know what multithreading is, you probably aren't using it - it's difficult to do by accident!

Returns
True if the operation completed/queued successfully

Return type
bool or None

`setupWindowFromParams`(*params*, *measureFrameRate=False*, *recreate=True*, *blocking=True*)

Create/setup a window from a dict of parameters

Parameters

- **params** (*dict*) – Dict of parameters to create the window from, keys should be from the __init__ signature of psychopy.visual.Window
- **measureFrameRate** (*bool*) – If True, will measure frame rate upon window creation.
- **recreate** (*bool*) – If True, will close and recreate the window as needed
- **blocking** (*bool*) – Should calling this method block the current thread?

  If True (default), the method runs as normal and won't return until completed. If False, the method is added to a *queue* and will be run by the while loop within *Session.start*. This will block the main thread, but won't block the thread this method was called from.

  If not using multithreading, this value is ignored. If you don't know what multithreading is, you probably aren't using it - it's difficult to do by accident!

Returns

True if the operation completed/queued successfully

Return type

bool or None

`showExpInfoDlgFromExperiment`(*key*, *expInfo=None*)

Update expInfo for this Session via the 'showExpInfoDlg` method from one of this Session's experiments.

Parameters

- **key** (*str*) – Key by which the experiment is stored (see *.addExperiment*).
- **expInfo** (*dict*) – Information about the experiment, created by the *setupExpInfo* function.

Returns

True if the operation completed successfully

Return type

bool or None

`skipTrials`(*n=1*)

Skip ahead n trials - the trials inbetween will be marked as "skipped". If you try to skip past the last trial, will log a warning and skip *to* the last trial.

Parameters

n (*int*) – Number of trials to skip ahead

`start`()

Start this Session running its queue. Not recommended unless running across multiple threads.

Returns

True if this Session was started safely.

Return type

bool

`stop`()

Stop this Session running the queue. Not recommended unless running across multiple threads.

stopExperiment()

Stop the currently running experiment.

Returns

True if the operation completed successfully

Return type

bool or None

updateCurrentExpInfo(*other*)

Update key:value pairs in the current expInfo dict from another dict.

Parameters

other (*dict*) – key:value pairs to update dict from.

Returns

True if operation completed successfully

Return type

bool

property win

Window associated with this Session. Defined as a property so as to be accessible from Liaison if needed.

11.4 psychopy.visual - many visual stimuli

11.4.1 Aperture

Stimulus class to restrict a stimulus visibility area to a basic shape or list of vertices. This is a lazy-imported class, therefore import using full path *from psychopy.visual.aperture import Aperture* when inheriting from it.

Attributes

Details

class psychopy.visual.**Aperture**(**args*, ***kwargs*)

Restrict a stimulus visibility area to a basic shape or list of vertices.

When enabled, any drawing commands will only operate on pixels within the Aperture. Once disabled, subsequent draw operations affect the whole screen as usual.

Supported shapes:

- 'square', 'triangle', 'circle' or *None*: a polygon with appropriate nVerts will be used (120 for 'circle')
- integer: a polygon with that many vertices will be used
- list or numpy array (Nx2): it will be used directly as the vertices to a `ShapeStim`
- a filename then it will be used to load and image as a *ImageStim*. Note that transparent parts in the image (e.g. in a PNG file) will not be included in the mask shape. The color of the image will be ignored.

See demos/stimuli/aperture.py for example usage

Author

2011, Yuri Spitsyn 2011, Jon Peirce added units options, Jeremy Gray added shape & orientation 2014, Jeremy Gray added .contains() option 2015, Thomas Emmerling added ImageStim option

11.4.2 BoundingBox

Attributes

BoundingBox([extents, dtype])	Class for representing object bounding boxes.

Details

class psychopy.visual.**BoundingBox**(*extents=None*, *dtype=None*)

Class for representing object bounding boxes.

A bounding box is a construct which represents a 3D rectangular volume about some pose, defined by its minimum and maximum extents in the reference frame of the pose. The axes of the bounding box are aligned to the axes of the world or the associated pose.

Bounding boxes are primarily used for visibility testing; to determine if the extents of an object associated with a pose (eg. the vertices of a model) falls completely outside of the viewing frustum. If so, the model can be culled during rendering to avoid wasting CPU/GPU resources on objects not visible to the viewer.

_computeCorners()

Compute the corners of the bounding box.

These values are cached to speed up computations if extents hasn't been updated.

clear()

Clear a bounding box, invalidating it.

property dtype

Data type used for computations and arrays (*numpy.dtype*).

property extents

fit(*verts*)

Fit the bounding box to vertices.

property isValid

True if the bounding box is valid.

11.4.3 BoxStim

Attributes

BoxStim(win[, size, flipFaces, pos, ori, ...])	Class for drawing 3D boxes.

Details

class psychopy.visual.**BoxStim**(*win*, *size=(0.5, 0.5, 0.5)*, *flipFaces=False*, *pos=(0.0, 0.0, 0.0)*, *ori=(0.0, 0.0, 0.0, 1.0)*, *color=(0.0, 0.0, 0.0)*, *colorSpace='rgb'*, *contrast=1.0*, *opacity=1.0*, *useMaterial=None*, *textureScale=None*, *name=''*, *autoLog=True*)

Class for drawing 3D boxes. This is a lazy-imported class, therefore import using full path *from psychopy.visual.stim3d import BoxStim* when inheriting from it.

Draws a rectangular box with dimensions specified by *size* (length, width, height) in scene units.

Calling the *draw* method will render the box to the current buffer. The render target (FBO or back buffer) must have a depth buffer attached to it for the object to be rendered correctly. Shading is used if the current window has light sources defined and lighting is enabled (by setting *useLights=True* before drawing the stimulus).

> **Warning**
>
> This class is experimental and may result in undefined behavior.

Parameters

- **win** (*~psychopy.visual.Window*) – Window this stimulus is associated with. Stimuli cannot be shared across windows unless they share the same context.
- **size** (*tuple or float*) – Dimensions of the mesh. If a single value is specified, the box will be a cube. Provide a tuple of floats to specify the width, length, and height of the box (eg. *size=(0.2, 1.3, 2.1)*) in scene units.
- **flipFaces** (*bool, optional*) – If *True*, normals and face windings will be set to point inward towards the center of the box. Texture coordinates will remain the same. Default is *False*.
- **pos** (*array_like*) – Position vector *[x, y, z]* for the origin of the rigid body.
- **ori** (*array_like*) – Orientation quaternion *[x, y, z, w]* where *x*, *y*, *z* are imaginary and *w* is real. If you prefer specifying rotations in axis-angle format, call *setOriAxisAngle* after initialization.
- **useMaterial** (*PhongMaterial, optional*) – Material to use. The material can be configured by accessing the *material* attribute after initialization. If not material is specified, the diffuse and ambient color of the shape will track the current color specified by *glColor*. color : array_like Diffuse and ambient color of the stimulus if *useMaterial* is not specified. Values are with respect to *colorSpace*.
- **colorSpace** (*str*) – Colorspace of *color* to use.
- **contrast** (*float*) – Contrast of the stimulus, value modulates the *color*.
- **opacity** (*float*) – Opacity of the stimulus ranging from 0.0 to 1.0. Note that transparent objects look best when rendered from farthest to nearest.
- **textureScale** (*array_like or float, optional*) – Scaling factors for texture coordinates (sx, sy). By default, a factor of 1 will have the entire texture cover the surface of the mesh. If a single number is provided, the texture will be scaled uniformly.
- **name** (*str*) – Name of this object for logging purposes.
- **autoLog** (*bool*) – Enable automatic logging on attribute changes.

property RGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type

DEPRECATED

_createVAO(*vertices*, *textureCoords*, *normals*, *faces*)

Create a vertex array object for handling vertex attribute data.

_getDesiredRGB(*rgb*, *colorSpace*, *contrast*)

Convert color to RGB while adding contrast. Requires self.rgb, self.colorSpace and self.contrast

_selectWindow(*win*)

Switch drawing to the specified window. Calls the window's _setCurrent() method which handles the switch.

`_updateList()`

The user shouldn't need this method since it gets called after every call to .set() Chooses between using and not using shaders each call.

`property anchor`

`property backColor`

Alternative way of setting fillColor

`property backColorSpace`

Deprecated, please use colorSpace to set color space for the entire object.

`property backRGB`

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type

DEPRECATED

`property backgroundColor`

Alternative way of setting fillColor

`property borderColor`

`property borderColorSpace`

Deprecated, please use colorSpace to set color space for the entire object

`property borderRGB`

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type

DEPRECATED

`borderWidth`

`property color`

Alternative way of setting *foreColor*.

`property colorSpace`

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```python
# A light green text
stim = visual.TextStim(win, 'Color me!',
                       color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

property contrast

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged).
Operations supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast =  1.0  # unchanged contrast
stim.contrast =  0.5  # decrease contrast
stim.contrast =  0.0  # uniform, no contrast
stim.contrast = -0.5  # slightly inverted
stim.contrast = -1.0  # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast =  1.2  # increases contrast
stim.contrast = -1.2  # inverts with increased contrast
```

draw(*win=None*)

Draw the stimulus.

This should work for stimuli using a single VAO and material. More complex stimuli with multiple materials should override this method to correctly handle that case.

Parameters
win (*~psychopy.visual.Window*) – Window this stimulus is associated with. Stimuli cannot be shared across windows unless they share the same context.

property fillColor

Set the fill color for the shape.

property fillColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property fillRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type
DEPRECATED

property flip

1x2 array for flipping vertices along each axis; set as True to flip or False to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (*._flip*) will give an array of 1s and -1s with which to multiply vertices.

property flipHoriz

property flipVert

property fontColor

Alternative way of setting *foreColor*.

property foreColor

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names <http://www.w3schools.com/html/html_colornames.asp>* can be used.
- *Colors by hex value*
- **numerically: (scalar or triplet) for DKL, RGB or**
  other *Color spaces*. For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue'  # (the case is actually ignored)
stim.color = '#DDA0DD'  # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0]  # if stim.colorSpace='rgb':
                # a red color in rgb space
stim.color = [0.0, 45.0, 1.0]  # if stim.colorSpace='dkl':
                # DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255]  # if stim.colorSpace='rgb255':
                # a blue stimulus using rgb255 space
stim.color = 255  # interpreted as (255, 255, 255)
                  # which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1]  # increment all guns by 1 value
stim.color *= -1  # multiply the color by -1 (which in this
                    # space inverts the contrast)
stim.color *= [0.5, 0, 1]  # decrease red, remove green, keep blue
```

You can use *setColor* if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

property foreColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property foreRGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type

DEPRECATED

getOri()

`getOriAxisAngle`(*degrees=True*)

Get the axis and angle of rotation for the 3D stimulus. Converts the orientation defined by the *ori* quaternion to and axis-angle representation.

Parameters

`degrees` (*`bool, optional`*) – Specify `True` if *angle* is in degrees, or else it will be treated as radians. Default is `True`.

Returns

Axis *[rx, ry, rz]* and angle.

Return type

tuple

`getPos`()

`getRayIntersectBounds`(*rayOrig*, *rayDir*)

Get the point which a ray intersects the bounding box of this mesh.

Parameters

- **`rayOrig`** (*`array_like`*) – Origin of the ray in space [x, y, z].
- **`rayDir`** (*`array_like`*) – Direction vector of the ray [x, y, z], should be normalized.

Returns

Coordinate in world space of the intersection and distance in scene units from *rayOrig*. Returns *None* if there is no intersection.

Return type

tuple

`property height`

`isVisible`()

Check if the object is visible to the observer.

Test if a pose's bounding box or position falls outside of an eye's view frustum.

Poses can be assigned bounding boxes which enclose any 3D models associated with them. A model is not visible if all the corners of the bounding box fall outside the viewing frustum. Therefore any primitives (i.e. triangles) associated with the pose can be culled during rendering to reduce CPU/GPU workload.

Returns

True if the object's bounding box is visible.

Return type

bool

Examples

You can avoid running draw commands if the object is not visible by doing a visibility test first:

```
if myStim.isVisible():
    myStim.draw()
```

`property lineColor`

Alternative way of setting *borderColor*.

`property lineColorSpace`

Deprecated, please use colorSpace to set color space for the entire object

`property lineRGB`

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type
DEPRECATED

`lineWidth`

`property ori`

Orientation quaternion (X, Y, Z, W).

`property pos`

Position vector (X, Y, Z).

`setAnchor`(*value*, *log=None*)

`setBackColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

`setBackRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

`setBackgroundColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

`setBorderColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

`setBorderRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

`setBorderWidth`(*newWidth*, *operation=''*, *log=None*)

`setColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

`setContrast`(*newContrast*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

`setDKL`(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

`setFillColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for fillColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

`setFillRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

`setFontColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

`setForeColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for foreColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

`setForeRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

`setLMS`(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setLineColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setLineRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setLineWidth(*newWidth*, *operation=''*, *log=None*)

setOri(*ori*)

setOriAxisAngle(*axis*, *angle*, *degrees=True*)

Set the orientation of the 3D stimulus using an *axis* and *angle*. This sets the quaternion at *ori*.

Parameters

- **axis** (*array_like*) – Axis of rotation [rx, ry, rz].
- **angle** (*float*) – Angle of rotation.
- **degrees** (*bool, optional*) – Specify True if *angle* is in degrees, or else it will be treated as radians. Default is True.

setPos(*pos*)

setRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

property size

property thePose

The pose of the rigid body. This is a class which has *pos* and *ori* attributes.

units

None, 'norm', 'cm', 'deg', 'degFlat', 'degFlatPos', or 'pix'

If None then the current units of the *Window* will be used. See *Units for the window and stimuli* for explanation of other options.

Note that when you change units, you don't change the stimulus parameters and it is likely to change appearance. Example:

```
# This stimulus is 20% wide and 50% tall with respect to window
stim = visual.PatchStim(win, units='norm', size=(0.2, 0.5)

# This stimulus is 0.2 degrees wide and 0.5 degrees tall.
stim.units = 'deg'
```

updateColors()

Placeholder method to update colours when set externally, for example updating the *pallette* attribute of a textbox

property vertices

property width

property win

The *Window* object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1  # stimulus will be drawn in win1
stim.draw()  # stimulus is now drawn to win1
stim.win = win2  # stimulus will be drawn in win2
stim.draw()  # it is now drawn in win2
win1.flip(waitBlanking=False)  # do not wait for next
             # monitor update
win2.flip()  # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

11.4.4 `BufferImageStim`

Attributes

Details

class psychopy.visual.**BufferImageStim**(**args*, ***kwargs*)

Take a “screen-shot”, save as an ImageStim (RBGA object). This is a lazy-imported class, therefore import using full path *from psychopy.visual.bufferimage import BufferImageStim* when inheriting from it.

The screen-shot is a single collage image composed of static elements that you can treat as being a single stimulus. The screen-shot can be of the visible screen (front buffer) or hidden (back buffer).

BufferImageStim aims to provide fast rendering, while still allowing dynamic orientation, position, and opacity. It’s fast to draw but slower to init (same as an ImageStim).

You specify the part of the screen to capture (in norm units), and optionally the stimuli themselves (as a list of items to be drawn). You get a screenshot of those pixels. If your OpenGL does not support arbitrary sizes, the image will be larger, using square powers of two if needed, with the excess image being invisible (using alpha). The aim is to preserve the buffer contents as rendered.

Checks for OpenGL 2.1+, or uses square-power-of-2 images.

Example:

```
# define lots of stimuli, make a list:
mySimpleImageStim = ...
myTextStim = ...
stimList = [mySimpleImageStim, myTextStim]

# draw stim list items & capture (slow; see EXP log for times):
screenshot = visual.BufferImageStim(myWin, stim=stimList)

# render to screen (very fast, except for the first draw):
while <conditions>:
    screenshot.draw()  # fast; can vary .ori, .pos, .opacity
    other_stuff.draw() # dynamic
    myWin.flip()
```

See coder Demos > stimuli > bufferImageStim.py for a demo, with timing stats.

Author

- 2010 Jeremy Gray, with on-going fixes

Parameters

buffer :
the screen buffer to capture from, default is 'back' (hidden). 'front' is the buffer in view after win.flip()

rect :
a list of edges [left, top, right, bottom] defining a screen rectangle which is the area to capture from the screen, given in norm units. default is fullscreen: [-1, 1, 1, -1]

stim :
a list of item(s) to be drawn to the back buffer (in order). The back buffer is first cleared (without the win being flip()ed), then stim items are drawn, and finally the buffer (or part of it) is captured. Each item needs to have its own .draw() method, and have the same window as win.

interpolate :
whether to use interpolation (default = True, generally good, especially if you change the orientation)

sqPower2 :

- False (default) = use rect for size if OpenGL = 2.1+
- True = use square, power-of-two image sizes

flipHoriz :
horizontally flip (mirror) the captured image, default = False

flipVert :
vertically flip (mirror) the captured image; default = False

11.4.5 `psychopy.visual.Circle`

Stimulus class for drawing circles. This is a lazy-imported class, therefore import using full path *from psychopy.visual.circle import Circle* when inheriting from it.

Attributes

Details

class psychopy.visual.circle.**Circle**(*win*, *radius=0.5*, *edges='circle'*, *units=''*, *lineWidth=1.5*, *lineColor=None*, *fillColor='white'*, *colorSpace='rgb'*, *pos=(0, 0)*, *size=1.0*, *anchor=None*, *ori=0.0*, *opacity=None*, *contrast=1.0*, *depth=0*, *interpolate=True*, *draggable=False*, *lineRGB=False*, *fillRGB=False*, *name=None*, *autoLog=None*, *autoDraw=False*, *color=undefined*, *fillColorSpace=undefined*, *lineColorSpace=undefined*)

Creates a Circle with a given radius as a special case of a `ShapeStim`

Parameters

- **win** (*Window*) – Window this shape is being drawn to. The stimulus instance will allocate its required resources using that Windows context. In many cases, a stimulus instance cannot be drawn on different windows unless those windows share the same OpenGL context, which permits resources to be shared between them.
- **edges** (*int*) – Number of edges to use to define the outline of the circle. The greater the number of edges, the 'rounder' the circle will appear.

- **radius** (*float*) – Initial radius of the circle in *units*.
- **units** (*str*) – Units to use when drawing. This will affect how parameters and attributes *pos*, *size* and *radius* are interpreted.
- **lineWidth** (*float*) – Width of the circle's outline.
- **lineColor** (array_like, str, `Color` or None) – Color of the circle's outline and fill. If *None*, a fully transparent color is used which makes the fill or outline invisible.
- **fillColor** (array_like, str, `Color` or None) – Color of the circle's outline and fill. If *None*, a fully transparent color is used which makes the fill or outline invisible.
- **pos** (*array_like*) – Initial position (*x*, *y*) of the circle on-screen relative to the origin located at the center of the window or buffer in *units* (unless changed by specifying *viewPos*). This can be updated after initialization by setting the *pos* property. The default value is *(0.0, 0.0)* which results in no translation.
- **size** (*float or array_like*) – Initial scale factor for adjusting the size of the circle. A single value (*float*) will apply uniform scaling, while an array (*sx*, *sy*) will result in anisotropic scaling in the horizontal (*sx*) and vertical (*sy*) direction. Providing negative values to *size* will cause the shape being mirrored. Scaling can be changed by setting the *size* property after initialization. The default value is *1.0* which results in no scaling.
- **ori** (*float*) – Initial orientation of the circle in degrees about its origin. Positive values will rotate the shape clockwise, while negative values will rotate counterclockwise. The default value for *ori* is 0.0 degrees.
- **opacity** (*float*) – Opacity of the shape. A value of 1.0 indicates fully opaque and 0.0 is fully transparent (therefore invisible). Values between 1.0 and 0.0 will result in colors being blended with objects in the background. This value affects the fill (*fillColor*) and outline (*lineColor*) colors of the shape.
- **contrast** (*float*) – Contrast level of the shape (0.0 to 1.0). This value is used to modulate the contrast of colors passed to *lineColor* and *fillColor*.
- **depth** (*int*) – Depth layer to draw the stimulus when *autoDraw* is enabled.
- **interpolate** (*bool*) – Enable smoothing (anti-aliasing) when drawing shape outlines. This produces a smoother (less-pixelated) outline of the shape.
- **draggable** (*bool*) – Can this stimulus be dragged by a mouse click?
- **name** (*str*) – Optional name of the stimuli for logging.
- **autoLog** (*bool*) – Enable auto-logging of events associated with this stimuli. Useful for debugging and to track timing when used in conjunction with *autoDraw*.
- **autoDraw** (*bool*) – Enable auto drawing. When *True*, the stimulus will be drawn every frame without the need to explicitly call the `draw()` method.
- **color** (array_like, str, `Color` or *None*) – Sets both the initial *lineColor* and *fillColor* of the shape.
- **colorSpace** (*str*) – Sets the colorspace, changing how values passed to *lineColor* and *fillColor* are interpreted.

radius

Radius of the shape. Avoid using *size* for adjusting figure dimensions if radius != 0.5 which will result in undefined behavior.

Type

float or int

`property RGB`

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type
DEPRECATED

`static _calcEquilateralVertices(`*edges*, *radius=0.5*`)`

Get vertices for an equilateral shape with a given number of sides, will assume radius is 0.5 (relative) but can be manually specified

`_calcPosRendered()`

DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix.

`_calcSizeRendered()`

DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix

`static _calculateMinEdges(`*lineWidth*, *threshold=180*`)`

Calculate how many points are needed in an equilateral polygon for the gap between line rects to be < 1px and for corner angles to exceed a threshold.

In other words, how many edges does a polygon need to have to appear smooth?

lineWidth
[int, float, np.ndarray] Width of the line in pixels

threshold
[int] Maximum angle (degrees) for corners of the polygon, useful for drawing a circle. Supply 180 for no maximum angle.

`_drawLegacyGL(`*win*, *keepMatrix*`)`

Legacy draw the stimulus in the relevant window.

You must call this method after every *win.flip()* if you want the stimulus to appear on that frame and then update the screen again.

`_getDesiredRGB(`*rgb*, *colorSpace*, *contrast*`)`

Convert color to RGB while adding contrast. Requires self.rgb, self.colorSpace and self.contrast

`_getPolyAsRendered()`

DEPRECATED. Return a list of vertices as rendered.

`_legacyTesselate(`*newVertices*`)`

Legacy tessellation method for ShapeStim.

`_selectWindow(`*win*`)`

Switch drawing to the specified window. Calls the window's _setCurrent() method which handles the switch.

`_set(`*attrib*, *val*, *op=''*, *log=None*`)`

DEPRECATED since 1.80.04 + 1. Use setAttribute() and val2array() instead.

`_tesselate(`*newVertices*`)`

Set the *.vertices* and *.border* to new values, invoking tessellation.

Parameters
`newVertices` (`array_like`) – Nx2 array of points (eg., *[[-0.5, 0], [0, 0.5], [0.5, 0]]*).

`_updateList()`

The user shouldn't need this method since it gets called after every call to .set() Chooses between using and not using shaders each call.

`_updateVertices()`

Sets Stim.verticesPix and ._borderPix from pos, size, ori, flipVert, flipHoriz

`alphaThreshold`

Threshold for alpha values.

If the alpha value of a pixel is below this threshold, the pixel will be rejected (not drawn). This can be useful for creating a mask from an image with an alpha channel. The default value is 0.0, which means that no thresholding will be applied.

`autoDraw`

Determines whether the stimulus should be automatically drawn on every frame flip.

Value should be: *True* or *False*. You do NOT need to set this on every frame flip!

`autoLog`

Whether every change in this stimulus should be auto logged.

Value should be: *True* or *False*. Set to *False* if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren't likely to be useful.

`property backColor`

Alternative way of setting fillColor

`property backColorSpace`

Deprecated, please use colorSpace to set color space for the entire object.

`property backRGB`

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type
DEPRECATED

`property backgroundColor`

Alternative way of setting fillColor

`property borderColorSpace`

Deprecated, please use colorSpace to set color space for the entire object

`property borderRGB`

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type
DEPRECATED

`closeShape`

Should the last vertex be automatically connected to the first?

If you're using *Polygon*, *Circle* or *Rect*, *closeShape=True* is assumed and shouldn't be changed.

`color`

Set the color of the shape. Sets both *fillColor* and *lineColor* simultaneously if applicable.

`property colorSpace`

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                       color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

`contains`(*x*, *y=None*, *units=None*)

Returns True if a point x,y is inside the stimulus' border.

Can accept variety of input options:

- two separate args, x and y
- one arg (list, tuple or array) containing two vals (x,y)
- **an object with a getPos() method that returns x,y, such**
  as a *`Mouse`*.

Returns *True* if the point is within the area defined either by its *border* attribute (if one defined), or its *vertices* attribute if there is no .border. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the *contains* method; the extent of the stimulus is determined purely by the size, position (pos), and orientation (ori) settings (and by the vertices for shape stimuli).

See Coder demos: shapeContains.py See Coder demos: shapeContains.py

`property contrast`

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged).
Operations supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast =  1.0  # unchanged contrast
stim.contrast =  0.5  # decrease contrast
stim.contrast =  0.0  # uniform, no contrast
stim.contrast = -0.5  # slightly inverted
stim.contrast = -1.0  # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast =  1.2  # increases contrast
stim.contrast = -1.2  # inverts with increased contrast
```

`depth`

DEPRECATED, depth is now controlled simply by drawing order.

`doDragging()`

If this stimulus is draggable, do the necessary actions on a frame flip to drag it.

`draggable`

Can this stimulus be dragged by a mouse click?

`draw`(*win=None*, *keepMatrix=False*)

Draw the stimulus in the relevant window.

You must call this method after every *win.flip()* if you want the stimulus to appear on that frame and then update the screen again.

Parameters

- **win** (`Window`, optional) – Window to draw the stimulus in. If not specified, the stimulus will be drawn in the window specified at initialization.
- **keepMatrix** (`bool, optional`) – *DEPRECATED* If *True*, the current transformation matrix will be preserved. This is useful when drawing multiple stimuli with the same transformation matrix. Default is *False*.

`edges`

Number of edges of the polygon. Floats are rounded to int.

Operations supported.

`property fillColor`

Set the fill color for the shape.

`property fillColorSpace`

Deprecated, please use colorSpace to set color space for the entire object.

`property fillRGB`

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type

DEPRECATED

`property flip`

1x2 array for flipping vertices along each axis; set as True to flip or False to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (*._flip*) will give an array of 1s and -1s with which to multiply vertices.

`property fontColor`

Alternative way of setting *foreColor*.

`property foreColor`

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names <http://www.w3schools.com/html/html_colornames.asp>* can be used.
- *Colors by hex value*
- **numerically: (scalar or triplet) for DKL, RGB or**
  other *Color spaces*. For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue'  # (the case is actually ignored)
stim.color = '#DDA0DD'  # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0]  # if stim.colorSpace='rgb':
                # a red color in rgb space
stim.color = [0.0, 45.0, 1.0]  # if stim.colorSpace='dkl':
                # DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255]  # if stim.colorSpace='rgb255':
                # a blue stimulus using rgb255 space
stim.color = 255  # interpreted as (255, 255, 255)
                  # which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1]  # increment all guns by 1 value
stim.color *= -1  # multiply the color by -1 (which in this
                    # space inverts the contrast)
stim.color *= [0.5, 0, 1]  # decrease red, remove green, keep blue
```

You can use *setColor* if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

property foreColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property foreRGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type
DEPRECATED

interpolate

If *True* the edge of the line will be anti-aliased.

property lineColor

Alternative way of setting *borderColor*.

property lineColorSpace

Deprecated, please use colorSpace to set color space for the entire object

property lineRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type
DEPRECATED

`lineWidth`

Width of the line in **pixels**.

Operations supported.

`name`

The name (*str*) of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = None your stimulus will be called "unnamed <type>", e.g. visual.TextStim(win) will be called "unnamed TextStim" in the logs.

`property opacity`

Determines how visible the stimulus is relative to background.

The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). *Operations* are supported. Precisely how this is used depends on the *Blend Mode*.

`ori`

The orientation of the stimulus (in degrees).

Should be a single value (*scalar*). *Operations* are supported.

Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and below zero values wrap appropriately.

`overlaps`(*polygon*)

Returns *True* if this stimulus intersects another one.

If *polygon* is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the *overlaps* method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

See coder demo, shapeContains.py

`property pos`

The position of the center of the stimulus in the stimulus *units*

value should be an *x,y-pair*. *Operations* are also supported.

Example:

```
stim.pos = (0.5, 0)  # Set slightly to the right of center
stim.pos += (0.5, -1)  # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2  # Move stim towards the center.
    Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:

```
from psychopy.tools.monitorunittools import posToPix
posPix = posToPix(stim)
```

`radius`

float, int, tuple, list or 2x1 array Radius of the Polygon (distance from the center to the corners). May be a -2tuple or list to stretch the polygon asymmetrically.

Operations supported.

Usually there's a setAttribute(value, log=False) method for each attribute. Use this if you want to disable logging.

`setAlphaThreshold`(*value*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

`setAutoDraw`(*value*, *log=None*)

Sets autoDraw. Usually you can use 'stim.attribute = value' syntax instead, but use this method to suppress the log message.

`setAutoLog`(*value=True*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

`setBackRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

`setBorderColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

`setBorderRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

`setColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Sets both the line and fill to be the same color.

`setContrast`(*newContrast*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

`setDKL`(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

`setDepth`(*newDepth*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

`setEdges`(*edges*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

`setFillColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for fillColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

`setFillRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

`setForeColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for foreColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

`setForeRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setLMS(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setLineRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setNVertices(*nVerts*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setOpacity(*newOpacity*, *operation=''*, *log=None*)

Hard setter for opacity, allows the suppression of log messages and calls the update method

setOri(*newOri*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setPos(*newPos*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setRadius(*radius*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setSize(*newSize*, *operation=''*, *units=None*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setVertices(*value=None*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

property size

The size (width, height) of the stimulus in the stimulus *units*

Value should be *x,y-pair*, *scalar* (applies to both dimensions) or None (resets to default). *Operations* are supported.

Sizes can be negative (causing a mirror-image reversal) and can extend beyond the window.

Example:

```
stim.size = 0.8  # Set size to (xsize, ysize) = (0.8, 0.8)
print(stim.size)  # Outputs array([0.8, 0.8])
stim.size += (0.5, -0.5)  # make wider and flatter: (1.3, 0.3)
```

Tip: if you can see the actual pixel range this corresponds to by looking at *stim._sizeRendered*

updateColors()

Placeholder method to update colours when set externally, for example updating the *pallette* attribute of a textbox

updateOpacity()

Placeholder method to update colours when set externally, for example updating the *pallette* attribute of a textbox.

property vertices

A list of lists or a numpy array (Nx2) specifying xy positions of each vertex, relative to the center of the field.

Assigning to vertices can be slow if there are many vertices.

Operations supported with *.setVertices()*.

property verticesPix

This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

property win

The *Window* object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1  # stimulus will be drawn in win1
stim.draw()  # stimulus is now drawn to win1
stim.win = win2  # stimulus will be drawn in win2
stim.draw()  # it is now drawn in win2
win1.flip(waitBlanking=False)  # do not wait for next
             # monitor update
win2.flip()  # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

11.4.6 CustomMouse

class psychopy.visual.**CustomMouse**(**args*, ***kwargs*)

Class for more control over the mouse, including the pointer graphic and bounding box. This is a lazy-imported class, therefore import using full path *from psychopy.visual.custommouse import CustomMouse* when inheriting from it.

Seems to work with pyglet or pygame. Not completely tested.

Known limitations:

- only norm units are working
- getRel() always returns [0,0]
- mouseMoved() is always False; maybe due to *self.mouse.visible* == *False* -> held at [0,0]
- no idea if clickReset() works

Author: Jeremy Gray, 2011

Class for customizing the appearance and behavior of the mouse.

Use a custom mouse for extra control over the pointer appearance and function. It's probably slower to render than the regular system mouse. Create your *visual.Window* before creating a CustomMouse.

Parameters

- **win** (required, *visual.Window*) – the window to which this mouse is attached
- **visible** (**True** or False) – makes the mouse invisible if necessary
- **newPos** (**None** or [x,y]) – gives the mouse a particular starting position
- **leftLimit** – left edge of a virtual box within which the mouse can move
- **topLimit** – top edge of virtual box
- **rightLimit** – right edge of virtual box
- **bottomLimit** – lower edge of virtual box
- **showLimitBox** (*default is False*) – display the boundary within which the mouse can move.
- **pointer** – The visual display item to use as the pointer; must have .draw() and setPos() methods. If your item has .setOpacity(), you can alter the mouse's opacity.
- **clickOnUp** (*when to count a mouse click as having occurred*) – default is False, record a click when the mouse is first pressed down. True means record a click when the mouse button is released.

11.4.7 DotStim

class psychopy.visual.**DotStim**(**args*, ***kwargs*)

This stimulus class defines a field of dots with an update rule that determines how they change on every call to the .draw() method. This is a lazy-imported class, therefore import using full path *from psychopy.visual.dot import DotStim* when inheriting from it.

This single class can be used to generate a wide variety of dot motion types. For a review of possible types and their pros and cons see Scase, Braddick & Raymond (1996). All six possible motions they describe can be generated with appropriate choices of the *signalDots* (which determines whether signal dots are the 'same' or 'different' on each frame), *noiseDots* (which determines the locations of the noise dots on each frame) and the *dotLife* (which determines for how many frames the dot will continue before being regenerated).

The default settings (as of v1.70.00) is for the noise dots to have identical velocity but random direction and signal dots remain the 'same' (once a signal dot, always a signal dot).

For further detail about the different configurations see *Dots Component* in the Builder Components section of the documentation.

If further customisation is required, then the DotStim should be subclassed and its _update_dotsXY and _newDotsXY methods overridden.

The maximum number of dots that can be drawn is limited by system performance.

fieldShape

'sqr' or 'circle'. Defines the envelope used to present the dots. If changed while drawing, dots outside new envelope will be respawned.

Type
str

dotSize

Dot size specified in pixels (overridden if *element* is specified). *operations* are supported.

Type
float

`dotLife`

Number of frames each dot lives for (-1=infinite). Dot lives are initiated randomly from a uniform distribution from 0 to dotLife. If changed while drawing, the lives of all dots will be randomly initiated again.

Type
int

`signalDots`

If 'same' then the signal and noise dots are constant. If 'different' then the choice of which is signal and which is noise gets randomised on each frame. This corresponds to Scase et al's (1996) categories of RDK.

Type
str

`noiseDots`

Determines the behaviour of the noise dots, taken directly from Scase et al's (1996) categories. For 'position', noise dots take a random position every frame. For 'direction' noise dots follow a random, but constant direction. For 'walk' noise dots vary their direction every frame, but keep a constant speed.

Type
str

`element`

This can be any object that has a `.draw()` method and a `.setPos([x,y])` method (e.g. a GratingStim, TextStim...)!! DotStim assumes that the element uses pixels as units. `None` defaults to dots.

Type
object

`fieldPos`

Specifying the location of the centre of the stimulus using a *x,y-pair*. See e.g. `ShapeStim` for more documentation/examples on how to set position. *operations* are supported.

Type
array_like

`fieldSize`

Specifying the size of the field of dots using a *x,y-pair*. See e.g. `ShapeStim` for more documentation/examples on how to set position. *operations* are supported.

Type
array_like

`coherence`

Change the coherence (%) of the DotStim. This will be rounded according to the number of dots in the stimulus.

Type
float

`dir`

Direction of the coherent dots in degrees. *operations* are supported.

Type
float

`speed`

Speed of the dots (in *units*/frame). *operations* are supported.

Type
float

Parameters

- **win** (window.Window) – Window this stimulus is associated with.
- **units** (*str*) – Units to use.
- **nDots** (*int*) – Number of dots to present in the field.
- **coherence** (*float*) – Proportion of dots which are coherent. This value can be set using the *coherence* property after initialization.
- **fieldPos** (*array_like*) – (x,y) or [x,y] position of the field. This value can be set using the *fieldPos* property after initialization.
- **fieldSize** (*array_like, int or float*) – (x,y) or [x,y] or single value (applied to both dimensions). Sizes can be negative and can extend beyond the window. This value can be set using the *fieldSize* property after initialization.
- **fieldShape** (*str*) – Defines the envelope used to present the dots. If changed while drawing by setting the *fieldShape* property, dots outside new envelope will be respawned., valid values are 'square', 'sqr' or 'circle'.
- **dotSize** (*array_like or float*) – Size of the dots. If given an array, the sizes of individual dots will be set. The array must have length *nDots*. If a single value is given, all dots will be set to the same size.
- **dotLife** (*int*) – Lifetime of a dot in frames. Dot lives are initiated randomly from a uniform distribution from 0 to dotLife. If changed while drawing, the lives of all dots will be randomly initiated again. A value of -1 results in dots having an infinite lifetime. This value can be set using the *dotLife* property after initialization.
- **dir** (*float*) – Direction of the coherent dots in degrees. At 0 degrees, coherent dots will move from left to right. Increasing the angle will rotate the direction counter-clockwise. This value can be set using the *dir* property after initialization.
- **speed** (*float*) – Speed of the dots (in *units* per frame). This value can be set using the *speed* property after initialization.
- **rgb** (*array_like, optional*) – Color of the dots in form (r, g, b) or [r, g, b]. **Deprecated**, use *color* instead.
- **color** (*array_like or str*) – Color of the dots in form (r, g, b) or [r, g, b].
- **colorSpace** (*str*) – Colorspace to use.
- **opacity** (*float*) – Opacity of the dots from 0.0 to 1.0.
- **contrast** (*float*) – Contrast of the dots 0.0 to 1.0. This value is simply multiplied by the *color* value.
- **depth** (*float*) – **Deprecated**, depth is now controlled simply by drawing order.
- **element** (*object*) – This can be any object that has a `.draw()` method and a `.setPos([x, y])` method (e.g. a GratingStim, TextStim...)!! DotStim assumes that the element uses pixels as units. `None` defaults to dots.
- **signalDots** (*str*) – If 'same' then the signal and noise dots are constant. If different then the choice of which is signal and which is noise gets randomised on each frame. This corresponds to Scase et al's (1996) categories of RDK. This value can be set using the *signalDots* property after initialization.
- **noiseDots** (*str*) – Determines the behaviour of the noise dots, taken directly from Scase et al's (1996) categories. For 'position', noise dots take a random position every frame. For 'direction' noise dots follow a random, but constant direction. For 'walk' noise dots vary their

direction every frame, but keep a constant speed. This value can be set using the *noiseDots* property after initialization.

- **name** (*str, optional*) – Optional name to use for logging.
- **autoLog** (*bool*) – Enable automatic logging.

11.4.8 ElementArrayStim

class psychopy.visual.**ElementArrayStim**(**args*, ***kwargs*)

This stimulus class defines a field of elements whose behaviour can be independently controlled. Suitable for creating 'global form' stimuli or more detailed random dot stimuli. This is a lazy-imported class, therefore import using full path *from psychopy.visual.elementarray import ElementArrayStim* when inheriting from it.

This stimulus can draw thousands of elements without dropping a frame, but in order to achieve this performance, uses several OpenGL extensions only available on modern graphics cards (supporting OpenGL2.0). See the ElementArray demo.

Parameters

win :
a *Window* object (required)

units
[**None**, 'height', 'norm', 'cm', 'deg' or 'pix'] If None then the current units of the *Window* will be used. See *Units for the window and stimuli* for explanation of other options.

nElements :
number of elements in the array.

11.4.9 Form

Attributes

Details

class psychopy.visual.form.**Form**(*win*, *name='default'*, *colorSpace='rgb'*, *fillColor=None*, *borderColor=None*, *itemColor='white'*, *responseColor='white'*, *markerColor='red'*, *items=None*, *font=None*, *textHeight=0.02*, *size=(0.5, 0.5)*, *pos=(0, 0)*, *style=None*, *itemPadding=0.05*, *units='height'*, *randomize=False*, *autoLog=True*, *depth=0*, *color=undefined*, *foreColor=undefined*)

A class to add Forms to a *psychopy.visual.Window*

The Form allows Psychopy to be used as a questionnaire tool, where participants can be presented with a series of questions requiring responses. Form items, defined as questions and response pairs, are presented simultaneously onscreen with a scrollable viewing window.

Example

survey = Form(win, items=[{}], size=(1.0, 0.7), pos=(0.0, 0.0))

Parameters

- **win** (psychopy.visual.Window) – The window object to present the form.
- **items** (*List of dicts or csv or xlsx file*) –

  a list of dicts or csv file should have the following key, value pairs / column headers:
  "index": The item index as a number "itemText": item question string, "itemWidth":

fraction of the form width 0:1 "type": type of rating e.g., 'radio', 'rating', 'slider' "responseWidth": fraction of the form width 0:1, "options": list of tick labels for options, "layout": Response object layout e.g., 'horiz' or 'vert'

- **textHeight** (*float*) – Text height.
- **size** (*tuple, list*) – Size of form on screen.
- **pos** (*tuple, list*) – Position of form on screen.
- **itemPadding** (*float*) – Space or padding between form items.
- **units** (*str*) – units for stimuli - Currently, Form class only operates with 'height' units.
- **randomize** (*bool*) – Randomize order of Form elements

property RGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type
DEPRECATED

_calcPosRendered()

DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix.

_calcSizeRendered()

DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix

_createItemCtrls()

Define layout of form

_drawCtrls()

Draw elements on form within border range.

Parameters
items (*List*) – List of TextStim or Slider item from survey

_drawDecorations()

Draw decorations on form.

_drawExternalDecorations()

Draw decorations outside the aperture

_getDesiredRGB(*rgb*, *colorSpace*, *contrast*)

Convert color to RGB while adding contrast. Requires self.rgb, self.colorSpace and self.contrast

_getItemHeight(*item*, *ctrl=None*)

Returns the full height of the item to be inserted in the form

_getItemRenderedWidth(*size*)

Returns text width for item text based on itemWidth and Form width.

Parameters
size (*float, int*) – The question width

Returns
Wrap width for question text

Return type
float

_getPolyAsRendered()

DEPRECATED. Return a list of vertices as rendered.

_getScrollOffset()

Calculate offset position of items in relation to markerPos. Offset is a proportion of *vheight* - *height*, meaning the max offset (when scrollbar.markerPos is 1) is enough to take the bottom element to the bottom of the border.

Returns
Offset position of items proportionate to scroll bar

Return type
float

_inRange(*item*)

Check whether item position falls within border area

Parameters
item (TextStim, *Slider object*) – TextStim or Slider item from survey

Returns
Returns True if item position falls within border area

Return type
bool

_layoutY()

This needs to be done when editable textboxes change their size because everything below them needs to move too

_makeSlider(*item*)

Creates Slider object for Form class

Parameters

- **item** (*dict*) – The dict entry for a single item
- **pos** (*tuple*) – position of response object

Returns

- *psychopy.visual.slider.Slider* – The Slider object for response
- *respHeight* – The height of the response object as type float

_makeTextBox(*item*)

Creates TextBox object for Form class

NOTE: The TextBox 2 in work in progress, and has not been added to Form class yet. :param item: The dict entry for a single item :type item: dict :param pos: position of response object :type pos: tuple

Returns

- *psychopy.visual.TextBox* – The TextBox object for response
- *respHeight* – The height of the response object as type float

_selectWindow(*win*)

Switch drawing to the specified window. Calls the window's _setCurrent() method which handles the switch.

_set(*attrib*, *val*, *op=''*, *log=None*)

DEPRECATED since 1.80.04 + 1. Use setAttribute() and val2array() instead.

`_setAperture``()`

Blocks text beyond border using Aperture

Returns

The aperture setting viewable area for forms

Return type

psychopy.visual.Aperture

`_setBorder``()`

Creates border using Rect

Returns

The border for the survey

Return type

psychopy.visual.Rect

`_setDecorations``()`

Sets Form decorations i.e., Border and scrollbar

`_setQuestion``(`*item*`)`

Creates TextStim object containing question

Parameters

`item` (*`dict`*) – The dict entry for a single item

Returns

- *psychopy.visual.text.TextStim* – The textstim object with the question string
- *questionHeight* – The height of the question bounding box as type float
- *questionWidth* – The width of the question bounding box as type float

`_setResponse``(`*item*`)`

Makes calls to methods which make Slider or TextBox response objects for Form

Parameters

- **`item`** (*`dict`*) – The dict entry for a single item
- **`question`** (`TextStim`) – The question text object

Returns

- *psychopy.visual.slider.Slider* – The Slider object for response
- *psychopy.visual.TextBox* – The TextBox object for response
- *respHeight* – The height of the response object as type float

`_setScrollBar``()`

Creates Slider object for scrollbar

Returns

The Slider object for scroll bar

Return type

psychopy.visual.slider.Slider

`_updateList``()`

The user shouldn't need this method since it gets called after every call to .set() Chooses between using and not using shaders each call.

`_updateVertices()`

Sets Stim.verticesPix and ._borderPix from pos, size, ori, flipVert, flipHoriz

`addDataToExp`(*exp*, *itemsAs='rows'*)

Gets the current Form data and inserts into an `ExperimentHandler` object either as rows or as columns

Parameters

- **exp** (`ExperimentHandler`)
- **itemsAs** (*'rows','cols' (or 'columns')*)

`alphaThreshold`

Threshold for alpha values.

If the alpha value of a pixel is below this threshold, the pixel will be rejected (not drawn). This can be useful for creating a mask from an image with an alpha channel. The default value is 0.0, which means that no thresholding will be applied.

`property anchor`

`autoDraw`

Determines whether the stimulus should be automatically drawn on every frame flip.

Value should be: *True* or *False*. You do NOT need to set this on every frame flip!

`autoLog`

Whether every change in this stimulus should be auto logged.

Value should be: *True* or *False*. Set to *False* if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren't likely to be useful.

`property backColor`

Alternative way of setting fillColor

`property backColorSpace`

Deprecated, please use colorSpace to set color space for the entire object.

`property backRGB`

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type
DEPRECATED

`property backgroundColor`

Alternative way of setting fillColor

`property borderColor`

Color of the line around the form

`property borderColorSpace`

Deprecated, please use colorSpace to set color space for the entire object

`property borderRGB`

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type
DEPRECATED

borderWidth

property color

Alternative way of setting *foreColor*.

property colorSpace

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                       color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

property complete

A read-only property to determine if the current form is complete

contains(*x*, *y=None*, *units=None*)

Returns True if a point x,y is inside the stimulus' border.

Can accept variety of input options:

- two separate args, x and y
- one arg (list, tuple or array) containing two vals (x,y)
- **an object with a getPos() method that returns x,y, such**
  as a `Mouse`.

Returns *True* if the point is within the area defined either by its *border* attribute (if one defined), or its *vertices* attribute if there is no .border. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the *contains* method; the extent of the stimulus is determined purely by the size, position (pos), and orientation (ori) settings (and by the vertices for shape stimuli).

See Coder demos: shapeContains.py See Coder demos: shapeContains.py

property contrast

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged).
Operations supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast =  1.0  # unchanged contrast
stim.contrast =  0.5  # decrease contrast
stim.contrast =  0.0  # uniform, no contrast
stim.contrast = -0.5  # slightly inverted
stim.contrast = -1.0  # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast =  1.2  # increases contrast
stim.contrast = -1.2  # inverts with increased contrast
```

depth

DEPRECATED, depth is now controlled simply by drawing order.

draw()

Draw all form elements

property fillColor

Color of the form's background

property fillColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property fillRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type
DEPRECATED

property flip

1x2 array for flipping vertices along each axis; set as True to flip or False to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (*._flip*) will give an array of 1s and -1s with which to multiply vertices.

property flipHoriz

property flipVert

property fontColor

Alternative way of setting *foreColor*.

property foreColor

Sets both *itemColor* and *responseColor* to the same value

property foreColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property foreRGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type
DEPRECATED

formComplete()

Deprecated in version 2020.2. Please use the Form.complete property

getAlphaThreshold()

getAnchor()

getAutoDraw()

getAutoLog()

getBackColor()

getBackColorSpace()

getBackRGB()

getBackgroundColor()

getBorderColor()

getBorderColorSpace()

getBorderRGB()

getBorderWidth()

getColor()

getColorSpace()

getComplete()

getContrast()

getData()

Extracts form questions, response ratings and response times from Form items

Returns

A copy of the data as a list of dicts

Return type

list

getDepth()

getFillColor()

getFillColorSpace()

getFillRGB()

getFlip()

getFlipHoriz()

getFlipVert()

getFontColor()

getForeColor()

getForeColorSpace()

getForeRGB()

getHeight()

getItemColor()

getLineColor()

getLineColorSpace()

getLineRGB()

getLineWidth()

getMarkerColor()

getName()

getOpacity()

getOri()

getPos()

getRGB()

getResponseColor()

getScrollbarWidth()

getSize()

getStyle()

getUnits()

getValues()

getVertices()

getVerticesPix()

getWidth()

getWin()

get_borderPix()

property **height**

importItems(*items*)

Import items from csv or excel sheet and convert to list of dicts. Will also accept a list of dicts.

Note, for csv and excel files, 'options' must contain comma separated values, e.g., one, two, three. No parenthesis, or quotation marks required.

Parameters

items (*Excel or CSV file, list of dicts*) – Items used to populate the Form

Returns

A list of dicts, where each list entry is a dict containing all fields for a single Form item

Return type
List of dicts

`property itemColor`

Color of the text on form items

`knownStyles = {'dark': {'borderColor': None, 'fillColor': [-0.19, -0.19, -0.14], 'font': 'Noto Sans', 'itemColor': 'white', 'markerColor': [0.89, -0.35, -0.28], 'responseColor': 'white'}, 'light': {'borderColor': None, 'fillColor': [0.89, 0.89, 0.89], 'font': 'Noto Sans', 'itemColor': 'black', 'markerColor': [0.89, -0.35, -0.28], 'responseColor': 'black'}}`

`property lineColor`

Alternative way of setting *borderColor*.

`property lineColorSpace`

Deprecated, please use colorSpace to set color space for the entire object

`property lineRGB`

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type
DEPRECATED

`lineWidth`

`property markerColor`

Color of the marker on any sliders in this form

`name`

The name (*str*) of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = None your stimulus will be called "unnamed <type>", e.g. visual.TextStim(win) will be called "unnamed TextStim" in the logs.

`property opacity`

Determines how visible the stimulus is relative to background.

The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). *Operations* are supported. Precisely how this is used depends on the *Blend Mode*.

`ori`

The orientation of the stimulus (in degrees).

Should be a single value (*scalar*). *Operations* are supported.

Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and below zero values wrap appropriately.

`overlaps`(*polygon*)

Returns *True* if this stimulus intersects another one.

If *polygon* is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the *overlaps* method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

See coder demo, shapeContains.py

property pos

The position of the center of the stimulus in the stimulus *units*

value should be an *x,y-pair*. *Operations* are also supported.

Example:

```
stim.pos = (0.5, 0)  # Set slightly to the right of center
stim.pos += (0.5, -1)  # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2  # Move stim towards the center.
    Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:

```
from psychopy.tools.monitorunittools import posToPix
posPix = posToPix(stim)
```

reset()

Clear all responses and set all items to their initial values.

property responseColor

Color of the lines and text on form responses

property scrollbarWidth

Width of the scrollbar for this Form, in the spatial units of this Form. Can also be set as a *layout.Vector* object.

setAlphaThreshold(*value*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setAnchor(*value*, *log=None*)

setAutoDraw(*value*, *log=None*)

Sets autoDraw for Form and any responseCtrl contained within

setAutoLog(*value=True*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setBackColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setBackColorSpace(*value*, *log=None*, *operation=False*, *stealth=False*)

setBackRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setBackgroundColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setBorderColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setBorderColorSpace(*value*, *log=None*, *operation=False*, *stealth=False*)

setBorderRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setBorderWidth(*newWidth*, *operation=''*, *log=None*)

setColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setColorSpace(*value*, *log=None*, *operation=False*, *stealth=False*)

setContrast(*newContrast*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setDKL(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setDepth(*newDepth*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setFillColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for fillColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setFillColorSpace(*value*, *log=None*, *operation=False*, *stealth=False*)

setFillRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setFlip(*value*, *log=None*, *operation=False*, *stealth=False*)

setFlipHoriz(*value*, *log=None*, *operation=False*, *stealth=False*)

setFlipVert(*value*, *log=None*, *operation=False*, *stealth=False*)

setFontColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setForeColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for foreColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setForeColorSpace(*value*, *log=None*, *operation=False*, *stealth=False*)

setForeRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setHeight(*value*, *log=None*, *operation=False*, *stealth=False*)

setItemColor(*value*, *log=None*, *operation=False*, *stealth=False*)

setLMS(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setLineColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setLineColorSpace(*value*, *log=None*, *operation=False*, *stealth=False*)

setLineRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb* `setLineWidth`(*newWidth*, *operation=''*, *log=None*)

`setMarkerColor`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setName`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setOpacity`(*newOpacity*, *operation=''*, *log=None*)

Hard setter for opacity, allows the suppression of log messages and calls the update method

`setOri`(*newOri*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

`setPos`(*newPos*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

`setRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

`setResponseColor`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setScrollSpeed`(*items*, *multiplier=2*)

Set scroll speed of Form. Higher multiplier gives smoother, but slower scroll.

Parameters

- **items** (*list of dicts*) – Items used to populate the form
- **multiplier** (*int (default=2)*) – Number used to calculate scroll speed

Returns

Scroll speed, calculated using N items by multiplier

Return type

int

`setScrollbarWidth`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setSize`(*newSize*, *operation=''*, *units=None*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

`setStyle`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setUnits`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setValues`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setVertices`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setWidth`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setWin`(*value*, *log=None*, *operation=False*, *stealth=False*)

`property size`

The size (width, height) of the stimulus in the stimulus *units*

Value should be *x,y-pair*, *scalar* (applies to both dimensions) or None (resets to default). *Operations* are supported.

Sizes can be negative (causing a mirror-image reversal) and can extend beyond the window.

Example:

```
stim.size = 0.8  # Set size to (xsize, ysize) = (0.8, 0.8)
print(stim.size)  # Outputs array([0.8, 0.8])
stim.size += (0.5, -0.5)  # make wider and flatter: (1.3, 0.3)
```

Tip: if you can see the actual pixel range this corresponds to by looking at *stim._sizeRendered*

property style

property units

updateColors()

Placeholder method to update colours when set externally, for example updating the *pallette* attribute of a textbox

updateOpacity()

Placeholder method to update colours when set externally, for example updating the *pallette* attribute of a textbox.

property values

property vertices

property verticesPix

This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

property width

property win

The `Window` object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1  # stimulus will be drawn in win1
stim.draw()  # stimulus is now drawn to win1
stim.win = win2  # stimulus will be drawn in win2
stim.draw()  # it is now drawn in win2
win1.flip(waitBlanking=False)  # do not wait for next
             # monitor update
win2.flip()  # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

11.4.10 GratingStim

Attributes

GratingStim(win[, tex, mask, units, anchor, ...])	Stimulus object for drawing arbitrary bitmaps that can repeat (cycle) in either dimension.
GratingStim.win	The *Window* object in which the stimulus will be rendered by default.
GratingStim.tex	Texture to used on the stimulus as a grating (aka carrier).
GratingStim.mask	The alpha mask (forming the shape of the image).
GratingStim.units	
GratingStim.sf	Spatial frequency of the grating texture.
GratingStim.pos	The position of the center of the stimulus in the stimulus *units*
GratingStim.ori	The orientation of the stimulus (in degrees).
GratingStim.size	The size (width, height) of the stimulus in the stimulus *units*
GratingStim.contrast	A value that is simply multiplied by the color.
GratingStim.color	Alternative way of setting *foreColor*.
GratingStim.colorSpace	The name of the color space currently being used
GratingStim.opacity	Determines how visible the stimulus is relative to background.
GratingStim.interpolate	Whether to interpolate (linearly) the texture in the stimulus.
GratingStim.texRes	Power-of-two int.
GratingStim.name	The name (*str*) of the object to be using during logged messages about this stim.
GratingStim.autoLog	Whether every change in this stimulus should be auto logged.
GratingStim.draw([win])	Draw the stimulus in its relevant window.
GratingStim.autoDraw	Determines whether the stimulus should be automatically drawn on every frame flip.

Details

class psychopy.visual.**GratingStim**(*win*, *tex='sin'*, *mask='none'*, *units=None*, *anchor='center'*, *pos=(0.0, 0.0)*, *size=None*, *sf=None*, *ori=0.0*, *phase=(0.0, 0.0)*, *texRes=128*, *rgb=None*, *dkl=None*, *lms=None*, *color=(1.0, 1.0, 1.0)*, *colorSpace='rgb'*, *contrast=1.0*, *opacity=None*, *depth=0*, *rgbPedestal=(0.0, 0.0, 0.0)*, *interpolate=False*, *draggable=False*, *blendmode='avg'*, *name=None*, *autoLog=None*, *autoDraw=False*, *maskParams=None*)

Stimulus object for drawing arbitrary bitmaps that can repeat (cycle) in either dimension. This is a lazy-imported class, therefore import using full path *from psychopy.visual.grating import GratingStim* when inheriting from it.

One of the main stimuli for PsychoPy.

Formally *GratingStim* is just a texture behind an optional transparency mask (an 'alpha mask'). Both the texture and mask can be arbitrary bitmaps and their combination allows an enormous variety of stimuli to be drawn in realtime.

A *GratingStim* can be rotated scaled and shifted in position, its texture can be drifted in X and/or Y and it can have a spatial frequency in X and/or Y (for an image file that simply draws multiple copies in the patch).

Also since transparency can be controlled, two *GratingStim* objects can be combined (e.g. to form a plaid.)

Using GratingStim with images from disk (jpg, tif, png, …)

Ideally texture images to be rendered should be square with 'power-of-2' dimensions e.g. 16 x 16, 128 x 128. Any image that is not will be up-scaled (with linear interpolation) to the nearest such texture by PsychoPy. The size of the stimulus should be specified in the normal way using the appropriate units (deg, pix, cm, …). Be sure to get the aspect ratio the same as the image (if you don't want it stretched!).

Parameters

- **win** (*Window*) – Window this shape is being drawn to. The stimulus instance will allocate its required resources using that Windows context. In many cases, a stimulus instance cannot be drawn on different windows unless those windows share the same OpenGL context, which permits resources to be shared between them.
- **tex** (*str or None*) – Texture to use for the primary carrier. Values may be one of *'sin'*, *'sin'*, *'sqr'*, *'saw'*, *'tri'*, or *None*.
- **mask** (*str or None*) – Optional mask to control the shape of the grating. Values may be one of *'circle'*, *'sin'*, *'sqr'*, *'saw'*, *'tri'*, or *None*.
- **units** (*str*) – Units to use when drawing. This will affect how parameters and attributes *pos*, *size* and *radius* are interpreted.
- **anchor** (*str*) – Anchor string to specify the origin of the stimulus.
- **pos** (*array_like*) – Initial position (*x*, *y*) of the shape on-screen relative to the origin located at the center of the window or buffer in *units*. This can be updated after initialization by setting the *pos* property. The default value is *(0.0, 0.0)* which results in no translation.
- **size** (*array_like, float, int or None*) – Width and height of the shape as *(w, h)* or *[w, h]*. If a single value is provided, the width and height will be set to the same specified value. If *None* is specified, the *size* will be set with values passed to *width* and *height*.
- **sf** (*float*) – Spatial frequency for the grating. Values are dependent on the units in use to draw the stimuli.
- **ori** (*float*) – Initial orientation of the shape in degrees about its origin. Positive values will rotate the shape clockwise, while negative values will rotate counterclockwise. The default value for *ori* is 0.0 degrees.
- **phase** (*ArrayLike*) – Initial phase of the grating along the vertical and horizontal dimension *(x, y)*.
- **texRes** (*int*) – Resolution of the texture. The higher the resolutions, the less aliasing artifacts will be visible. However, this comes at the expense of higher video memory use. Power-of-two values are recommended (e.g. 256, 512, 1024, etc.)
- **color** (array_like, str, *Color* or None) – Sets both the initial *lineColor* and *fillColor* of the shape.
- **colorSpace** (*str*) – Sets the colorspace, changing how values passed to *lineColor* and *fillColor* are interpreted.
- **contrast** (*float*) – Contrast level of the shape (0.0 to 1.0). This value is used to modulate the contrast of colors passed to *lineColor* and *fillColor*.
- **opacity** (*float*) – Opacity of the shape. A value of 1.0 indicates fully opaque and 0.0 is fully transparent (therefore invisible). Values between 1.0 and 0.0 will result in colors being blended with objects in the background. This value affects the fill (*fillColor*) and outline (*lineColor*) colors of the shape.
- **depth** (*int*) – Depth layer to draw the shape when *autoDraw* is enabled. *DEPRECATED*

- **rgbPedestal** (*ArrayLike*) – Pedestal color *(r, g, b)*, presently unused.
- **interpolate** (*bool*) – Enable smoothing (anti-aliasing) when drawing shape outlines. This produces a smoother (less-pixelated) outline of the shape.
- **draggable** (*bool*) – Can this stimulus be dragged by a mouse click?
- **name** (*str*) – Optional name of the stimuli for logging.
- **autoLog** (*bool*) – Enable auto-logging of events associated with this stimuli. Useful for debugging and to track timing when used in conjunction with *autoDraw*.
- **autoDraw** (*bool*) – Enable auto drawing. When *True*, the stimulus will be drawn every frame without the need to explicitly call the `draw()` method.

Examples

Creating a circular grating with a sinusoidal pattern:

```
myGrat = GratingStim(tex='sin', mask='circle')
```

Create a 'Gabor':

```
myGabor = GratingStim(tex='sin', mask='gauss')
```

property RGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type
DEPRECATED

_calcPosRendered()

DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix.

_calcSizeRendered()

DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix

_createTexture(*tex*, *id*, *pixFormat*, *stim*, *res=128*, *maskParams=None*, *forcePOW2=True*, *dataType=None*, *wrapping=True*)

Create a new OpenGL 2D image texture.

Parameters

- **tex** (*Any*) – Texture data. Value can be anything that resembles image data.
- **id** (int or GLint) – Texture ID.
- **pixFormat** (GLenum or int) – Pixel format to use, values can be *GL_ALPHA* or *GL_RGB*.
- **stim** (*Any*) – Stimulus object using the texture.
- **res** (*int*) – The resolution of the texture (unless a bitmap image is used).
- **maskParams** (*dict or None*) – Additional parameters to configure the mask used with this texture.
- **forcePOW2** (*bool*) – Force the texture to be stored in a square memory area. For grating stimuli (anything that needs multiple cycles) *forcePOW2* should be set to be *True*. Otherwise the wrapping of the texture will not work.
- **dataType** (class:~*pyglet.gl.GLenum*, int or None) – None, *GL_UNSIGNED_BYTE*, *GL_FLOAT*. Only affects image files (numpy arrays will be float).

- **wrapping** (*bool*) – Enable wrapping of the texture. A texture will be set to repeat (or tile).

_drawLegacyGL(*win*)

Legacy draw function for older OpenGL versions.

_getDesiredRGB(*rgb*, *colorSpace*, *contrast*)

Convert color to RGB while adding contrast. Requires self.rgb, self.colorSpace and self.contrast

_getPolyAsRendered()

DEPRECATED. Return a list of vertices as rendered.

_selectWindow(*win*)

Switch drawing to the specified window. Calls the window's _setCurrent() method which handles the switch.

_set(*attrib*, *val*, *op=''*, *log=None*)

DEPRECATED since 1.80.04 + 1. Use setAttribute() and val2array() instead.

_updateList()

The user shouldn't need this method since it gets called after every call to .set() Chooses between using and not using shaders each call.

_updateListShaders()

The user shouldn't need this method since it gets called after every call to .set() Basically it updates the OpenGL representation of your stimulus if some parameter of the stimulus changes. Call it if you change a property manually rather than using the .set() command

_updateVertices()

Sets Stim.verticesPix and ._borderPix from pos, size, ori, flipVert, flipHoriz

alphaThreshold

Threshold for alpha values.

If the alpha value of a pixel is below this threshold, the pixel will be rejected (not drawn). This can be useful for creating a mask from an image with an alpha channel. The default value is 0.0, which means that no thresholding will be applied.

property **anchor**

autoDraw

Determines whether the stimulus should be automatically drawn on every frame flip.

Value should be: *True* or *False*. You do NOT need to set this on every frame flip!

autoLog

Whether every change in this stimulus should be auto logged.

Value should be: *True* or *False*. Set to *False* if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren't likely to be useful.

property **backColor**

Alternative way of setting fillColor

property **backColorSpace**

Deprecated, please use colorSpace to set color space for the entire object.

property backRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type
DEPRECATED

property backgroundColor

Alternative way of setting fillColor

blendmode

The OpenGL mode in which the stimulus is draw

Can the ‘avg’ or ‘add’. Average (avg) places the new stimulus over the old one with a transparency given by its opacity. Opaque stimuli will hide other stimuli transparent stimuli won’t. Add performs the arithmetic sum of the new stimulus and the ones already present.

property borderColor

property borderColorSpace

Deprecated, please use colorSpace to set color space for the entire object

property borderRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type
DEPRECATED

borderWidth

clearTextures()

Clear all textures associated with the stimulus.

As of v1.61.00 this is called automatically during garbage collection of your stimulus, so doesn’t need calling explicitly by the user.

property color

Alternative way of setting *foreColor*.

property colorSpace

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                       color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

contains(*x*, *y=None*, *units=None*)

Returns True if a point x,y is inside the stimulus' border.

Can accept variety of input options:

- two separate args, x and y
- one arg (list, tuple or array) containing two vals (x,y)
- **an object with a getPos() method that returns x,y, such**
  as a `Mouse`.

Returns *True* if the point is within the area defined either by its *border* attribute (if one defined), or its *vertices* attribute if there is no .border. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the *contains* method; the extent of the stimulus is determined purely by the size, position (pos), and orientation (ori) settings (and by the vertices for shape stimuli).

See Coder demos: shapeContains.py See Coder demos: shapeContains.py

property contrast

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged).
Operations supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast =  1.0  # unchanged contrast
stim.contrast =  0.5  # decrease contrast
stim.contrast =  0.0  # uniform, no contrast
stim.contrast = -0.5  # slightly inverted
stim.contrast = -1.0  # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast =  1.2  # increases contrast
stim.contrast = -1.2  # inverts with increased contrast
```

depth

DEPRECATED, depth is now controlled simply by drawing order.

doDragging()

If this stimulus is draggable, do the necessary actions on a frame flip to drag it.

draggable

Can this stimulus be dragged by a mouse click?

draw(*win=None*)

Draw the stimulus in its relevant window.

You must call this method after every *MyWin.flip()* if you want the stimulus to appear on that frame and then update the screen again.

Parameters

win (*~psychopy.visual.Window* or *None*) – Window to draw the stimulus to. Context sharing must be enabled if any other window beside the one specified during creation of this stimulus is specified.

property fillColor

Set the fill color for the shape.

property fillColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property fillRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type

DEPRECATED

property flip

1x2 array for flipping vertices along each axis; set as True to flip or False to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (*._flip*) will give an array of 1s and -1s with which to multiply vertices.

property flipHoriz

property flipVert

property fontColor

Alternative way of setting *foreColor*.

property foreColor

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names <http://www.w3schools.com/html/html_colornames.asp>* can be used.
- *Colors by hex value*
- **numerically: (scalar or triplet) for DKL, RGB or**
  other *Color spaces*. For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue'  # (the case is actually ignored)
stim.color = '#DDA0DD'  # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0]  # if stim.colorSpace='rgb':
                # a red color in rgb space
stim.color = [0.0, 45.0, 1.0]  # if stim.colorSpace='dkl':
                # DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255]  # if stim.colorSpace='rgb255':
                # a blue stimulus using rgb255 space
stim.color = 255  # interpreted as (255, 255, 255)
                  # which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1]  # increment all guns by 1 value
stim.color *= -1  # multiply the color by -1 (which in this
                    # space inverts the contrast)
stim.color *= [0.5, 0, 1]  # decrease red, remove green, keep blue
```

You can use *setColor* if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

property foreColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property foreRGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type
DEPRECATED

getAlphaThreshold()

getAnchor()

getAutoDraw()

getAutoLog()

getBackColor()

getBackColorSpace()

getBackRGB()

getBackgroundColor()

getBlendmode()

getBorderColor()

getBorderColorSpace()

getBorderRGB()

getBorderWidth()

getColor()

getColorSpace()

getContrast()

getDepth()

getDraggable()

getFillColor()

getFillColorSpace()

getFillRGB()

getFlip()

getFlipHoriz()

getFlipVert()

getFontColor()

getForeColor()

getForeColorSpace()

getForeRGB()

getHeight()

getInterpolate()

getLineColor()

getLineColorSpace()

getLineRGB()

getLineWidth()

getMask()

getMaskParams()

getName()

getOpacity()

getOri()

getPhase()

getPos()

getRGB()

getSf()

getSize()

getTex()

getTexRes()

getUnits()

getVertices()

getVerticesPix()

`getWidth()`

`getWin()`

`get_borderPix()`

`property height`

`interpolate`

Whether to interpolate (linearly) the texture in the stimulus.

If set to False then nearest neighbour will be used when needed, otherwise some form of interpolation will be used.

`isDragging = False`

`property lineColor`

Alternative way of setting *borderColor*.

`property lineColorSpace`

Deprecated, please use colorSpace to set color space for the entire object

`property lineRGB`

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type
DEPRECATED

`lineWidth`

`mask`

The alpha mask (forming the shape of the image).

This can be one of various options:

- 'circle', 'gauss', 'raisedCos', 'cross'
- **None** (resets to default)
- the name of an image file (most formats supported)
- a numpy array (1xN or NxN) ranging -1:1

`maskParams`

Various types of input. Default to *None*.

This is used to pass additional parameters to the mask if those are needed.

- **For 'gauss' mask, pass dict {'sd': 5} to control**
  standard deviation.
- **For the 'raisedCos' mask, pass a dict: {'fringeWidth':0.2},**
  where 'fringeWidth' is a parameter (float, 0-1), determining the proportion of the patch that will be blurred by the raised cosine edge.

`name`

The name (*str*) of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = None your stimulus will be called "unnamed <type>", e.g. visual.TextStim(win) will be called "unnamed TextStim" in the logs.

property opacity

Determines how visible the stimulus is relative to background.

The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). *Operations* are supported. Precisely how this is used depends on the *Blend Mode*.

ori

The orientation of the stimulus (in degrees).

Should be a single value (*scalar*). *Operations* are supported.

Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and below zero values wrap appropriately.

overlaps(*polygon*)

Returns *True* if this stimulus intersects another one.

If *polygon* is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the *overlaps* method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

See coder demo, shapeContains.py

phase

Phase of the stimulus in each dimension of the texture.

Should be an *x,y-pair* or *scalar*

NB phase has modulus 1 (rather than 360 or 2*pi) This is a little unconventional but has the nice effect that setting phase=t*n drifts a stimulus at *n* Hz.

property pos

The position of the center of the stimulus in the stimulus *units*

value should be an *x,y-pair*. *Operations* are also supported.

Example:

```
stim.pos = (0.5, 0)  # Set slightly to the right of center
stim.pos += (0.5, -1)  # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2  # Move stim towards the center.
    Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:

```
from psychopy.tools.monitorunittools import posToPix
posPix = posToPix(stim)
```

setAlphaThreshold(*value*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setAnchor(*value*, *log=None*)

setAutoDraw(*value*, *log=None*)

Sets autoDraw. Usually you can use 'stim.attribute = value' syntax instead, but use this method to suppress the log message.

setAutoLog(*value=True*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setBackColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setBackColorSpace(*value*, *log=None*, *operation=False*, *stealth=False*)

setBackRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setBackgroundColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setBlendmode(*value*, *log=None*)

DEPRECATED. Use 'stim.parameter = value' syntax instead

setBorderColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setBorderColorSpace(*value*, *log=None*, *operation=False*, *stealth=False*)

setBorderRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setBorderWidth(*newWidth*, *operation=''*, *log=None*)

setColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setColorSpace(*value*, *log=None*, *operation=False*, *stealth=False*)

setContrast(*newContrast*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setDKL(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setDepth(*newDepth*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setDraggable(*value*, *log=None*, *operation=False*, *stealth=False*)

setFillColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for fillColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setFillColorSpace(*value*, *log=None*, *operation=False*, *stealth=False*)

setFillRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setFlip(*value*, *log=None*, *operation=False*, *stealth=False*) `setFlipHoriz`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setFlipVert`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setFontColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

`setForeColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for foreColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

`setForeColorSpace`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setForeRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

`setHeight`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setInterpolate`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setLMS`(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

`setLineColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

`setLineColorSpace`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setLineRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

`setLineWidth`(*newWidth*, *operation=''*, *log=None*)

`setMask`(*value*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

`setMaskParams`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setName`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setOpacity`(*newOpacity*, *operation=''*, *log=None*)

Hard setter for opacity, allows the suppression of log messages and calls the update method

`setOri`(*newOri*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

`setPhase`(*value*, *operation=''*, *log=None*)

DEPRECATED. Use 'stim.parameter = value' syntax instead

`setPos`(*newPos*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

`setRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

`setSF`(*value*, *operation=''*, *log=None*)

DEPRECATED. Use 'stim.parameter = value' syntax instead

setSf(*value*, *log=None*, *operation=False*, *stealth=False*)

setSize(*newSize*, *operation=''*, *units=None*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setTex(*value*, *log=None*)

DEPRECATED. Use 'stim.parameter = value' syntax instead

setTexRes(*value*, *log=None*, *operation=False*, *stealth=False*)

setUnits(*value*, *log=None*, *operation=False*, *stealth=False*)

setVertices(*value*, *log=None*, *operation=False*, *stealth=False*)

setWidth(*value*, *log=None*, *operation=False*, *stealth=False*)

setWin(*value*, *log=None*, *operation=False*, *stealth=False*)

sf

Spatial frequency of the grating texture.

Should be a *x,y-pair* or *scalar* or None. If *units* == 'deg' or 'cm' units are in cycles per deg or cm as appropriate. If *units* == 'norm' then sf units are in cycles per stimulus (and so SF scales with stimulus size). If texture is an image loaded from a file then sf=None defaults to 1/stimSize to give one cycle of the image.

property size

The size (width, height) of the stimulus in the stimulus *units*

Value should be *x,y-pair*, *scalar* (applies to both dimensions) or None (resets to default). *Operations* are supported.

Sizes can be negative (causing a mirror-image reversal) and can extend beyond the window.

Example:

```python
stim.size = 0.8  # Set size to (xsize, ysize) = (0.8, 0.8)
print(stim.size)  # Outputs array([0.8, 0.8])
stim.size += (0.5, -0.5)  # make wider and flatter: (1.3, 0.3)
```

Tip: if you can see the actual pixel range this corresponds to by looking at *stim._sizeRendered*

tex

Texture to used on the stimulus as a grating (aka carrier).

This can be one of various options:

- **'sin'**,'sqr', 'saw', 'tri', None (resets to default)
- the name of an image file (most formats supported)
- a numpy array (1xN or NxN) ranging -1:1

If specifying your own texture using an image or numpy array you should ensure that the image has square power-of-two dimensions (e.g. 256 x 256). If not then PsychoPy will up-sample your stimulus to the next larger power of two.

texRes

Power-of-two int. Sets the resolution of the mask and texture. texRes is overridden if an array or image is provided as mask.

Operations supported.

property units

updateColors()

Placeholder method to update colours when set externally, for example updating the *pallette* attribute of a textbox

updateOpacity()

Placeholder method to update colours when set externally, for example updating the *pallette* attribute of a textbox.

property vertices

property verticesPix

This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

property width

property win

The `Window` object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1  # stimulus will be drawn in win1
stim.draw()  # stimulus is now drawn to win1
stim.win = win2  # stimulus will be drawn in win2
stim.draw()  # it is now drawn in win2
win1.flip(waitBlanking=False)  # do not wait for next
             # monitor update
win2.flip()  # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

11.4.11 Helper functions

psychopy.visual.helpers.**pointInPolygon**(*x*, *y*, *poly*)

Determine if a point is inside a polygon; returns True if inside.

(*x*, *y*) is the point to test. *poly* is a list of 3 or more vertices as (x,y) pairs. If given an object, such as a *ShapeStim*, will try to use its vertices and position as the polygon.

Same as the *.contains()* method elsewhere.

`psychopy.visual.helpers.`**`polygonsOverlap`**`(`*poly1*, *poly2*`)`

Determine if two polygons intersect; can fail for very pointy polygons.

Accepts two polygons, as lists of vertices (x,y) pairs. If given an object with with (vertices + pos), will try to use that as the polygon.

Checks if any vertex of one polygon is inside the other polygon. Same as the *.overlaps()* method elsewhere.

Notes

We implement special handling for the *Line* stimulus as it is not a proper polygon. We do not check for class instances because this would require importing of *visual.Line*, creating a circular import. Instead, we assume that a "polygon" with only two vertices is meant to specify a line. Pixels between the endpoints get interpolated before testing for overlap.

`psychopy.visual.helpers.`**`groupFlipVert`**`(`*flipList*, *yReflect=0*`)`

Reverses the vertical mirroring of all items in list `flipList`.

Reverses the .flipVert status, vertical (y) positions, and angular rotation (.ori). Flipping preserves the relations among the group's visual elements. The parameter `yReflect` is the y-value of an imaginary horizontal line around which to reflect the items; default = 0 (screen center).

Typical usage is to call once prior to any display; call again to un-flip. Can be called with a list of all stim to be presented in a given routine.

Will flip a) all psychopy.visual.xyzStim that have a setFlipVert method, b) the y values of .vertices, and c) items in n x 2 lists that are mutable (i.e., list, np.array, no tuples): [[x1, y1], [x2, y2], . . .]

11.4.12 ImageStim

As of version 1.79.00 *some* of the properties for this stimulus can be set using the syntax:

```
stim.pos = newPos
```

others need to be set with the older syntax:

```
stim.setImage(newImage)
```

Attributes

ImageStim(win[, image, mask, units, pos, ...])	Display an image on a *psychopy.visual.Window*
ImageStim.win	The *Window* object in which the stimulus will be rendered by default.
ImageStim.setImage(value[, log])	Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.
ImageStim.setMask(value[, log])	Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.
ImageStim.units	
ImageStim.pos	The position of the center of the stimulus in the stimulus *units*
ImageStim.ori	The orientation of the stimulus (in degrees).
ImageStim.size	The size (width, height) of the stimulus in the stimulus *units*
ImageStim.contrast	A value that is simply multiplied by the color.
ImageStim.color	Alternative way of setting *foreColor*.
ImageStim.colorSpace	The name of the color space currently being used
ImageStim.opacity	Determines how visible the stimulus is relative to background.
ImageStim.interpolate	Whether to interpolate (linearly) the texture in the stimulus.
ImageStim.contains(x[, y, units])	Returns True if a point x,y is inside the stimulus' border.
ImageStim.overlaps(polygon)	Returns *True* if this stimulus intersects another one.
ImageStim.name	The name (*str*) of the object to be using during logged messages about this stim.
ImageStim.autoLog	Whether every change in this stimulus should be auto logged.
ImageStim.draw([win])	Draw the stimulus on the window.
ImageStim.autoDraw	Determines whether the stimulus should be automatically drawn on every frame flip.
ImageStim.clearTextures()	Clear all textures associated with the stimulus.

Details

class psychopy.visual.**ImageStim**(*win*, *image=None*, *mask=None*, *units=''*, *pos=(0.0, 0.0)*, *size=None*, *anchor='center'*, *ori=0.0*, *color=(1.0, 1.0, 1.0)*, *colorSpace='rgb'*, *contrast=1.0*, *opacity=None*, *depth=0*, *interpolate=False*, *draggable=False*, *flipHoriz=False*, *flipVert=False*, *texRes=128*, *name=None*, *autoLog=None*, *maskParams=None*)

Display an image on a *psychopy.visual.Window*

property RGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type

DEPRECATED

_calcPosRendered()

DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix.

`_calcSizeRendered()`

DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix

`_createTexture`(*tex*, *id*, *pixFormat*, *stim*, *res=128*, *maskParams=None*, *forcePOW2=True*, *dataType=None*, *wrapping=True*)

Create a new OpenGL 2D image texture.

Parameters

- **tex** (*Any*) – Texture data. Value can be anything that resembles image data.
- **id** (int or `GLint`) – Texture ID.
- **pixFormat** (`GLenum` or int) – Pixel format to use, values can be *GL_ALPHA* or *GL_RGB*.
- **stim** (*Any*) – Stimulus object using the texture.
- **res** (*int*) – The resolution of the texture (unless a bitmap image is used).
- **maskParams** (*dict or None*) – Additional parameters to configure the mask used with this texture.
- **forcePOW2** (*bool*) – Force the texture to be stored in a square memory area. For grating stimuli (anything that needs multiple cycles) *forcePOW2* should be set to be *True*. Otherwise the wrapping of the texture will not work.
- **dataType** (class:*~pyglet.gl.GLenum*, int or None) – None, *GL_UNSIGNED_BYTE*, *GL_FLOAT*. Only affects image files (numpy arrays will be float).
- **wrapping** (*bool*) – Enable wrapping of the texture. A texture will be set to repeat (or tile).

`_drawLegacyGL`(*win*)

Legacy draw routine.

`_getDesiredRGB`(*rgb*, *colorSpace*, *contrast*)

Convert color to RGB while adding contrast. Requires self.rgb, self.colorSpace and self.contrast

`_getPolyAsRendered()`

DEPRECATED. Return a list of vertices as rendered.

`_movieFrameToTexture`(*movieSrc*)

Convert a movie frame to a texture and use it.

This method is used internally to copy pixel data from a camera object into a texture. This enables the *ImageStim* to be used as a ‘viewfinder’ of sorts for the camera to view a live video stream on a window.

Parameters

movieSrc (*~psychopy.hardware.camera.Camera*) – Movie source object.

`_selectWindow`(*win*)

Switch drawing to the specified window. Calls the window’s _setCurrent() method which handles the switch.

`_set`(*attrib*, *val*, *op=''*, *log=None*)

DEPRECATED since 1.80.04 + 1. Use setAttribute() and val2array() instead.

`_updateList()`

The user shouldn’t need this method since it gets called after every call to .set() Chooses between using and not using shaders each call.

`_updateListShaders()`

The user shouldn't need this method since it gets called after every call to .set() Basically it updates the OpenGL representation of your stimulus if some parameter of the stimulus changes. Call it if you change a property manually rather than using the .set() command

`_updateVertices()`

Sets Stim.verticesPix and ._borderPix from pos, size, ori, flipVert, flipHoriz

`alphaThreshold`

Threshold for alpha values.

If the alpha value of a pixel is below this threshold, the pixel will be rejected (not drawn). This can be useful for creating a mask from an image with an alpha channel. The default value is 0.0, which means that no thresholding will be applied.

`property anchor`

`property aspectRatio`

Aspect ratio of original image, before taking into account the *.size* attribute of this object.

returns :

Aspect ratio as a (w, h) tuple, simplified using the smallest common denominator (e.g. 1080x720 pixels becomes (3, 2))

`autoDraw`

Determines whether the stimulus should be automatically drawn on every frame flip.

Value should be: *True* or *False*. You do NOT need to set this on every frame flip!

`autoLog`

Whether every change in this stimulus should be auto logged.

Value should be: *True* or *False*. Set to *False* if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren't likely to be useful.

`property backColor`

Alternative way of setting fillColor

`property backColorSpace`

Deprecated, please use colorSpace to set color space for the entire object.

`property backRGB`

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type

DEPRECATED

`property backgroundColor`

Alternative way of setting fillColor

`property borderColor`

`property borderColorSpace`

Deprecated, please use colorSpace to set color space for the entire object

`property borderRGB`

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type
DEPRECATED

`borderWidth`

`clearTextures()`

Clear all textures associated with the stimulus.

As of v1.61.00 this is called automatically during garbage collection of your stimulus, so doesn't need calling explicitly by the user.

`property color`

Alternative way of setting *foreColor*.

`property colorSpace`

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                       color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

`contains`(*x*, *y=None*, *units=None*)

Returns True if a point x,y is inside the stimulus' border.

Can accept variety of input options:

- two separate args, x and y
- one arg (list, tuple or array) containing two vals (x,y)
- **an object with a getPos() method that returns x,y, such**
  as a `Mouse`.

Returns *True* if the point is within the area defined either by its *border* attribute (if one defined), or its *vertices* attribute if there is no .border. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the *contains* method; the extent of the stimulus is determined purely by the size, position (pos), and orientation (ori) settings (and by the vertices for shape stimuli).

See Coder demos: shapeContains.py See Coder demos: shapeContains.py

`property contrast`

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged).
Operations supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast =  1.0  # unchanged contrast
stim.contrast =  0.5  # decrease contrast
stim.contrast =  0.0  # uniform, no contrast
stim.contrast = -0.5  # slightly inverted
stim.contrast = -1.0  # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast =  1.2  # increases contrast
stim.contrast = -1.2  # inverts with increased contrast
```

depth

DEPRECATED, depth is now controlled simply by drawing order.

doDragging()

If this stimulus is draggable, do the necessary actions on a frame flip to drag it.

draggable

Can this stimulus be dragged by a mouse click?

draw(*win=None*)

Draw the stimulus on the window.

Parameters
win (*~psychopy.visual.Window*, optional) – The window to draw the stimulus on. If None, the stimulus will be drawn on the window that was passed to the constructor.

property fillColor

Set the fill color for the shape.

property fillColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property fillRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type
DEPRECATED

property flip

1x2 array for flipping vertices along each axis; set as True to flip or False to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (*._flip*) will give an array of 1s and -1s with which to multiply vertices.

property flipHoriz

property flipVert

property fontColor

Alternative way of setting *foreColor*.

property foreColor

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names <http://www.w3schools.com/html/html_colornames.asp>* can be used.
- *Colors by hex value*
- **numerically: (scalar or triplet) for DKL, RGB or**
  other *Color spaces*. For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue'  # (the case is actually ignored)
stim.color = '#DDA0DD'  # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0]  # if stim.colorSpace='rgb':
                # a red color in rgb space
stim.color = [0.0, 45.0, 1.0]  # if stim.colorSpace='dkl':
                # DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255]  # if stim.colorSpace='rgb255':
                # a blue stimulus using rgb255 space
stim.color = 255  # interpreted as (255, 255, 255)
                  # which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1]  # increment all guns by 1 value
stim.color *= -1  # multiply the color by -1 (which in this
                    # space inverts the contrast)
stim.color *= [0.5, 0, 1]  # decrease red, remove green, keep blue
```

You can use *setColor* if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

property foreColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property foreRGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type
DEPRECATED

getAlphaThreshold()

getAnchor()

getAspectRatio()

getAutoDraw()

getAutoLog()

getBackColor()

getBackColorSpace()

getBackRGB()

getBackgroundColor()

getBorderColor()

getBorderColorSpace()

getBorderRGB()

getBorderWidth()

getColor()

getColorSpace()

getContrast()

getDepth()

getDraggable()

getFillColor()

getFillColorSpace()

getFillRGB()

getFlip()

getFlipHoriz()

getFlipVert()

getFontColor()

getForeColor()

getForeColorSpace()

getForeRGB()

getHeight()

getImage()

getInterpolate()

getLineColor()

getLineColorSpace()

getLineRGB()

getLineWidth()

getMask()

getMaskParams()

getName()

getOpacity()

getOri()

getPos()

getRGB()

getSize()

getTexRes()

getUnits()

getVertices()

getVerticesPix()

getWidth()

getWin()

get_borderPix()

property height

image

The image file to be presented (most formats supported).

This can be a path-like object to an image file, or a numpy array of shape [H, W, C] where C are channels. The third dim will usually have length 1 (defining an intensity-only image), 3 (defining an RGB image) or 4 (defining an RGBA image).

If passing a numpy array to the image attribute, the size attribute of ImageStim must be set explicitly.

interpolate

Whether to interpolate (linearly) the texture in the stimulus.

If set to False then nearest neighbour will be used when needed, otherwise some form of interpolation will be used.

isDragging = False

property lineColor

Alternative way of setting *borderColor*.

`property lineColorSpace`

Deprecated, please use colorSpace to set color space for the entire object

`property lineRGB`

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type
DEPRECATED

`lineWidth`

`mask`

The alpha mask that can be used to control the outer shape of the stimulus

- **None**, 'circle', 'gauss', 'raisedCos'
- or the name of an image file (most formats supported)
- or a numpy array (1xN or NxN) ranging -1:1

`maskParams`

Various types of input. Default to *None*.

This is used to pass additional parameters to the mask if those are needed.

- **For 'gauss' mask, pass dict {'sd': 5} to control**
  standard deviation.
- **For the 'raisedCos' mask, pass a dict: {'fringeWidth':0.2},**
  where 'fringeWidth' is a parameter (float, 0-1), determining the proportion of the patch that will be blurred by the raised cosine edge.

`name`

The name (*str*) of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = None your stimulus will be called "unnamed <type>", e.g. visual.TextStim(win) will be called "unnamed TextStim" in the logs.

`property opacity`

Determines how visible the stimulus is relative to background.

The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). *Operations* are supported. Precisely how this is used depends on the *Blend Mode*.

`ori`

The orientation of the stimulus (in degrees).

Should be a single value (*scalar*). *Operations* are supported.

Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and below zero values wrap appropriately.

`overlaps`(*polygon*)

Returns *True* if this stimulus intersects another one.

If *polygon* is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the *overlaps* method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

See coder demo, shapeContains.py

property pos

The position of the center of the stimulus in the stimulus *units*

value should be an *x,y-pair*. *Operations* are also supported.

Example:

```
stim.pos = (0.5, 0)  # Set slightly to the right of center
stim.pos += (0.5, -1)  # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2  # Move stim towards the center.
    Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:

```
from psychopy.tools.monitorunittools import posToPix
posPix = posToPix(stim)
```

setAlphaThreshold(*value*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setAnchor(*value*, *log=None*)

setAutoDraw(*value*, *log=None*)

Sets autoDraw. Usually you can use 'stim.attribute = value' syntax instead, but use this method to suppress the log message.

setAutoLog(*value=True*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setBackColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setBackColorSpace(*value*, *log=None*, *operation=False*, *stealth=False*)

setBackRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setBackgroundColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setBorderColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setBorderColorSpace(*value*, *log=None*, *operation=False*, *stealth=False*)

setBorderRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setBorderWidth(*newWidth*, *operation=''*, *log=None*)

setColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setColorSpace(*value*, *log=None*, *operation=False*, *stealth=False*)

setContrast(*newContrast*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setDKL(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setDepth(*newDepth*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setDraggable(*value*, *log=None*, *operation=False*, *stealth=False*)

setFillColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for fillColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setFillColorSpace(*value*, *log=None*, *operation=False*, *stealth=False*)

setFillRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setFlip(*value*, *log=None*, *operation=False*, *stealth=False*)

setFlipHoriz(*value*, *log=None*, *operation=False*, *stealth=False*)

setFlipVert(*value*, *log=None*, *operation=False*, *stealth=False*)

setFontColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setForeColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for foreColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setForeColorSpace(*value*, *log=None*, *operation=False*, *stealth=False*)

setForeRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setHeight(*value*, *log=None*, *operation=False*, *stealth=False*)

setImage(*value*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setInterpolate(*value*, *log=None*, *operation=False*, *stealth=False*)

setLMS(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setLineColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setLineColorSpace(*value*, *log=None*, *operation=False*, *stealth=False*)

setLineRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setLineWidth(*newWidth*, *operation=''*, *log=None*)

setMask(*value*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setMaskParams(*value*, *log=None*, *operation=False*, *stealth=False*)

setName(*value*, *log=None*, *operation=False*, *stealth=False*)

setOpacity(*newOpacity*, *operation=''*, *log=None*)

Hard setter for opacity, allows the suppression of log messages and calls the update method

setOri(*newOri*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setPos(*newPos*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setSize(*newSize*, *operation=''*, *units=None*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setTexRes(*value*, *log=None*, *operation=False*, *stealth=False*)

setUnits(*value*, *log=None*, *operation=False*, *stealth=False*)

setVertices(*value*, *log=None*, *operation=False*, *stealth=False*)

setWidth(*value*, *log=None*, *operation=False*, *stealth=False*)

setWin(*value*, *log=None*, *operation=False*, *stealth=False*)

property size

The size (width, height) of the stimulus in the stimulus *units*

Value should be *x,y-pair*, *scalar* (applies to both dimensions) or None (resets to default). *Operations* are supported.

Sizes can be negative (causing a mirror-image reversal) and can extend beyond the window.

Example:

```python
stim.size = 0.8  # Set size to (xsize, ysize) = (0.8, 0.8)
print(stim.size)  # Outputs array([0.8, 0.8])
stim.size += (0.5, -0.5)  # make wider and flatter: (1.3, 0.3)
```

Tip: if you can see the actual pixel range this corresponds to by looking at *stim._sizeRendered*

texRes

Power-of-two int. Sets the resolution of the mask and texture. texRes is overridden if an array or image is provided as mask.

Operations supported.

property units

updateColors()

Placeholder method to update colours when set externally, for example updating the *pallette* attribute of a textbox

updateOpacity()

Placeholder method to update colours when set externally, for example updating the *pallette* attribute of a textbox.

property vertices

property verticesPix

This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

property width

property win

The *Window* object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1  # stimulus will be drawn in win1
stim.draw()  # stimulus is now drawn to win1
stim.win = win2  # stimulus will be drawn in win2
stim.draw()  # it is now drawn in win2
win1.flip(waitBlanking=False)  # do not wait for next
             # monitor update
win2.flip()  # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

11.4.13 LightSource

Attributes

LightSource(win[, pos, diffuseColor, ...])	Class for representing a light source in a scene.

Details

class psychopy.visual.**LightSource**(*win*, *pos=(0.0, 0.0, 0.0)*, *diffuseColor=(1.0, 1.0, 1.0)*, *specularColor=(1.0, 1.0, 1.0)*, *ambientColor=(0.0, 0.0, 0.0)*, *colorSpace='rgb'*, *contrast=1.0*, *lightType='point'*, *attenuation=(1, 0, 0)*)

Class for representing a light source in a scene. This is a lazy-imported class, therefore import using full path *from psychopy.visual.stim3d import LightSource* when inheriting from it.

Only point and directional lighting is supported by this object for now. The ambient color of the light source contributes to the scene ambient color defined by *ambientLight*.

⚠ **Warning**

This class is experimental and may result in undefined behavior.

Parameters

- **win** (*~psychopy.visual.Window*) – Window associated with this light source.
- **pos** (*array_like*) – Position of the light source (x, y, z, w). If *w=1.0* the light will be a point source and *x*, *y*, and *z* is the position in the scene. If *w=0.0*, the light source will be directional and *x*, *y*, and *z* will define the vector pointing to the direction the light source is coming from. For instance, a vector of (0, 1, 0, 0) will indicate that a light source is coming from above.
- **diffuseColor** (*array_like*) – Diffuse light color.
- **specularColor** (*array_like*) – Specular light color.
- **ambientColor** (*array_like*) – Ambient light color.
- **colorSpace** (*str or None*) – Colorspace for diffuse, specular, and ambient color components.
- **contrast** (*float*) – Contrast of the lighting color components. This acts as a 'gain' factor which scales color values. Must be between 0.0 and 1.0.
- **attenuation** (*array_like*) – Values for the constant, linear, and quadratic terms of the lighting attenuation formula. Default is (1, 0, 0) which results in no attenuation.

`property ambientColor`

Ambient color of the light source (*psychopy.color.Color*, *ArrayLike* or None).

The ambient color component is used to simulate indirect lighting caused by the light source. For instance, light bouncing off adjacent surfaces or atmospheric scattering if the light source is a sun. This is independent of the global ambient color.

`property ambientRGB`

Ambient RGB1 color of the material. This value is passed to OpenGL.

`property attenuation`

Values for the constant, linear, and quadratic terms of the lighting attenuation formula.

`property colorSpace`

The name of the color space currently being used (*str* or *None*).

For strings and hex values this is not needed. If *None* the default *colorSpace* for the stimulus is used (defined during initialisation).

Please note that changing *colorSpace* does not change stimulus parameters. Thus, you usually want to specify *colorSpace* before setting the color.

`property contrast`

A value that is simply multiplied by the color (*float*).

This may be used to adjust the gain of the light source. This is applied to all lighting color components.

Examples

Basic usage:

```
stim.contrast =  1.0  # unchanged contrast
stim.contrast =  0.5  # decrease contrast
stim.contrast =  0.0  # uniform, no contrast
stim.contrast = -0.5  # slightly inverted
stim.contrast = -1.0  # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast =  1.2  # increases contrast
stim.contrast = -1.2  # inverts with increased contrast
```

property diffuseColor

Diffuse color for the light source (*psychopy.color.Color*, *ArrayLike* or None).

property diffuseRGB

Diffuse RGB1 color of the material. This value is passed to OpenGL.

property lightType

Type of light source, can be 'point' or 'directional'.

property pos

Position of the light source in the scene in scene units.

setAmbientColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Set the ambient color for the light source.

Use this function if you wish to supress logging or apply operations on the color component.

Parameters

- **color** (ArrayLike or ~*psychopy.colors.Color*) – Color to set as the ambient component of the light source.
- **colorSpace** (*str or None*) – Colorspace to use. This is only used to set the color, the value of *ambientColor* after setting uses the color space of the object.
- **operation** (*str*) – Operation string.
- **log** (*bool or None*) – Enable logging.

setDiffuseColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Set the diffuse color for the light source. Use this function if you wish to supress logging or apply operations on the color component.

Parameters

- **color** (ArrayLike or ~*psychopy.colors.Color*) – Color to set as the diffuse component of the light source.
- **colorSpace** (*str or None*) – Colorspace to use. This is only used to set the color, the value of *diffuseColor* after setting uses the color space of the object.
- **operation** (*str*) – Operation string.
- **log** (*bool or None*) – Enable logging.

setSpecularColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Set the diffuse color for the light source. Use this function if you wish to supress logging or apply operations on the color component.

Parameters

- **color** (ArrayLike or *~psychopy.colors.Color*) – Color to set as the specular component of the light source.
- **colorSpace** (*str or None*) – Colorspace to use. This is only used to set the color, the value of *diffuseColor* after setting uses the color space of the object.
- **operation** (*str*) – Operation string.
- **log** (*bool or None*) – Enable logging.

property specularColor

Specular color of the light source (*psychopy.color.Color*, *ArrayLike* or None).

property specularRGB

Specular RGB1 color of the material. This value is passed to OpenGL.

11.4.14 psychopy.visual.Line

Stimulus class for drawing lines.

Overview

Details

class psychopy.visual.line.**Line**(*win*, *start=(-0.5, -0.5)*, *end=(0.5, 0.5)*, *units=None*, *lineWidth=1.5*, *lineColor='white'*, *colorSpace='rgb'*, *pos=(0, 0)*, *size=1.0*, *anchor='center'*, *ori=0.0*, *opacity=None*, *contrast=1.0*, *depth=0*, *interpolate=True*, *draggable=False*, *name=None*, *autoLog=None*, *autoDraw=False*, *color=undefined*, *fillColor=undefined*, *lineColorSpace=undefined*, *lineRGB=undefined*, *fillRGB=undefined*)

Creates a Line between two points. This is a lazy-imported class, therefore import using full path *from psychopy.visual.line import Line* when inheriting from it.

Line accepts all input parameters, that ShapeStim accepts, except for *vertices*, *closeShape* and *fillColor*.

(New in version 1.72.00)

Parameters

- **win** (*Window*) – Window this line is being drawn to. The stimulus instance will allocate its required resources using that Windows context. In many cases, a stimulus instance cannot be drawn on different windows unless those windows share the same OpenGL context, which permits resources to be shared between them.
- **start** (*array_like*) – Coordinate *(x, y)* of the starting point of the line.
- **end** (*array_like*) – Coordinate *(x, y)* of the end-point of the line.
- **units** (*str*) – Units to use when drawing. This will affect how parameters and attributes *pos*, *size* and *radius* are interpreted.
- **lineWidth** (*float*) – Width of the line.
- **lineColor** (array_like, str, *Color* or None) – Color of the line. If *None*, a fully transparent color is used which makes the line invisible. *Deprecated* use *color* instead.

- **pos** (*array_like*) – Initial translation (*x*, *y*) of the line on-screen relative to the origin located at the center of the window or buffer in *units*. This can be updated after initialization by setting the *pos* property. The default value is *(0.0, 0.0)* which results in no translation.
- **size** (*float or array_like*) – Initial scale factor for adjusting the size of the line. A single value (*float*) will apply uniform scaling, while an array (*sx*, *sy*) will result in anisotropic scaling in the horizontal (*sx*) and vertical (*sy*) direction. Providing negative values to *size* will cause the line to be mirrored. Scaling can be changed by setting the *size* property after initialization. The default value is *1.0* which results in no scaling.
- **ori** (*float*) – Initial orientation of the line in degrees about its origin. Positive values will rotate the line clockwise, while negative values will rotate counterclockwise. The default value for *ori* is 0.0 degrees.
- **opacity** (*float*) – Opacity of the line. A value of 1.0 indicates fully opaque and 0.0 is fully transparent (therefore invisible). Values between 1.0 and 0.0 will result in colors being blended with objects in the background. This value affects the fill (*fillColor*) and outline (*lineColor*) colors of the shape.
- **contrast** (*float*) – Contrast level of the line (0.0 to 1.0). This value is used to modulate the contrast of colors passed to *lineColor* and *fillColor*.
- **depth** (*int*) – Depth layer to draw the stimulus when *autoDraw* is enabled.
- **interpolate** (*bool*) – Enable smoothing (anti-aliasing) when drawing lines. This produces a smoother (less-pixelated) line.
- **draggable** (*bool*) – Can this stimulus be dragged by a mouse click?
- **name** (*str*) – Optional name of the stimuli for logging.
- **autoLog** (*bool*) – Enable auto-logging of events associated with this stimuli. Useful for debugging and to track timing when used in conjunction with *autoDraw*.
- **autoDraw** (*bool*) – Enable auto drawing. When *True*, the stimulus will be drawn every frame without the need to explicitly call the `draw()` method.
- **color** (array_like, str, `Color` or None) – Sets both the initial *lineColor* and *fillColor* of the shape.
- **colorSpace** (*str*) – Sets the colorspace, changing how values passed to *lineColor* and *fillColor* are interpreted.

Notes

The *contains* method always return *False* because a line is not a proper (2D) polygon.

start, end

Coordinates *(x, y)* for the start- and end-point of the line.

Type

array_like

property RGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type

DEPRECATED

static _calcEquilateralVertices(*edges*, *radius=0.5*)

Get vertices for an equilateral shape with a given number of sides, will assume radius is 0.5 (relative) but can be manually specified `_calcPosRendered()`

DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix.

`_calcSizeRendered()`

DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix

`static _calculateMinEdges(`*lineWidth*, *threshold=180*`)`

Calculate how many points are needed in an equilateral polygon for the gap between line rects to be < 1px and for corner angles to exceed a threshold.

In other words, how many edges does a polygon need to have to appear smooth?

lineWidth
[int, float, np.ndarray] Width of the line in pixels

threshold
[int] Maximum angle (degrees) for corners of the polygon, useful for drawing a circle. Supply 180 for no maximum angle.

`_drawLegacyGL(`*win*, *keepMatrix*`)`

Legacy draw the stimulus in the relevant window.

You must call this method after every *win.flip()* if you want the stimulus to appear on that frame and then update the screen again.

`_getDesiredRGB(`*rgb*, *colorSpace*, *contrast*`)`

Convert color to RGB while adding contrast. Requires self.rgb, self.colorSpace and self.contrast

`_getPolyAsRendered()`

DEPRECATED. Return a list of vertices as rendered.

`_legacyTesselate(`*newVertices*`)`

Legacy tessellation method for ShapeStim.

`_selectWindow(`*win*`)`

Switch drawing to the specified window. Calls the window's _setCurrent() method which handles the switch.

`_set(`*attrib*, *val*, *op=''*, *log=None*`)`

DEPRECATED since 1.80.04 + 1. Use setAttribute() and val2array() instead.

`_tesselate(`*newVertices*`)`

Set the *.vertices* and *.border* to new values, invoking tessellation.

Parameters
`newVertices` (*array_like*) – Nx2 array of points (eg., *[[-0.5, 0], [0, 0.5], [0.5, 0]*).

`_updateList()`

The user shouldn't need this method since it gets called after every call to .set() Chooses between using and not using shaders each call.

`_updateVertices()`

Sets Stim.verticesPix and ._borderPix from pos, size, ori, flipVert, flipHoriz

`alphaThreshold`

Threshold for alpha values.

If the alpha value of a pixel is below this threshold, the pixel will be rejected (not drawn). This can be useful for creating a mask from an image with an alpha channel. The default value is 0.0, which means that no thresholding will be applied.

`autoDraw`

Determines whether the stimulus should be automatically drawn on every frame flip.

Value should be: *True* or *False*. You do NOT need to set this on every frame flip!

`autoLog`

Whether every change in this stimulus should be auto logged.

Value should be: *True* or *False*. Set to *False* if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren't likely to be useful.

`property backColor`

Alternative way of setting fillColor

`property backColorSpace`

Deprecated, please use colorSpace to set color space for the entire object.

`property backRGB`

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type
DEPRECATED

`property backgroundColor`

Alternative way of setting fillColor

`property borderColorSpace`

Deprecated, please use colorSpace to set color space for the entire object

`property borderRGB`

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type
DEPRECATED

`closeShape`

Should the last vertex be automatically connected to the first?

If you're using *Polygon*, *Circle* or *Rect*, *closeShape=True* is assumed and shouldn't be changed.

`property color`

Set the color of the shape. Sets both *fillColor* and *lineColor* simultaneously if applicable.

`property colorSpace`

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                       color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
```

(continues on next page)

(continued from previous page)

```
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

contains(**args*, ***kwargs*)

Returns True if a point x,y is inside the stimulus' border.

Can accept variety of input options:

- two separate args, x and y
- one arg (list, tuple or array) containing two vals (x,y)
- **an object with a getPos() method that returns x,y, such**
  as a `Mouse`.

Returns *True* if the point is within the area defined either by its *border* attribute (if one defined), or its *vertices* attribute if there is no .border. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the *contains* method; the extent of the stimulus is determined purely by the size, position (pos), and orientation (ori) settings (and by the vertices for shape stimuli).

See Coder demos: shapeContains.py See Coder demos: shapeContains.py

property contrast

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged).
Operations supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast =  1.0  # unchanged contrast
stim.contrast =  0.5  # decrease contrast
stim.contrast =  0.0  # uniform, no contrast
stim.contrast = -0.5  # slightly inverted
stim.contrast = -1.0  # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast =  1.2  # increases contrast
stim.contrast = -1.2  # inverts with increased contrast
```

depth

DEPRECATED, depth is now controlled simply by drawing order.

doDragging()

If this stimulus is draggable, do the necessary actions on a frame flip to drag it.

draggable

Can this stimulus be dragged by a mouse click?

`draw`(*win=None*, *keepMatrix=False*)

Draw the stimulus in the relevant window.

You must call this method after every *win.flip()* if you want the stimulus to appear on that frame and then update the screen again.

Parameters

- **win** (*Window*, optional) – Window to draw the stimulus in. If not specified, the stimulus will be drawn in the window specified at initialization.
- **keepMatrix** (*bool, optional*) – *DEPRECATED* If *True*, the current transformation matrix will be preserved. This is useful when drawing multiple stimuli with the same transformation matrix. Default is *False*.

`end`

tuple, list or 2x1 array

Specifies the position of the end of the line. *Operations* supported.

`property fillColor`

Set the fill color for the shape.

`property fillColorSpace`

Deprecated, please use colorSpace to set color space for the entire object.

`property fillRGB`

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type
DEPRECATED

`property flip`

1x2 array for flipping vertices along each axis; set as True to flip or False to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (*._flip*) will give an array of 1s and -1s with which to multiply vertices.

`property fontColor`

Alternative way of setting *foreColor*.

`property foreColor`

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names <http://www.w3schools.com/html/html_colornames.asp>* can be used.
- *Colors by hex value*
- **numerically: (scalar or triplet) for DKL, RGB or**
  other *Color spaces*. For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue'  # (the case is actually ignored)
stim.color = '#DDA0DD'  # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0]  # if stim.colorSpace='rgb':
                # a red color in rgb space
stim.color = [0.0, 45.0, 1.0]  # if stim.colorSpace='dkl':
                # DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255]  # if stim.colorSpace='rgb255':
                # a blue stimulus using rgb255 space
stim.color = 255  # interpreted as (255, 255, 255)
                  # which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1]  # increment all guns by 1 value
stim.color *= -1  # multiply the color by -1 (which in this
                    # space inverts the contrast)
stim.color *= [0.5, 0, 1]  # decrease red, remove green, keep blue
```

You can use *setColor* if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

property foreColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property foreRGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type
DEPRECATED

interpolate

If *True* the edge of the line will be anti-aliased.

property lineColor

Alternative way of setting *borderColor*.

property lineColorSpace

Deprecated, please use colorSpace to set color space for the entire object

property lineRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type
DEPRECATED

lineWidth

Width of the line in **pixels**.

Operations supported.

name

The name (*str*) of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = None your stimulus will be called "unnamed <type>", e.g. visual.TextStim(win) will be called "unnamed TextStim" in the logs.

property opacity

Determines how visible the stimulus is relative to background.

The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). *Operations* are supported. Precisely how this is used depends on the *Blend Mode*.

ori

The orientation of the stimulus (in degrees).

Should be a single value (*scalar*). *Operations* are supported.

Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and below zero values wrap appropriately.

overlaps(*polygon*)

Returns *True* if this stimulus intersects another one.

If *polygon* is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the *overlaps* method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

See coder demo, shapeContains.py

property pos

The position of the center of the stimulus in the stimulus *units*

value should be an *x,y-pair*. *Operations* are also supported.

Example:

```
stim.pos = (0.5, 0)  # Set slightly to the right of center
stim.pos += (0.5, -1)  # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2  # Move stim towards the center.
    Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:

```
from psychopy.tools.monitorunittools import posToPix
posPix = posToPix(stim)
```

setAlphaThreshold(*value*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setAutoDraw(*value*, *log=None*)

Sets autoDraw. Usually you can use 'stim.attribute = value' syntax instead, but use this method to suppress the log message.

`setAutoLog`(*value=True*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

`setBackRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

`setBorderColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

`setBorderRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

`setColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Sets both the line and fill to be the same color.

`setContrast`(*newContrast*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

`setDKL`(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

`setDepth`(*newDepth*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

`setEnd`(*end*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

`setFillColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for fillColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

`setFillRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

`setForeColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for foreColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

`setForeRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

`setLMS`(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

`setLineRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

`setOpacity`(*newOpacity*, *operation=''*, *log=None*)

Hard setter for opacity, allows the suppression of log messages and calls the update method

`setOri`(*newOri*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message `setPos`(*newPos*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

`setRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

`setSize`(*newSize*, *operation=''*, *units=None*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

`setStart`(*start*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

`setVertices`(*value=None*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

property size

The size (width, height) of the stimulus in the stimulus *units*

Value should be *x,y-pair*, *scalar* (applies to both dimensions) or None (resets to default). *Operations* are supported.

Sizes can be negative (causing a mirror-image reversal) and can extend beyond the window.

Example:

```
stim.size = 0.8  # Set size to (xsize, ysize) = (0.8, 0.8)
print(stim.size)  # Outputs array([0.8, 0.8])
stim.size += (0.5, -0.5)  # make wider and flatter: (1.3, 0.3)
```

Tip: if you can see the actual pixel range this corresponds to by looking at *stim._sizeRendered*

start

tuple, list or 2x1 array.

Specifies the position of the start of the line. *Operations* supported.

`updateColors`()

Placeholder method to update colours when set externally, for example updating the *pallette* attribute of a textbox

`updateOpacity`()

Placeholder method to update colours when set externally, for example updating the *pallette* attribute of a textbox.

property vertices

A list of lists or a numpy array (Nx2) specifying xy positions of each vertex, relative to the center of the field.

Assigning to vertices can be slow if there are many vertices.

Operations supported with *.setVertices()*.

property verticesPix

This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

property win

The *Window* object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1  # stimulus will be drawn in win1
stim.draw()  # stimulus is now drawn to win1
stim.win = win2  # stimulus will be drawn in win2
stim.draw()  # it is now drawn in win2
win1.flip(waitBlanking=False)  # do not wait for next
             # monitor update
win2.flip()  # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

11.4.15 MovieStim

Class for presenting movie clips as stimuli. This is a lazy-imported class, therefore import using full path *from psychopy.visual.movies import MovieStim* when inheriting from it.

Attributes

MovieStim(win[, filename, movieLib, units, ...])	Class for presenting movie clips as stimuli.
MovieStim.win	The *Window* object in which the stimulus will be rendered by default.
MovieStim.units	
MovieStim.pos	The position of the center of the stimulus in the stimulus *units*
MovieStim.ori	The orientation of the stimulus (in degrees).
MovieStim.size	The size (width, height) of the stimulus in the stimulus *units*
MovieStim.opacity	Determines how visible the stimulus is relative to background.
MovieStim.name	The name (*str*) of the object to be using during logged messages about this stim.
MovieStim.autoLog	Whether every change in this stimulus should be auto logged.
MovieStim.draw([win])	Draw the current frame to a particular window.
MovieStim.autoDraw	Determines whether the stimulus should be automatically drawn on every frame flip.
MovieStim.loadMovie(filename)	Load a movie file from disk.
MovieStim.play([log])	Start or continue a paused movie from current position.
MovieStim.seek(timestamp[, log])	Seek to a particular timestamp in the movie.
MovieStim.pause([log])	Pause the current point in the movie.
MovieStim.stop([log])	Stop the current point in the movie (sound will stop, current frame will not advance and remain on-screen).
MovieStim.setFlipHoriz(value[, log, ...])	
MovieStim.setFlipVert(value[, log, ...])	

Details

class psychopy.visual.**MovieStim**(*win*, *filename=''*, *movieLib='ffpyplayer'*, *units='pix'*, *size=None*, *pos=(0.0, 0.0)*, *ori=0.0*, *anchor='center'*, *draggable=False*, *flipVert=False*, *flipHoriz=False*, *color=(1.0, 1.0, 1.0)*, *colorSpace='rgb'*, *opacity=1.0*, *contrast=1*, *volume=1.0*, *name=''*, *loop=False*, *autoLog=True*, *depth=0.0*, *noAudio=False*, *interpolate=True*, *autoStart=True*)

Class for presenting movie clips as stimuli.

Parameters

- **win** (*Window*) – Window the video is being drawn to.
- **filename** (*str*) – Name of the file or stream URL to play. If an empty string, no file will be loaded on initialization but can be set later.
- **movieLib** (*str or None*) – Library to use for video decoding. By default, the 'preferred' library by PsychoPy developers is used. Default is *'ffpyplayer'*. An alert is raised if you are not using the preferred player.
- **units** (*str*) – Units to use when sizing the video frame on the window, affects how *size* is interpreted.

- **size** (*ArrayLike or None*) – Size of the video frame on the window in *units*. If *None*, the native size of the video will be used.
- **draggable** (*bool*) – Can this stimulus be dragged by a mouse click?
- **flipVert** (*bool*) – If *True* then the movie will be top-bottom flipped.
- **flipHoriz** (*bool*) – If *True* then the movie will be right-left flipped.
- **volume** (*int or float*) – If specifying an *int* the nominal level is 100, and 0 is silence. If a *float*, values between 0 and 1 may be used.
- **loop** (*bool*) – Whether to start the movie over from the beginning if draw is called and the movie is done. Default is *False*.
- **autoStart** (*bool*) – Automatically begin playback of the video when *flip()* is called.

`property RGB`

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type
DEPRECATED

`_calcPosRendered()`

DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix.

`_calcSizeRendered()`

DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix

`_drawRectangle()`

Draw the video frame to the window.

This is called by the *draw()* method to blit the video to the display window.

`_freeBuffers()`

Free texture and pixel buffers. Call this when tearing down this class or if a movie is stopped.

`_getDesiredRGB`(*rgb*, *colorSpace*, *contrast*)

Convert color to RGB while adding contrast. Requires self.rgb, self.colorSpace and self.contrast

`_getPolyAsRendered()`

DEPRECATED. Return a list of vertices as rendered.

`_pixelTransfer()`

Copy pixel data from video frame to texture.

`_selectWindow`(*win*)

Switch drawing to the specified window. Calls the window's _setCurrent() method which handles the switch.

`_set`(*attrib*, *val*, *op=''*, *log=None*)

DEPRECATED since 1.80.04 + 1. Use setAttribute() and val2array() instead.

`_setupTextureBuffers()`

Setup texture buffers which hold frame data. This creates a 2D RGB texture and pixel buffer. The pixel buffer serves as the store for texture color data. Each frame, the pixel buffer memory is mapped and frame data is copied over to the GPU from the decoder.

This is called every time a video file is loaded. The *_freeBuffers* method is called in this routine prior to creating new buffers, so it's safe to call this right after loading a new movie without having to *_freeBuffers* first.

`_updateList()`

The user shouldn't need this method since it gets called after every call to .set() Chooses between using and not using shaders each call.

`_updateVertices()`

Sets Stim.verticesPix and ._borderPix from pos, size, ori, flipVert, flipHoriz

`alphaThreshold`

Threshold for alpha values.

If the alpha value of a pixel is below this threshold, the pixel will be rejected (not drawn). This can be useful for creating a mask from an image with an alpha channel. The default value is 0.0, which means that no thresholding will be applied.

`property anchor`

`autoDraw`

Determines whether the stimulus should be automatically drawn on every frame flip.

Value should be: *True* or *False*. You do NOT need to set this on every frame flip!

`autoLog`

Whether every change in this stimulus should be auto logged.

Value should be: *True* or *False*. Set to *False* if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren't likely to be useful.

`property autoStart`

Start playback when *.draw()* is called (*bool*).

`property backColor`

Alternative way of setting fillColor

`property backColorSpace`

Deprecated, please use colorSpace to set color space for the entire object.

`property backRGB`

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type
DEPRECATED

`property backgroundColor`

Alternative way of setting fillColor

`property borderColor`

`property borderColorSpace`

Deprecated, please use colorSpace to set color space for the entire object

`property borderRGB`

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type
DEPRECATED

`borderWidth`

property color

Alternative way of setting *foreColor*.

property colorSpace

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                       color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

contains(*x*, *y=None*, *units=None*)

Returns True if a point x,y is inside the stimulus' border.

Can accept variety of input options:

- two separate args, x and y
- one arg (list, tuple or array) containing two vals (x,y)
- **an object with a getPos() method that returns x,y, such**
  as a *Mouse*.

Returns *True* if the point is within the area defined either by its *border* attribute (if one defined), or its *vertices* attribute if there is no .border. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the *contains* method; the extent of the stimulus is determined purely by the size, position (pos), and orientation (ori) settings (and by the vertices for shape stimuli).

See Coder demos: shapeContains.py See Coder demos: shapeContains.py

property contrast

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged).
Operations supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast =  1.0  # unchanged contrast
stim.contrast =  0.5  # decrease contrast
stim.contrast =  0.0  # uniform, no contrast
```

(continues on next page)

(continued from previous page)

```
stim.contrast = -0.5  # slightly inverted
stim.contrast = -1.0  # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast =  1.2  # increases contrast
stim.contrast = -1.2  # inverts with increased contrast
```

depth

DEPRECATED, depth is now controlled simply by drawing order.

doDragging()

If this stimulus is draggable, do the necessary actions on a frame flip to drag it.

draggable

Can this stimulus be dragged by a mouse click?

draw(*win=None*)

Draw the current frame to a particular window.

The current position in the movie will be determined automatically. This method should be called on every frame that the movie is meant to appear. If *.autoStart==True* the video will begin playing when this is called.

Parameters

win (*Window* or *None*) – Window the video is being drawn to. If *None*, the window specified at initialization will be used instead.

Returns

True if the frame was updated this draw call.

Return type

bool

property duration

Duration of the loaded video in seconds (*float*). Not valid unless the video has been started.

fastForward(*seconds=5*, *log=True*)

Fast-forward the video.

Parameters

- **seconds** (*float*) – Time in seconds to fast forward from the current position. Default is 5 seconds.
- **log** (*bool*) – Log this event.

property filename

File name for the loaded video (*str*).

property fillColor

Set the fill color for the shape.

property fillColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property fillRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type
DEPRECATED

property flip

1x2 array for flipping vertices along each axis; set as True to flip or False to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (*._flip*) will give an array of 1s and -1s with which to multiply vertices.

property flipHoriz

property flipVert

property fontColor

Alternative way of setting *foreColor*.

property foreColor

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names <http://www.w3schools.com/html/html_colornames.asp>* can be used.
- *Colors by hex value*
- **numerically: (scalar or triplet) for DKL, RGB or**
  other *Color spaces*. For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue'  # (the case is actually ignored)
stim.color = '#DDA0DD'  # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0]  # if stim.colorSpace='rgb':
                # a red color in rgb space
stim.color = [0.0, 45.0, 1.0]  # if stim.colorSpace='dkl':
                # DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255]  # if stim.colorSpace='rgb255':
                # a blue stimulus using rgb255 space
stim.color = 255  # interpreted as (255, 255, 255)
                  # which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1]  # increment all guns by 1 value
stim.color *= -1  # multiply the color by -1 (which in this
                    # space inverts the contrast)
stim.color *= [0.5, 0, 1]  # decrease red, remove green, keep blue
```

You can use *setColor* if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

property foreColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property foreRGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type
DEPRECATED

property fps

Movie frames per second (*float*).

property frameIndex

Current frame index being displayed (*int*).

property frameRate

Frame rate of the movie in Hertz (*float*).

property frameSize

Size of the video *(w, h)* in pixels (*tuple*). Alias of *videoSize*.

property frameTexture

Texture ID for the current video frame (*GLuint*). You can use this as a video texture. However, you must periodically call *updateVideoFrame* to keep this up to date.

getAlphaThreshold()

getAnchor()

getAutoDraw()

getAutoLog()

getAutoStart()

getBackColor()

getBackColorSpace()

getBackRGB()

getBackgroundColor()

getBorderColor()

getBorderColorSpace()

getBorderRGB()

getBorderWidth()

getColor()

getColorSpace()

getContrast()

getCurrentFrameNumber()

Get the current movie frame number (*int*), same as *frameIndex*.

getDepth()

getDraggable()

getDuration()

getFPS()

Movie frames per second.

Returns

Nominal number of frames to be displayed per second.

Return type

float

getFilename()

getFillColor()

getFillColorSpace()

getFillRGB()

getFlip()

getFlipHoriz()

getFlipVert()

getFontColor()

getForeColor()

getForeColorSpace()

getForeRGB()

getFps()

getFrameIndex()

getFrameRate()

getFrameSize()

getFrameTexture()

getHeight()

getIsFinished()

getIsNotStarted()

getIsPaused()

`getIsPlaying()`

`getIsStopped()`

`getLineColor()`

`getLineColorSpace()`

`getLineRGB()`

`getLineWidth()`

`getLoopCount()`

`getMuted()`

`getName()`

`getOpacity()`

`getOri()`

`getOrigSize()`

`getPercentageComplete()`

Provides a value between 0.0 and 100.0, indicating the amount of the movie that has been already played (*float*).

`getPos()`

`getPts()`

`getRGB()`

`getSize()`

`getUnits()`

`getVertices()`

`getVerticesPix()`

`getVideoSize()`

`getVolume()`

`getWidth()`

`getWin()`

`get_borderPix()`

`get_hasPlayer()`

`property height`

`isDragging = False`

`property isFinished`

True if the video is finished (*bool*).

property isNotStarted

True if the video may not have started yet (*bool*). This status is given after a video is loaded and play has yet to be called.

property isPaused

True if the video is presently paused (*bool*).

property isPlaying

True if the video is presently playing (*bool*).

property isStopped

True if the video is stopped (*bool*). It will resume from the beginning if *play()* is called.

property lineColor

Alternative way of setting *borderColor*.

property lineColorSpace

Deprecated, please use colorSpace to set color space for the entire object

property lineRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type
DEPRECATED

lineWidth

load(*filename*)

Load a movie file from disk (alias of *loadMovie*).

Parameters
filename (*str*) – Path to movie file. Must be a format that FFMPEG supports.

loadMovie(*filename*)

Load a movie file from disk.

Parameters
filename (*str*) – Path to movie file. Must be a format that FFMPEG supports.

property loopCount

Number of loops completed since playback started (*int*). Incremented each time the movie begins another loop.

Examples

Compute how long a looping video has been playing until now:

```
totalMovieTime = (mov.loopCount + 1) * mov.pts
```

property muted

True if the stream audio is muted (*bool*).

name

The name (*str*) of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = None your stimulus will be called “unnamed <type>”, e.g. visual.TextStim(win) will be called “unnamed TextStim” in the logs.

property opacity

Determines how visible the stimulus is relative to background.

The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). *Operations* are supported. Precisely how this is used depends on the *Blend Mode*.

ori

The orientation of the stimulus (in degrees).

Should be a single value (*scalar*). *Operations* are supported.

Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and below zero values wrap appropriately.

property origSize

Alias of videoSize

overlaps(*polygon*)

Returns *True* if this stimulus intersects another one.

If *polygon* is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the *overlaps* method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

See coder demo, shapeContains.py

pause(*log=True*)

Pause the current point in the movie. The image of the last frame will persist on-screen until *play()* or *stop()* are called.

Parameters

log (*bool*) – Log this event.

play(*log=True*)

Start or continue a paused movie from current position.

Parameters

log (*bool*) – Log the play event.

property pos

The position of the center of the stimulus in the stimulus *units*

value should be an *x,y-pair*. *Operations* are also supported.

Example:

```
stim.pos = (0.5, 0)  # Set slightly to the right of center
stim.pos += (0.5, -1)  # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2  # Move stim towards the center.
    Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:

```
from psychopy.tools.monitorunittools import posToPix
posPix = posToPix(stim)
```

property pts

Presentation timestamp of the most recent frame (*float*).

This value corresponds to the time in movie/stream time the frame is scheduled to be presented.

replay(*log=True*)

Replay the movie from the beginning.

Parameters

log (*bool*) – Log this event.

Notes

- This tears down the current media player instance and creates a new one. Similar to calling *stop()* and *loadMovie()*. Use *seek(0.0)* if you would like to restart the movie without reloading.

rewind(*seconds=5*, *log=True*)

Rewind the video.

Parameters

- **seconds** (*float*) – Time in seconds to rewind from the current position. Default is 5 seconds.
- **log** (*bool*) – Log this event.

seek(*timestamp*, *log=True*)

Seek to a particular timestamp in the movie.

Parameters

- **timestamp** (*float*) – Time in seconds.
- **log** (*bool*) – Log this event.

setAlphaThreshold(*value*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setAnchor(*value*, *log=None*)

setAutoDraw(*value*, *log=None*)

Sets autoDraw. Usually you can use 'stim.attribute = value' syntax instead, but use this method to suppress the log message.

setAutoLog(*value=True*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setAutoStart(*value*, *log=None*, *operation=False*, *stealth=False*)

setBackColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setBackColorSpace(*value*, *log=None*, *operation=False*, *stealth=False*)

setBackRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setBackgroundColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setBorderColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setBorderColorSpace(*value*, *log=None*, *operation=False*, *stealth=False*)

setBorderRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setBorderWidth(*newWidth*, *operation=''*, *log=None*)

setColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setColorSpace(*value*, *log=None*, *operation=False*, *stealth=False*)

setContrast(*newContrast*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setDKL(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setDepth(*newDepth*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setDraggable(*value*, *log=None*, *operation=False*, *stealth=False*)

setFilename(*value*, *log=None*, *operation=False*, *stealth=False*)

setFillColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for fillColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setFillColorSpace(*value*, *log=None*, *operation=False*, *stealth=False*)

setFillRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setFlip(*value*, *log=None*, *operation=False*, *stealth=False*)

setFlipHoriz(*value*, *log=None*, *operation=False*, *stealth=False*)

setFlipVert(*value*, *log=None*, *operation=False*, *stealth=False*)

setFontColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setForeColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for foreColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setForeColorSpace(*value*, *log=None*, *operation=False*, *stealth=False*)

setForeRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setHeight(*value*, *log=None*, *operation=False*, *stealth=False*)

setLMS(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setLineColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setLineColorSpace(*value*, *log=None*, *operation=False*, *stealth=False*)

setLineRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setLineWidth(*newWidth*, *operation=''*, *log=None*)

setMovie(*value*)

setMuted(*value*, *log=None*, *operation=False*, *stealth=False*)

setName(*value*, *log=None*, *operation=False*, *stealth=False*)

setOpacity(*newOpacity*, *operation=''*, *log=None*)

Hard setter for opacity, allows the suppression of log messages and calls the update method

setOri(*newOri*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setPos(*newPos*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setSize(*newSize*, *operation=''*, *units=None*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setUnits(*value*, *log=None*, *operation=False*, *stealth=False*)

setVertices(*value*, *log=None*, *operation=False*, *stealth=False*)

setVolume(*value*, *log=None*, *operation=False*, *stealth=False*)

setWidth(*value*, *log=None*, *operation=False*, *stealth=False*)

setWin(*value*, *log=None*, *operation=False*, *stealth=False*)

property size

The size (width, height) of the stimulus in the stimulus *units*

Value should be *x,y-pair*, *scalar* (applies to both dimensions) or None (resets to default). *Operations* are supported.

Sizes can be negative (causing a mirror-image reversal) and can extend beyond the window.

Example:

```
stim.size = 0.8  # Set size to (xsize, ysize) = (0.8, 0.8)
print(stim.size)  # Outputs array([0.8, 0.8])
stim.size += (0.5, -0.5)  # make wider and flatter: (1.3, 0.3)
```

Tip: if you can see the actual pixel range this corresponds to by looking at *stim._sizeRendered*

stop(*log=True*)

Stop the current point in the movie (sound will stop, current frame will not advance and remain on-screen). Once stopped the movie can be restarted from the beginning by calling *play()*.

Parameters
log (*bool*) – Log this event.

toggle(*log=True*)

Switch between playing and pausing the movie. If the movie is playing, this function will pause it. If the movie is paused, this function will play it.

Parameters
log (*bool*) – Log this event.

property units

unload(*log=True*)

Stop and unload the movie.

Parameters
log (*bool*) – Log this event.

updateColors()

Placeholder method to update colours when set externally, for example updating the *pallette* attribute of a textbox

updateOpacity()

Placeholder method to update colours when set externally, for example updating the *pallette* attribute of a textbox.

updateVideoFrame()

Update the present video frame. The next call to *draw()* will make the retrieved frame appear.

Returns
If *True*, the video texture has been updated and the frame index is advanced by one. If *False*, the last frame should be kept on-screen.

Return type
bool

property vertices

property verticesPix

This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

property videoSize

Size of the video *(w, h)* in pixels (*tuple*). Returns *(0, 0)* if no video is loaded.

property volume

Volume for the audio track for this movie (*int* or *float*).

volumeDown(*amount=0.05*)

Decrease the volume by a fixed amount.

Parameters
amount (*float or int*) – Amount to decrease the volume relative to the current volume.

volumeUp(*amount=0.05*)

Increase the volume by a fixed amount.

Parameters

amount (*float or int*) – Amount to increase the volume relative to the current volume.

property width

property win

The *Window* object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1  # stimulus will be drawn in win1
stim.draw()  # stimulus is now drawn to win1
stim.win = win2  # stimulus will be drawn in win2
stim.draw()  # it is now drawn in win2
win1.flip(waitBlanking=False)  # do not wait for next
             # monitor update
win2.flip()  # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

11.4.16 NoiseStim

Attributes

Details

class psychopy.visual.**NoiseStim**(**args*, ***kwargs*)

psychopy.visual.noise is now located within the psychopy-visionscience plugin.

When initialised, rather than creating an object, will log an error.

11.4.17 ObjMeshStim

Attributes

ObjMeshStim(win, objFile[, pos, ori, ...])	Class for loading and presenting 3D stimuli in the Wavefront OBJ format.

Details

class psychopy.visual.**ObjMeshStim**(*win*, *objFile*, *pos=(0, 0, 0)*, *ori=(0, 0, 0, 1)*, *useMaterial=None*, *loadMtllib=True*, *color=(0.0, 0.0, 0.0)*, *colorSpace='rgb'*, *contrast=1.0*, *opacity=1.0*, *name=''*, *autoLog=True*)

Class for loading and presenting 3D stimuli in the Wavefront OBJ format. This is a lazy-imported class, therefore import using full path *from psychopy.visual.stim3d import ObjMeshStim* when inheriting from it.

Calling the *draw* method will render the mesh to the current buffer. The render target (FBO or back buffer) must have a depth buffer attached to it for the object to be rendered correctly. Shading is used if the current window has light sources defined and lighting is enabled (by setting *useLights=True* before drawing the stimulus).

Vertex positions, texture coordinates, and normals are loaded and packed into a single vertex buffer object (VBO). Vertex array objects (VAO) are created for each material with an index buffer referencing vertices assigned that material in the VBO. For maximum performance, keep the number of materials per object as low as possible, as switching between VAOs has some overhead.

Material attributes are read from the material library file (*.MTL) associated with the *.OBJ file. This file will be automatically searched for and read during loading. Afterwards you can edit material properties by accessing the data structure of the *materials* attribute.

Keep in mind that OBJ shapes are rigid bodies, the mesh itself cannot be deformed during runtime. However, meshes can be positioned and rotated as desired by manipulating the *RigidBodyPose* instance accessed through the *thePose* attribute.

Warning

Loading an *.OBJ file is a slow process, be sure to do this outside of any time-critical routines! This class is experimental and may result in undefined behavior.

Examples

Loading an *.OBJ file from a disk location:

```
myObjStim = ObjMeshStim(win, '/path/to/file/model.obj')
```

Parameters

- **win** (*~psychopy.visual.Window*) – Window this stimulus is associated with. Stimuli cannot be shared across windows unless they share the same context.
- **size** (*tuple or float*) – Dimensions of the mesh. If a single value is specified, the plane will be a square. Provide a tuple of floats to specify the width and length of the box (eg. *size=(0.2, 1.3)*).
- **pos** (*array_like*) – Position vector *[x, y, z]* for the origin of the rigid body.
- **ori** (*array_like*) – Orientation quaternion *[x, y, z, w]* where *x*, *y*, *z* are imaginary and *w* is real. If you prefer specifying rotations in axis-angle format, call *setOriAxisAngle* after initialization. By default, the plane is oriented with normal facing the +Z axis of the scene.
- **useMaterial** (*PhongMaterial, optional*) – Material to use for all sub-meshes. The material can be configured by accessing the *material* attribute after initialization. If no material is specified, *color* will modulate the diffuse and ambient colors for all meshes in the model. If *loadMtllib* is *True*, this value should be *None*.
- **loadMtllib** (*bool*) – Load materials from the MTL file associated with the mesh. This will override *useMaterial* if it is *None*. The value of *materials* after initialization will be a dictionary where keys are material names and values are materials. Any textures associated with the model will be loaded as per the material requirements.

property RGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type

DEPRECATED

`_createVAO`(*vertices*, *textureCoords*, *normals*, *faces*)

Create a vertex array object for handling vertex attribute data.

`_getDesiredRGB`(*rgb*, *colorSpace*, *contrast*)

Convert color to RGB while adding contrast. Requires self.rgb, self.colorSpace and self.contrast

`_loadMtlLib`(*mtlFile*)

Load a material library associated with the OBJ file. This is usually called by the constructor for this class.

Parameters

mtlFile (*str*) – Path to MTL file.

`_selectWindow`(*win*)

Switch drawing to the specified window. Calls the window's _setCurrent() method which handles the switch.

`_updateList`()

The user shouldn't need this method since it gets called after every call to .set() Chooses between using and not using shaders each call.

`property anchor`

`property backColor`

Alternative way of setting fillColor

`property backColorSpace`

Deprecated, please use colorSpace to set color space for the entire object.

`property backRGB`

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type

DEPRECATED

`property backgroundColor`

Alternative way of setting fillColor

`property borderColor`

`property borderColorSpace`

Deprecated, please use colorSpace to set color space for the entire object

`property borderRGB`

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type

DEPRECATED

`borderWidth`

`property color`

Alternative way of setting *foreColor*.

`property colorSpace`

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                       color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

property contrast

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged).
Operations supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast =  1.0  # unchanged contrast
stim.contrast =  0.5  # decrease contrast
stim.contrast =  0.0  # uniform, no contrast
stim.contrast = -0.5  # slightly inverted
stim.contrast = -1.0  # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast =  1.2  # increases contrast
stim.contrast = -1.2  # inverts with increased contrast
```

draw(*win=None*)

Draw the mesh.

Parameters
win (*~psychopy.visual.Window*) – Window this stimulus is associated with. Stimuli cannot be shared across windows unless they share the same context.

property fillColor

Set the fill color for the shape.

property fillColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property fillRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type
DEPRECATED

property flip

1x2 array for flipping vertices along each axis; set as True to flip or False to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (*._flip*) will give an array of 1s and -1s with which to multiply vertices.

property flipHoriz

property flipVert

property fontColor

Alternative way of setting *foreColor*.

property foreColor

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names <http://www.w3schools.com/html/html_colornames.asp>* can be used.
- *Colors by hex value*
- **numerically: (scalar or triplet) for DKL, RGB or**
  other *Color spaces*. For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue'  # (the case is actually ignored)
stim.color = '#DDA0DD'  # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0]  # if stim.colorSpace='rgb':
                # a red color in rgb space
stim.color = [0.0, 45.0, 1.0]  # if stim.colorSpace='dkl':
                # DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255]  # if stim.colorSpace='rgb255':
                # a blue stimulus using rgb255 space
stim.color = 255  # interpreted as (255, 255, 255)
                  # which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1]  # increment all guns by 1 value
stim.color *= -1  # multiply the color by -1 (which in this
                    # space inverts the contrast)
stim.color *= [0.5, 0, 1]  # decrease red, remove green, keep blue
```

You can use *setColor* if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

property foreColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property foreRGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type
DEPRECATED

getOri()

getOriAxisAngle(*degrees=True*)

Get the axis and angle of rotation for the 3D stimulus. Converts the orientation defined by the *ori* quaternion to and axis-angle representation.

Parameters
`degrees` (*bool, optional*) – Specify `True` if *angle* is in degrees, or else it will be treated as radians. Default is `True`.

Returns
Axis *[rx, ry, rz]* and angle.

Return type
tuple

getPos()

getRayIntersectBounds(*rayOrig*, *rayDir*)

Get the point which a ray intersects the bounding box of this mesh.

Parameters

- `rayOrig` (*array_like*) – Origin of the ray in space [x, y, z].
- `rayDir` (*array_like*) – Direction vector of the ray [x, y, z], should be normalized.

Returns
Coordinate in world space of the intersection and distance in scene units from *rayOrig*. Returns *None* if there is no intersection.

Return type
tuple

property height

isVisible()

Check if the object is visible to the observer.

Test if a pose's bounding box or position falls outside of an eye's view frustum.

Poses can be assigned bounding boxes which enclose any 3D models associated with them. A model is not visible if all the corners of the bounding box fall outside the viewing frustum. Therefore any primitives (i.e. triangles) associated with the pose can be culled during rendering to reduce CPU/GPU workload.

Returns
True if the object's bounding box is visible.

Return type
bool

Examples

You can avoid running draw commands if the object is not visible by doing a visibility test first:

```
if myStim.isVisible():
    myStim.draw()
```

property lineColor

Alternative way of setting *borderColor*.

property lineColorSpace

Deprecated, please use colorSpace to set color space for the entire object

property lineRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type

DEPRECATED

lineWidth

property ori

Orientation quaternion (X, Y, Z, W).

property pos

Position vector (X, Y, Z).

setAnchor(*value*, *log=None*)

setBackColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setBackRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setBackgroundColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setBorderColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setBorderRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setBorderWidth(*newWidth*, *operation=''*, *log=None*)

setColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setContrast(*newContrast*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setDKL(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setFillColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for fillColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setFillRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setFontColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setForeColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for foreColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setForeRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setLMS(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setLineColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setLineRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setLineWidth(*newWidth*, *operation=''*, *log=None*)

setOri(*ori*)

setOriAxisAngle(*axis*, *angle*, *degrees=True*)

Set the orientation of the 3D stimulus using an *axis* and *angle*. This sets the quaternion at *ori*.

Parameters

- **axis** (*array_like*) – Axis of rotation [rx, ry, rz].
- **angle** (*float*) – Angle of rotation.
- **degrees** (*bool, optional*) – Specify `True` if *angle* is in degrees, or else it will be treated as radians. Default is `True`.

setPos(*pos*)

setRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

property size

property thePose

The pose of the rigid body. This is a class which has *pos* and *ori* attributes.

units

None, 'norm', 'cm', 'deg', 'degFlat', 'degFlatPos', or 'pix'

If None then the current units of the `Window` will be used. See *Units for the window and stimuli* for explanation of other options.

Note that when you change units, you don't change the stimulus parameters and it is likely to change appearance. Example:

```
# This stimulus is 20% wide and 50% tall with respect to window
stim = visual.PatchStim(win, units='norm', size=(0.2, 0.5))

# This stimulus is 0.2 degrees wide and 0.5 degrees tall.
stim.units = 'deg'
```

updateColors()

Placeholder method to update colours when set externally, for example updating the *pallette* attribute of a textbox

property vertices

property width

property win

The *Window* object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1  # stimulus will be drawn in win1
stim.draw()  # stimulus is now drawn to win1
stim.win = win2  # stimulus will be drawn in win2
stim.draw()  # it is now drawn in win2
win1.flip(waitBlanking=False)  # do not wait for next
             # monitor update
win2.flip()  # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

11.4.18 PatchStim (deprecated)

class psychopy.visual.PatchStim(**args*, ***kwargs*)

psychopy.visual.patch is now located within the psychopy-legacy plugin.

When initialised, rather than creating an object, will log an error.

11.4.19 BlinnPhongMaterial

Attributes

BlinnPhongMaterial([win, diffuseColor, ...])	Class representing a material using the Blinn-Phong lighting model.

Details

class psychopy.visual.BlinnPhongMaterial(*win=None*, *diffuseColor=(-1.0, -1.0, -1.0)*, *specularColor=(-1.0, -1.0, -1.0)*, *ambientColor=(-1.0, -1.0, -1.0)*, *emissionColor=(-1.0, -1.0, -1.0)*, *shininess=10.0*, *colorSpace='rgb'*, *diffuseTexture=None*, *opacity=1.0*, *contrast=1.0*, *face='front'*)

Class representing a material using the Blinn-Phong lighting model. This is a lazy-imported class, therefore import using full path *from psychopy.visual.stim3d import BlinnPhongMaterial* when inheriting from it.

This class stores material information to modify the appearance of drawn primitives with respect to lighting, such as color (diffuse, specular, ambient, and emission), shininess, and textures. Simple materials are intended to work with features supported by the fixed-function OpenGL pipeline. However, one may use shaders that implement the Blinn-Phong shading model for per-pixel lighting.

If shaders are enabled, the colors of objects will appear different than without. This is due to the lighting/material colors being computed on a per-pixel basis, and the formulation of the lighting model. The Phong shader determines the ambient color/intensity by adding up both the scene and light ambient colors, then multiplies them by the diffuse color of the material, as the ambient light's color should be a product of the surface reflectance (albedo) and the light color (the ambient light needs to reflect off something to be visible). Diffuse reflectance is Lambertian, where the cosine angle between the incident light ray and surface normal determines color. The size of specular highlights are related to the *shininess* factor which ranges from 1.0 to 128.0. The greater this number, the tighter the specular highlight making the surface appear smoother. If shaders are not being used, specular highlights will be computed using the Phong lighting model. The emission color is optional, it simply adds to the color of every pixel much like ambient lighting does. Usually, you would not really want this, but it can be used to add bias to the overall color of the shape.

If there are no lights in the scene, the diffuse color is simply multiplied by the scene and material ambient color to give the final color.

Lights are attenuated (fall-off with distance) using the formula:

```
attenuationFactor = 1.0 / (k0 + k1 * distance + k2 * pow(distance, 2))
```

The coefficients for attenuation can be specified by setting *attenuation* in the lighting object. Values *k0=1.0, k1=0.0, and k2=0.0* results in a light that does not fall-off with distance.

Parameters

- **win** (*~psychopy.visual.Window* or *None*) – Window this material is associated with, required for shaders and some color space conversions.
- **diffuseColor** (*array_like*) – Diffuse material color (r, g, b) with values between -1.0 and 1.0.
- **specularColor** (*array_like*) – Specular material color (r, g, b) with values between -1.0 and 1.0.
- **ambientColor** (*array_like*) – Ambient material color (r, g, b) with values between -1.0 and 1.0.
- **emissionColor** (*array_like*) – Emission material color (r, g, b) with values between -1.0 and 1.0.
- **shininess** (*float*) – Material shininess, usually ranges from 0.0 to 128.0.
- **colorSpace** (*str*) – Color space for *diffuseColor*, *specularColor*, *ambientColor*, and *emissionColor*. This is no longer used.
- **opacity** (*float*) – Opacity of the material. Ranges from 0.0 to 1.0 where 1.0 is fully opaque.
- **contrast** (*float*) – Contrast of the material colors.
- **diffuseTexture** (*TexImage2D*) – Optional 2D texture to apply to the material. Color values from the texture are blended with the *diffuseColor* of the material. The target primitives must have texture coordinates to specify how texels are mapped to the surface.
- **face** (*str*) – Face to apply material to. Values are *front*, *back* or *both*.

> **Warning**
>
> This class is experimental and may result in undefined behavior.

property ambientColor

Ambient color *(r, g, b)* of the material (*psychopy.color.Color*, *ArrayLike* or *None*).

property ambientRGB

RGB values of the ambient color of the material (*numpy.ndarray*).

begin(*useTextures=True*)

Use this material for successive rendering calls.

Parameters

useTextures (*bool*) – Enable textures.

property colorSpace

The name of the color space currently being used (*str* or *None*).

For strings and hex values this is not needed. If *None* the default *colorSpace* for the stimulus is used (defined during initialisation).

Please note that changing *colorSpace* does not change stimulus parameters. Thus, you usually want to specify *colorSpace* before setting the color.

property contrast

A value that is simply multiplied by the color (*float*).

This may be used to adjust the lightness of the material. This is applied to all material color components.

Examples

Basic usage:

```
stim.contrast =  1.0  # unchanged contrast
stim.contrast =  0.5  # decrease contrast
stim.contrast =  0.0  # uniform, no contrast
stim.contrast = -0.5  # slightly inverted
stim.contrast = -1.0  # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast =  1.2  # increases contrast
stim.contrast = -1.2  # inverts with increased contrast
```

property diffuseColor

Diffuse color *(r, g, b)* for the material (*psychopy.color.Color*, *ArrayLike* or *None*).

property diffuseRGB

RGB values of the diffuse color of the material (*numpy.ndarray*).

property diffuseTexture

Diffuse texture of the material (*psychopy.tools.gltools.TexImage2D* or *None*).

property emissionColor

Emission color *(r, g, b)* of the material (*psychopy.color.Color*, *ArrayLike* or *None*).

property emissionRGB

RGB values of the emission color of the material (*numpy.ndarray*).

end(*clear=True*)

Stop using this material.

Must be called after *begin* before using another material or else later drawing operations may have undefined behavior.

Upon returning, *GL_COLOR_MATERIAL* is enabled so material colors will track the current *glColor*.

Parameters

clear (*bool*) – Overwrite material state settings with default values. This ensures material colors are set to OpenGL defaults. You can forgo clearing if successive materials are used which overwrite *glMaterialfv* values for *GL_DIFFUSE*, *GL_SPECULAR*, *GL_AMBIENT*, *GL_EMISSION*, and *GL_SHININESS*. This reduces a bit of overhead if there is no need to return to default values intermittently between successive material *begin* and *end* calls. Textures and shaders previously enabled will still be disabled.

property face

Face to apply the material to (*str*). Possible values are one of *'front'*, *'back'* or *'both'*.

setAmbientColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Set the ambient color for the material.

Use this function if you wish to supress logging or apply operations on the color component.

Parameters

- **color** (ArrayLike or *~psychopy.colors.Color*) – Color to set as the ambient component of the light source.
- **colorSpace** (*str or None*) – Colorspace to use. This is only used to set the color, the value of *ambientColor* after setting uses the color space of the object.
- **operation** (*str*) – Operation string.
- **log** (*bool or None*) – Enable logging.

setDiffuseColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Set the diffuse color for the material.

Use this method if you wish to supress logging or apply operations on the color component.

Parameters

- **color** (ArrayLike or *~psychopy.colors.Color*) – Color to set as the diffuse component of the material.
- **colorSpace** (*str or None*) – Colorspace to use. This is only used to set the color, the value of *diffuseColor* after setting uses the color space of the object.
- **operation** (*str*) – Operation string.
- **log** (*bool or None*) – Enable logging.

setEmissionColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Set the emission color for the material.

Use this function if you wish to supress logging or apply operations on the color component.

Parameters

- **color** (ArrayLike or ~*psychopy.colors.Color*) – Color to set as the ambient component of the light source.
- **colorSpace** (`str or None`) – Colorspace to use. This is only used to set the color, the value of *ambientColor* after setting uses the color space of the object.
- **operation** (`str`) – Operation string.
- **log** (`bool or None`) – Enable logging.

setSpecularColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Set the diffuse color for the material. Use this function if you wish to supress logging or apply operations on the color component.

Parameters

- **color** (ArrayLike or ~*psychopy.colors.Color*) – Color to set as the specular component of the light source.
- **colorSpace** (`str or None`) – Colorspace to use. This is only used to set the color, the value of *diffuseColor* after setting uses the color space of the object.
- **operation** (`str`) – Operation string.
- **log** (`bool or None`) – Enable logging.

property shininess

Material shininess coefficient (*float*).

This is used to specify the 'tightness' of the specular highlights. Values usually range between 0 and 128, but the range depends on the specular highlight formula used by the shader.

property specularColor

Specular color *(r, g, b)* of the material (*psychopy.color.Color*, *ArrayLike* or *None*).

property specularRGB

RGB values of the specular color of the material (*numpy.ndarray*).

11.4.20 psychopy.visual.Pie

Stimulus class for drawing semi-circles and wedges.

Overview

Pie(win[, radius, start, end, edges, units, ...])	Creates a pie shape which is a circle with a wedge cut-out.
Pie.start	Start angle of the slice/wedge in degrees (*float* or *int*).
Pie.end	End angle of the slice/wedge in degrees (*float* or *int*).
Pie.radius	Radius of the shape in *units* (*float* or *int*).
Pie.units	
Pie.lineWidth	Width of the line in **pixels**.
Pie.lineColor	Alternative way of setting *borderColor*.
Pie.lineColorSpace	Deprecated, please use colorSpace to set color space for the entire object
Pie.fillColor	Set the fill color for the shape.
Pie.fillColorSpace	Deprecated, please use colorSpace to set color space for the entire object.
Pie.pos	The position of the center of the stimulus in the stimulus *units*
Pie.size	The size (width, height) of the stimulus in the stimulus *units*
Pie.ori	The orientation of the stimulus (in degrees).
Pie.opacity	Determines how visible the stimulus is relative to background.
Pie.contrast	A value that is simply multiplied by the color.
Pie.depth	DEPRECATED, depth is now controlled simply by drawing order.
Pie.interpolate	If *True* the edge of the line will be anti-aliased.
Pie.lineRGB	Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*
Pie.fillRGB	Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*
Pie.name	The name (*str*) of the object to be using during logged messages about this stim.
Pie.autoLog	Whether every change in this stimulus should be auto logged.
Pie.autoDraw	Determines whether the stimulus should be automatically drawn on every frame flip.
Pie.color	Set the color of the shape.
Pie.colorSpace	The name of the color space currently being used

Details

class psychopy.visual.pie.**Pie**(*win*, *radius=0.5*, *start=0.0*, *end=90.0*, *edges=32*, *units=''*, *lineWidth=1.5*, *lineColor=None*, *fillColor=None*, *pos=(0, 0)*, *size=1.0*, *ori=0.0*, *opacity=1.0*, *contrast=1.0*, *depth=0*, *interpolate=True*, *name=None*, *autoLog=None*, *autoDraw=False*, *colorSpace=None*, *color=False*, *fillColorSpace='rgb'*, *lineColorSpace='rgb'*, *lineRGB=False*, *fillRGB=False*)

Creates a pie shape which is a circle with a wedge cut-out. This is a lazy-imported class, therefore import using full path *from psychopy.visual.pie import Pie* when inheriting from it.

This shape is sometimes referred to as a Pac-Man shape which is often used for creating Kanizsa figures. However, the shape can be adapted for other uses.

Parameters

- **win** (*Window*) – Window this shape is being drawn to. The stimulus instance will allocate its required resources using that Windows context. In many cases, a stimulus instance cannot be drawn on different windows unless those windows share the same OpenGL context, which permits resources to be shared between them.
- **radius** (*float or int*) – Radius of the shape. Avoid using *size* for adjusting figure dimensions if radius != 0.5 which will result in undefined behavior.
- **start** (*float or int*) – Start and end angles of the filled region of the shape in degrees. Shapes are filled counter clockwise between the specified angles.
- **end** (*float or int*) – Start and end angles of the filled region of the shape in degrees. Shapes are filled counter clockwise between the specified angles.
- **edges** (*int*) – Number of edges to use when drawing the figure. A greater number of edges will result in smoother curves, but will require more time to compute.
- **units** (*str*) – Units to use when drawing. This will affect how parameters and attributes *pos*, *size* and *radius* are interpreted.
- **lineWidth** (*float*) – Width of the shape's outline.
- **lineColor** (array_like, str, *Color* or None) – Color of the shape outline and fill. If *None*, a fully transparent color is used which makes the fill or outline invisible.
- **fillColor** (array_like, str, *Color* or None) – Color of the shape outline and fill. If *None*, a fully transparent color is used which makes the fill or outline invisible.
- **pos** (*array_like*) – Initial position (*x*, *y*) of the shape on-screen relative to the origin located at the center of the window or buffer in *units*. This can be updated after initialization by setting the *pos* property. The default value is *(0.0, 0.0)* which results in no translation.
- **size** (*array_like, float, int or None*) – Width and height of the shape as *(w, h)* or *[w, h]*. If a single value is provided, the width and height will be set to the same specified value. If *None* is specified, the *size* will be set with values passed to *width* and *height*.
- **ori** (*float*) – Initial orientation of the shape in degrees about its origin. Positive values will rotate the shape clockwise, while negative values will rotate counterclockwise. The default value for *ori* is 0.0 degrees.
- **opacity** (*float*) – Opacity of the shape. A value of 1.0 indicates fully opaque and 0.0 is fully transparent (therefore invisible). Values between 1.0 and 0.0 will result in colors being blended with objects in the background. This value affects the fill (*fillColor*) and outline (*lineColor*) colors of the shape.
- **contrast** (*float*) – Contrast level of the shape (0.0 to 1.0). This value is used to modulate the contrast of colors passed to *lineColor* and *fillColor*.
- **depth** (*int*) – Depth layer to draw the shape when *autoDraw* is enabled. *DEPRECATED*
- **interpolate** (*bool*) – Enable smoothing (anti-aliasing) when drawing shape outlines. This produces a smoother (less-pixelated) outline of the shape.
- **name** (*str*) – Optional name of the stimuli for logging.
- **autoLog** (*bool*) – Enable auto-logging of events associated with this stimuli. Useful for debugging and to track timing when used in conjunction with *autoDraw*.
- **autoDraw** (*bool*) – Enable auto drawing. When *True*, the stimulus will be drawn every frame without the need to explicitly call the `draw()` method.
- **color** (array_like, str, *Color* or None) – Sets both the initial *lineColor* and *fillColor* of the shape.

- **colorSpace** (*str*) – Sets the colorspace, changing how values passed to *lineColor* and *fillColor* are interpreted.

start, end

Start and end angles of the filled region of the shape in degrees. Shapes are filled counter clockwise between the specified angles.

Type
float or int

radius

Radius of the shape. Avoid using *size* for adjusting figure dimensions if radius != 0.5 which will result in undefined behavior.

Type
float or int

property RGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type
DEPRECATED

static _calcEquilateralVertices(*edges*, *radius=0.5*)

Get vertices for an equilateral shape with a given number of sides, will assume radius is 0.5 (relative) but can be manually specified

_calcPosRendered()

DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix.

_calcSizeRendered()

DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix

_calcVertices()

Calculate the required vertices for the figure.

static _calculateMinEdges(*lineWidth*, *threshold=180*)

Calculate how many points are needed in an equilateral polygon for the gap between line rects to be < 1px and for corner angles to exceed a threshold.

In other words, how many edges does a polygon need to have to appear smooth?

lineWidth
[int, float, np.ndarray] Width of the line in pixels

threshold
[int] Maximum angle (degrees) for corners of the polygon, useful for drawing a circle. Supply 180 for no maximum angle.

_drawLegacyGL(*win*, *keepMatrix*)

Legacy draw the stimulus in its relevant window.

You must call this method after every MyWin.flip() if you want the stimulus to appear on that frame and then update the screen again.

_getDesiredRGB(*rgb*, *colorSpace*, *contrast*)

Convert color to RGB while adding contrast. Requires self.rgb, self.colorSpace and self.contrast

_getPolyAsRendered()

DEPRECATED. Return a list of vertices as rendered.

`_selectWindow(`*win*`)`

Switch drawing to the specified window. Calls the window's _setCurrent() method which handles the switch.

`_set(`*attrib*, *val*, *op=''*, *log=None*`)`

DEPRECATED since 1.80.04 + 1. Use setAttribute() and val2array() instead.

`_updateList()`

The user shouldn't need this method since it gets called after every call to .set() Chooses between using and not using shaders each call.

`_updateVertices()`

Sets Stim.verticesPix and ._borderPix from pos, size, ori, flipVert, flipHoriz

`alphaThreshold`

Threshold for alpha values.

If the alpha value of a pixel is below this threshold, the pixel will be rejected (not drawn). This can be useful for creating a mask from an image with an alpha channel. The default value is 0.0, which means that no thresholding will be applied.

`autoDraw`

Determines whether the stimulus should be automatically drawn on every frame flip.

Value should be: *True* or *False*. You do NOT need to set this on every frame flip!

`autoLog`

Whether every change in this stimulus should be auto logged.

Value should be: *True* or *False*. Set to *False* if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren't likely to be useful.

`property backColor`

Alternative way of setting fillColor

`property backColorSpace`

Deprecated, please use colorSpace to set color space for the entire object.

`property backRGB`

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type
DEPRECATED

`property backgroundColor`

Alternative way of setting fillColor

`property borderColorSpace`

Deprecated, please use colorSpace to set color space for the entire object

`property borderRGB`

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type
DEPRECATED

`closeShape`

Should the last vertex be automatically connected to the first?

If you're using *Polygon*, *Circle* or *Rect*, *closeShape=True* is assumed and shouldn't be changed.

`color`

Set the color of the shape. Sets both *fillColor* and *lineColor* simultaneously if applicable.

`property colorSpace`

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                       color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

`contains`(*x*, *y=None*, *units=None*)

Returns True if a point x,y is inside the stimulus' border.

Can accept variety of input options:

- two separate args, x and y
- one arg (list, tuple or array) containing two vals (x,y)
- **an object with a getPos() method that returns x,y, such**
  as a *`Mouse`*.

Returns *True* if the point is within the area defined either by its *border* attribute (if one defined), or its *vertices* attribute if there is no .border. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the *contains* method; the extent of the stimulus is determined purely by the size, position (pos), and orientation (ori) settings (and by the vertices for shape stimuli).

See Coder demos: shapeContains.py See Coder demos: shapeContains.py

`property contrast`

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged).
Operations supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast =  1.0  # unchanged contrast
stim.contrast =  0.5  # decrease contrast
stim.contrast =  0.0  # uniform, no contrast
stim.contrast = -0.5  # slightly inverted
stim.contrast = -1.0  # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast =  1.2  # increases contrast
stim.contrast = -1.2  # inverts with increased contrast
```

depth

DEPRECATED, depth is now controlled simply by drawing order.

doDragging()

If this stimulus is draggable, do the necessary actions on a frame flip to drag it.

draggable

Can this stimulus be dragged by a mouse click?

draw(*win=None*, *keepMatrix=False*)

Draw the stimulus in its relevant window.

You must call this method after every MyWin.flip() if you want the stimulus to appear on that frame and then update the screen again.

end

End angle of the slice/wedge in degrees (*float* or *int*).

Operations supported.

property fillColor

Set the fill color for the shape.

property fillColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property fillRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type
DEPRECATED

property flip

1x2 array for flipping vertices along each axis; set as True to flip or False to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (*._flip*) will give an array of 1s and -1s with which to multiply vertices.

property fontColor

Alternative way of setting *foreColor*.

property foreColor

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names <http://www.w3schools.com/html/html_colornames.asp>* can be used.

- *Colors by hex value*
- **numerically: (scalar or triplet) for DKL, RGB or**
  other *Color spaces*. For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue'  # (the case is actually ignored)
stim.color = '#DDA0DD'  # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0]  # if stim.colorSpace='rgb':
                # a red color in rgb space
stim.color = [0.0, 45.0, 1.0]  # if stim.colorSpace='dkl':
                # DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255]  # if stim.colorSpace='rgb255':
                # a blue stimulus using rgb255 space
stim.color = 255  # interpreted as (255, 255, 255)
                  # which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1]  # increment all guns by 1 value
stim.color *= -1  # multiply the color by -1 (which in this
                  # space inverts the contrast)
stim.color *= [0.5, 0, 1]  # decrease red, remove green, keep blue
```

You can use *setColor* if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

property foreColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property foreRGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type
DEPRECATED

interpolate

If *True* the edge of the line will be anti-aliased.

property lineColor

Alternative way of setting *borderColor*.

property lineColorSpace

Deprecated, please use colorSpace to set color space for the entire object

`property lineRGB`

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type
DEPRECATED

`lineWidth`

Width of the line in **pixels**.

Operations supported.

`name`

The name (*str*) of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = None your stimulus will be called "unnamed <type>", e.g. visual.TextStim(win) will be called "unnamed TextStim" in the logs.

`property opacity`

Determines how visible the stimulus is relative to background.

The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). *Operations* are supported. Precisely how this is used depends on the *Blend Mode*.

`ori`

The orientation of the stimulus (in degrees).

Should be a single value (*scalar*). *Operations* are supported.

Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and below zero values wrap appropriately.

`overlaps`(*polygon*)

Returns *True* if this stimulus intersects another one.

If *polygon* is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the *overlaps* method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

See coder demo, shapeContains.py

`property pos`

The position of the center of the stimulus in the stimulus *units*

value should be an *x,y-pair*. *Operations* are also supported.

Example:

```
stim.pos = (0.5, 0)  # Set slightly to the right of center
stim.pos += (0.5, -1)  # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2  # Move stim towards the center.
    Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:

```
from psychopy.tools.monitorunittools import posToPix
posPix = posToPix(stim)
```

radius

Radius of the shape in *units* (*float* or *int*).

Operations supported.

setAlphaThreshold(*value*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setAutoDraw(*value*, *log=None*)

Sets autoDraw. Usually you can use 'stim.attribute = value' syntax instead, but use this method to suppress the log message.

setAutoLog(*value=True*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setBackRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setBorderColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setBorderRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Sets both the line and fill to be the same color.

setContrast(*newContrast*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setDKL(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setDepth(*newDepth*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setEnd(*end*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setFillColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for fillColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setFillRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setForeColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for foreColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setForeRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setLMS(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setLineRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setOpacity(*newOpacity*, *operation=''*, *log=None*)

Hard setter for opacity, allows the suppression of log messages and calls the update method

setOri(*newOri*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setPos(*newPos*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setRadius(*end*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setSize(*newSize*, *operation=''*, *units=None*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setStart(*start*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setVertices(*value=None*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

property size

The size (width, height) of the stimulus in the stimulus *units*

Value should be *x,y-pair*, *scalar* (applies to both dimensions) or None (resets to default). *Operations* are supported.

Sizes can be negative (causing a mirror-image reversal) and can extend beyond the window.

Example:

```
stim.size = 0.8  # Set size to (xsize, ysize) = (0.8, 0.8)
print(stim.size)  # Outputs array([0.8, 0.8])
stim.size += (0.5, -0.5)  # make wider and flatter: (1.3, 0.3)
```

Tip: if you can see the actual pixel range this corresponds to by looking at *stim._sizeRendered*

start

Start angle of the slice/wedge in degrees (*float* or *int*).

Operations supported.

updateColors()

Placeholder method to update colours when set externally, for example updating the *pallette* attribute of a textbox

updateOpacity()

Placeholder method to update colours when set externally, for example updating the *pallette* attribute of a textbox.

property verticesPix

This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

property win

The `Window` object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1  # stimulus will be drawn in win1
stim.draw()  # stimulus is now drawn to win1
stim.win = win2  # stimulus will be drawn in win2
stim.draw()  # it is now drawn in win2
win1.flip(waitBlanking=False)  # do not wait for next
             # monitor update
win2.flip()  # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

11.4.21 PlaneStim

Attributes

PlaneStim(win[, size, pos, ori, color, ...])	Class for drawing planes.

Details

class psychopy.visual.**PlaneStim**(*win*, *size=(0.5, 0.5)*, *pos=(0.0, 0.0, 0.0)*, *ori=(0.0, 0.0, 0.0, 1.0)*, *color=(0.0, 0.0, 0.0)*, *colorSpace='rgb'*, *contrast=1.0*, *opacity=1.0*, *useMaterial=None*, *textureScale=None*, *name=''*, *autoLog=True*)

Class for drawing planes. This is a lazy-imported class, therefore import using full path *from psychopy.visual.stim3d import PlaneStim* when inheriting from it.

Draws a plane with dimensions specified by *size* (length, width) in scene units.

Calling the *draw* method will render the plane to the current buffer. The render target (FBO or back buffer) must have a depth buffer attached to it for the object to be rendered correctly. Shading is used if the current window has light sources defined and lighting is enabled (by setting *useLights=True* before drawing the stimulus).

> ⚠ **Warning**
>
> This class is experimental and may result in undefined behavior.

Parameters

- **win** (*~psychopy.visual.Window*) – Window this stimulus is associated with. Stimuli cannot be shared across windows unless they share the same context.
- **size** (`tuple or float`) – Dimensions of the mesh. If a single value is specified, the plane will be a square. Provide a tuple of floats to specify the width and length of the plane (eg. *size=(0.2, 1.3)*).
- **pos** (`array_like`) – Position vector *[x, y, z]* for the origin of the rigid body.
- **ori** (`array_like`) – Orientation quaternion *[x, y, z, w]* where *x*, *y*, *z* are imaginary and *w* is real. If you prefer specifying rotations in axis-angle format, call *setOriAxisAngle* after initialization. By default, the plane is oriented with normal facing the +Z axis of the scene.
- **useMaterial** (`PhongMaterial, optional`) – Material to use. The material can be configured by accessing the *material* attribute after initialization. If not material is specified, the diffuse and ambient color of the shape will track the current color specified by *glColor*.
- **colorSpace** (`str`) – Colorspace of *color* to use.
- **contrast** (`float`) – Contrast of the stimulus, value modulates the *color*.
- **opacity** (`float`) – Opacity of the stimulus ranging from 0.0 to 1.0. Note that transparent objects look best when rendered from farthest to nearest.
- **textureScale** (`array_like or float, optional`) – Scaling factors for texture coordinates (sx, sy). By default, a factor of 1 will have the entire texture cover the surface of the mesh. If a single number is provided, the texture will be scaled uniformly.
- **name** (`str`) – Name of this object for logging purposes.
- **autoLog** (`bool`) – Enable automatic logging on attribute changes.

property RGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type

DEPRECATED

_createVAO(*vertices*, *textureCoords*, *normals*, *faces*)

Create a vertex array object for handling vertex attribute data.

_getDesiredRGB(*rgb*, *colorSpace*, *contrast*)

Convert color to RGB while adding contrast. Requires self.rgb, self.colorSpace and self.contrast

_selectWindow(*win*)

Switch drawing to the specified window. Calls the window's _setCurrent() method which handles the switch.

_updateList()

The user shouldn't need this method since it gets called after every call to .set() Chooses between using and not using shaders each call.

property anchor

property backColor

Alternative way of setting fillColor

property backColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property backRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type
DEPRECATED

property backgroundColor

Alternative way of setting fillColor

property borderColor

property borderColorSpace

Deprecated, please use colorSpace to set color space for the entire object

property borderRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type
DEPRECATED

borderWidth

property color

Alternative way of setting *foreColor*.

property colorSpace

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                       color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

property contrast

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged).
Operations supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast =  1.0  # unchanged contrast
stim.contrast =  0.5  # decrease contrast
stim.contrast =  0.0  # uniform, no contrast
stim.contrast = -0.5  # slightly inverted
stim.contrast = -1.0  # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast =  1.2  # increases contrast
stim.contrast = -1.2  # inverts with increased contrast
```

draw(*win=None*)

Draw the stimulus.

This should work for stimuli using a single VAO and material. More complex stimuli with multiple materials should override this method to correctly handle that case.

Parameters

win (*~psychopy.visual.Window*) – Window this stimulus is associated with. Stimuli cannot be shared across windows unless they share the same context.

property fillColor

Set the fill color for the shape.

property fillColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property fillRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type

DEPRECATED

property flip

1x2 array for flipping vertices along each axis; set as True to flip or False to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (*._flip*) will give an array of 1s and -1s with which to multiply vertices.

property flipHoriz

property flipVert

property fontColor

Alternative way of setting *foreColor*.

property foreColor

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names <http://www.w3schools.com/html/html_colornames.asp>* can be used.
- *Colors by hex value*

- **numerically: (scalar or triplet) for DKL, RGB or**
  other *Color spaces*. For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue'  # (the case is actually ignored)
stim.color = '#DDA0DD'  # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0]  # if stim.colorSpace='rgb':
                # a red color in rgb space
stim.color = [0.0, 45.0, 1.0]  # if stim.colorSpace='dkl':
                # DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255]  # if stim.colorSpace='rgb255':
                # a blue stimulus using rgb255 space
stim.color = 255  # interpreted as (255, 255, 255)
                  # which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1]  # increment all guns by 1 value
stim.color *= -1  # multiply the color by -1 (which in this
                    # space inverts the contrast)
stim.color *= [0.5, 0, 1]  # decrease red, remove green, keep blue
```

You can use *setColor* if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

property foreColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property foreRGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type
DEPRECATED

getOri()

getOriAxisAngle(*degrees=True*)

Get the axis and angle of rotation for the 3D stimulus. Converts the orientation defined by the *ori* quaternion to and axis-angle representation.

Parameters
degrees (*bool, optional*) – Specify `True` if *angle* is in degrees, or else it will be treated as radians. Default is `True`.

Returns
Axis *[rx, ry, rz]* and angle.

Return type
tuple

getPos()

getRayIntersectBounds(*rayOrig*, *rayDir*)

Get the point which a ray intersects the bounding box of this mesh.

Parameters

- **rayOrig** (*array_like*) – Origin of the ray in space [x, y, z].
- **rayDir** (*array_like*) – Direction vector of the ray [x, y, z], should be normalized.

Returns
Coordinate in world space of the intersection and distance in scene units from *rayOrig*. Returns *None* if there is no intersection.

Return type
tuple

property height

isVisible()

Check if the object is visible to the observer.

Test if a pose's bounding box or position falls outside of an eye's view frustum.

Poses can be assigned bounding boxes which enclose any 3D models associated with them. A model is not visible if all the corners of the bounding box fall outside the viewing frustum. Therefore any primitives (i.e. triangles) associated with the pose can be culled during rendering to reduce CPU/GPU workload.

Returns
True if the object's bounding box is visible.

Return type
bool

Examples

You can avoid running draw commands if the object is not visible by doing a visibility test first:

```python
if myStim.isVisible():
    myStim.draw()
```

property lineColor

Alternative way of setting *borderColor*.

property lineColorSpace

Deprecated, please use colorSpace to set color space for the entire object

property lineRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type
DEPRECATED

lineWidth

property ori

Orientation quaternion (X, Y, Z, W).

property pos

Position vector (X, Y, Z).

setAnchor(*value*, *log=None*)

setBackColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setBackRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setBackgroundColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setBorderColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setBorderRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setBorderWidth(*newWidth*, *operation=''*, *log=None*)

setColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setContrast(*newContrast*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setDKL(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setFillColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for fillColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setFillRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setFontColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setForeColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for foreColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setForeRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setLMS(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setLineColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setLineRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setLineWidth(*newWidth*, *operation=''*, *log=None*)

setOri(*ori*)

setOriAxisAngle(*axis*, *angle*, *degrees=True*)

Set the orientation of the 3D stimulus using an *axis* and *angle*. This sets the quaternion at *ori*.

Parameters

- **axis** (*array_like*) – Axis of rotation [rx, ry, rz].
- **angle** (*float*) – Angle of rotation.
- **degrees** (*bool, optional*) – Specify True if *angle* is in degrees, or else it will be treated as radians. Default is True.

setPos(*pos*)

setRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

property size

property thePose

The pose of the rigid body. This is a class which has *pos* and *ori* attributes.

units

None, 'norm', 'cm', 'deg', 'degFlat', 'degFlatPos', or 'pix'

If None then the current units of the *Window* will be used. See *Units for the window and stimuli* for explanation of other options.

Note that when you change units, you don't change the stimulus parameters and it is likely to change appearance. Example:

```
# This stimulus is 20% wide and 50% tall with respect to window
stim = visual.PatchStim(win, units='norm', size=(0.2, 0.5)

# This stimulus is 0.2 degrees wide and 0.5 degrees tall.
stim.units = 'deg'
```

updateColors()

Placeholder method to update colours when set externally, for example updating the *pallette* attribute of a textbox

property vertices

property width

property win

The *Window* object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1  # stimulus will be drawn in win1
stim.draw()  # stimulus is now drawn to win1
stim.win = win2  # stimulus will be drawn in win2
stim.draw()  # it is now drawn in win2
win1.flip(waitBlanking=False)  # do not wait for next
             # monitor update
win2.flip()  # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

11.4.22 psychopy.visual.Polygon

Stimulus class for drawing regular polygons.

Overview

Polygon(win[, edges, radius, units, ...])	Creates a regular polygon (triangles, pentagons, ...).
Polygon.radius	float, int, tuple, list or 2x1 array Radius of the Polygon (distance from the center to the corners).
Polygon.edges	Number of edges of the polygon.
Polygon.units	
Polygon.lineWidth	Width of the line in **pixels**.
Polygon.lineColor	Alternative way of setting *borderColor*.
Polygon.lineColorSpace	Deprecated, please use colorSpace to set color space for the entire object
Polygon.fillColor	Set the fill color for the shape.
Polygon.fillColorSpace	Deprecated, please use colorSpace to set color space for the entire object.
Polygon.pos	The position of the center of the stimulus in the stimulus *units*
Polygon.size	The size (width, height) of the stimulus in the stimulus *units*
Polygon.ori	The orientation of the stimulus (in degrees).
Polygon.opacity	Determines how visible the stimulus is relative to background.
Polygon.contrast	A value that is simply multiplied by the color.
Polygon.depth	DEPRECATED, depth is now controlled simply by drawing order.
Polygon.interpolate	If *True* the edge of the line will be anti-aliased.
Polygon.lineRGB	Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*
Polygon.fillRGB	Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*
Polygon.name	The name (*str*) of the object to be using during logged messages about this stim.
Polygon.autoLog	Whether every change in this stimulus should be auto logged.
Polygon.autoDraw	Determines whether the stimulus should be automatically drawn on every frame flip.
Polygon.color	Set the color of the shape.
Polygon.colorSpace	The name of the color space currently being used

Details

class psychopy.visual.polygon.**Polygon**(*win*, *edges=3*, *radius=0.5*, *units=''*, *lineWidth=1.5*, *lineColor='white'*, *fillColor='white'*, *pos=(0, 0)*, *size=1.0*, *anchor=None*, *ori=0.0*, *opacity=None*, *contrast=1.0*, *depth=0*, *interpolate=True*, *draggable=False*, *name=None*, *autoLog=None*, *autoDraw=False*, *colorSpace='rgb'*, *color=undefined*, *fillColorSpace=undefined*, *lineColorSpace=undefined*, *lineRGB=undefined*, *fillRGB=undefined*)

Creates a regular polygon (triangles, pentagons, …). This is a lazy-imported class, therefore import using full path *from psychopy.visual.polygon import Polygon* when inheriting from it.

This class is a special case of a `ShapeStim` that accepts the same parameters except *closeShape* and *vertices*.

Parameters

- **win** (*Window*) – Window this shape is being drawn to. The stimulus instance will allocate its required resources using that Windows context. In many cases, a stimulus instance cannot be drawn on different windows unless those windows share the same OpenGL context, which permits resources to be shared between them.
- **edges** (*int*) – Number of sides for the polygon (for instance, *edges=3* will result in a triangle).
- **radius** (*float*) – Initial radius of the polygon in *units*. This specifies how far out to place the corners (vertices) of the shape.
- **units** (*str*) – Units to use when drawing. This will affect how parameters and attributes *pos*, *size* and *radius* are interpreted.
- **lineWidth** (*float*) – Width of the polygon's outline.
- **lineColor** (array_like, str, *Color* or *None*) – Color of the shape's outline and fill. If *None*, a fully transparent color is used which makes the fill or outline invisible.
- **fillColor** (array_like, str, *Color* or *None*) – Color of the shape's outline and fill. If *None*, a fully transparent color is used which makes the fill or outline invisible.
- **pos** (*array_like*) – Initial position (*x*, *y*) of the shape on-screen relative to the origin located at the center of the window or buffer in *units*. This can be updated after initialization by setting the *pos* property. The default value is *(0.0, 0.0)* which results in no translation.
- **size** (*float or array_like*) – Initial scale factor for adjusting the size of the shape. A single value (*float*) will apply uniform scaling, while an array (*sx*, *sy*) will result in anisotropic scaling in the horizontal (*sx*) and vertical (*sy*) direction. Providing negative values to *size* will cause the shape being mirrored. Scaling can be changed by setting the *size* property after initialization. The default value is *1.0* which results in no scaling.
- **ori** (*float*) – Initial orientation of the shape in degrees about its origin. Positive values will rotate the shape clockwise, while negative values will rotate counterclockwise. The default value for *ori* is 0.0 degrees.
- **opacity** (*float*) – Opacity of the shape. A value of 1.0 indicates fully opaque and 0.0 is fully transparent (therefore invisible). Values between 1.0 and 0.0 will result in colors being blended with objects in the background. This value affects the fill (*fillColor*) and outline (*lineColor*) colors of the shape.
- **contrast** (*float*) – Contrast level of the shape (0.0 to 1.0). This value is used to modulate the contrast of colors passed to *lineColor* and *fillColor*.
- **depth** (*int*) – Depth layer to draw the stimulus when *autoDraw* is enabled.

- **interpolate** (*bool*) – Enable smoothing (anti-aliasing) when drawing shape outlines. This produces a smoother (less-pixelated) outline of the shape.
- **name** (*str*) – Optional name of the stimuli for logging.
- **autoLog** (*bool*) – Enable auto-logging of events associated with this stimuli. Useful for debugging and to track timing when used in conjunction with *autoDraw*.
- **autoDraw** (*bool*) – Enable auto drawing. When *True*, the stimulus will be drawn every frame without the need to explicitly call the `draw()` method.
- **color** (array_like, str, `Color` or *None*) – Sets both the initial *lineColor* and *fillColor* of the shape.
- **colorSpace** (*str*) – Sets the colorspace, changing how values passed to *lineColor* and *fillColor* are interpreted.
- **draggable** (*bool*) – Can this stimulus be dragged by a mouse click?

property RGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type
DEPRECATED

static _calcEquilateralVertices(*edges*, *radius=0.5*)

Get vertices for an equilateral shape with a given number of sides, will assume radius is 0.5 (relative) but can be manually specified

_calcPosRendered()

DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix.

_calcSizeRendered()

DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix

static _calculateMinEdges(*lineWidth*, *threshold=180*)

Calculate how many points are needed in an equilateral polygon for the gap between line rects to be < 1px and for corner angles to exceed a threshold.

In other words, how many edges does a polygon need to have to appear smooth?

lineWidth
[int, float, np.ndarray] Width of the line in pixels

threshold
[int] Maximum angle (degrees) for corners of the polygon, useful for drawing a circle. Supply 180 for no maximum angle.

_drawLegacyGL(*win*, *keepMatrix*)

Legacy draw the stimulus in the relevant window.

You must call this method after every *win.flip()* if you want the stimulus to appear on that frame and then update the screen again.

_getDesiredRGB(*rgb*, *colorSpace*, *contrast*)

Convert color to RGB while adding contrast. Requires self.rgb, self.colorSpace and self.contrast

_getPolyAsRendered()

DEPRECATED. Return a list of vertices as rendered.

`_legacyTesselate`(*newVertices*)

Legacy tessellation method for ShapeStim.

`_selectWindow`(*win*)

Switch drawing to the specified window. Calls the window's _setCurrent() method which handles the switch.

`_set`(*attrib*, *val*, *op=''*, *log=None*)

DEPRECATED since 1.80.04 + 1. Use setAttribute() and val2array() instead.

`_tesselate`(*newVertices*)

Set the *.vertices* and *.border* to new values, invoking tessellation.

Parameters

`newVertices` (*array_like*) – Nx2 array of points (eg., *[[-0.5, 0], [0, 0.5], [0.5, 0]*).

`_updateList`()

The user shouldn't need this method since it gets called after every call to .set() Chooses between using and not using shaders each call.

`_updateVertices`()

Sets Stim.verticesPix and ._borderPix from pos, size, ori, flipVert, flipHoriz

`alphaThreshold`

Threshold for alpha values.

If the alpha value of a pixel is below this threshold, the pixel will be rejected (not drawn). This can be useful for creating a mask from an image with an alpha channel. The default value is 0.0, which means that no thresholding will be applied.

`autoDraw`

Determines whether the stimulus should be automatically drawn on every frame flip.

Value should be: *True* or *False*. You do NOT need to set this on every frame flip!

`autoLog`

Whether every change in this stimulus should be auto logged.

Value should be: *True* or *False*. Set to *False* if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren't likely to be useful.

`property backColor`

Alternative way of setting fillColor

`property backColorSpace`

Deprecated, please use colorSpace to set color space for the entire object.

`property backRGB`

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type

DEPRECATED

`property backgroundColor`

Alternative way of setting fillColor

`property borderColorSpace`

Deprecated, please use colorSpace to set color space for the entire object

property borderRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type

DEPRECATED

closeShape

Should the last vertex be automatically connected to the first?

If you're using *Polygon*, *Circle* or *Rect*, *closeShape=True* is assumed and shouldn't be changed.

color

Set the color of the shape. Sets both *fillColor* and *lineColor* simultaneously if applicable.

property colorSpace

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                       color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

contains(*x*, *y=None*, *units=None*)

Returns True if a point x,y is inside the stimulus' border.

Can accept variety of input options:

- two separate args, x and y
- one arg (list, tuple or array) containing two vals (x,y)
- **an object with a getPos() method that returns x,y, such**
  as a `Mouse`.

Returns *True* if the point is within the area defined either by its *border* attribute (if one defined), or its *vertices* attribute if there is no .border. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the *contains* method; the extent of the stimulus is determined purely by the size, position (pos), and orientation (ori) settings (and by the vertices for shape stimuli).

See Coder demos: shapeContains.py See Coder demos: shapeContains.py

property contrast

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged).
Operations supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast =  1.0  # unchanged contrast
stim.contrast =  0.5  # decrease contrast
stim.contrast =  0.0  # uniform, no contrast
stim.contrast = -0.5  # slightly inverted
stim.contrast = -1.0  # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast =  1.2  # increases contrast
stim.contrast = -1.2  # inverts with increased contrast
```

depth

DEPRECATED, depth is now controlled simply by drawing order.

doDragging()

If this stimulus is draggable, do the necessary actions on a frame flip to drag it.

draggable

Can this stimulus be dragged by a mouse click?

draw(*win=None*, *keepMatrix=False*)

Draw the stimulus in the relevant window.

You must call this method after every *win.flip()* if you want the stimulus to appear on that frame and then update the screen again.

Parameters

- **win** (*Window*, optional) – Window to draw the stimulus in. If not specified, the stimulus will be drawn in the window specified at initialization.
- **keepMatrix** (*bool, optional*) – *DEPRECATED* If *True*, the current transformation matrix will be preserved. This is useful when drawing multiple stimuli with the same transformation matrix. Default is *False*.

edges

Number of edges of the polygon. Floats are rounded to int.

Operations supported.

property fillColor

Set the fill color for the shape.

property fillColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property fillRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type

DEPRECATED

property flip

1x2 array for flipping vertices along each axis; set as True to flip or False to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (*._flip*) will give an array of 1s and -1s with which to multiply vertices.

property fontColor

Alternative way of setting *foreColor*.

property foreColor

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names <http://www.w3schools.com/html/html_colornames.asp>* can be used.
- *Colors by hex value*
- **numerically: (scalar or triplet) for DKL, RGB or**
  other *Color spaces*. For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue'  # (the case is actually ignored)
stim.color = '#DDA0DD'  # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0]  # if stim.colorSpace='rgb':
                # a red color in rgb space
stim.color = [0.0, 45.0, 1.0]  # if stim.colorSpace='dkl':
                # DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255]  # if stim.colorSpace='rgb255':
                # a blue stimulus using rgb255 space
stim.color = 255  # interpreted as (255, 255, 255)
                  # which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1]  # increment all guns by 1 value
stim.color *= -1  # multiply the color by -1 (which in this
                    # space inverts the contrast)
stim.color *= [0.5, 0, 1]  # decrease red, remove green, keep blue
```

You can use *setColor* if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

property foreColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

`property foreRGB`

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type
DEPRECATED

`interpolate`

If *True* the edge of the line will be anti-aliased.

`property lineColor`

Alternative way of setting *borderColor*.

`property lineColorSpace`

Deprecated, please use colorSpace to set color space for the entire object

`property lineRGB`

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type
DEPRECATED

`lineWidth`

Width of the line in **pixels**.

Operations supported.

`name`

The name (*str*) of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = None your stimulus will be called "unnamed <type>", e.g. visual.TextStim(win) will be called "unnamed TextStim" in the logs.

`property opacity`

Determines how visible the stimulus is relative to background.

The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). *Operations* are supported. Precisely how this is used depends on the *Blend Mode*.

`ori`

The orientation of the stimulus (in degrees).

Should be a single value (*scalar*). *Operations* are supported.

Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and below zero values wrap appropriately.

`overlaps`(*polygon*)

Returns *True* if this stimulus intersects another one.

If *polygon* is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the *overlaps* method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

See coder demo, shapeContains.py

property pos

The position of the center of the stimulus in the stimulus *units*

value should be an *x,y-pair*. *Operations* are also supported.

Example:

```
stim.pos = (0.5, 0)  # Set slightly to the right of center
stim.pos += (0.5, -1)  # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2  # Move stim towards the center.
    Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:

```
from psychopy.tools.monitorunittools import posToPix
posPix = posToPix(stim)
```

radius

float, int, tuple, list or 2x1 array Radius of the Polygon (distance from the center to the corners). May be a -2tuple or list to stretch the polygon asymmetrically.

Operations supported.

Usually there's a setAttribute(value, log=False) method for each attribute. Use this if you want to disable logging.

setAlphaThreshold(*value*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setAutoDraw(*value*, *log=None*)

Sets autoDraw. Usually you can use 'stim.attribute = value' syntax instead, but use this method to suppress the log message.

setAutoLog(*value=True*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setBackRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setBorderColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setBorderRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Sets both the line and fill to be the same color.

setContrast(*newContrast*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setDKL(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute `setDepth`(*newDepth*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

`setEdges`(*edges*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

`setFillColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for fillColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

`setFillRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

`setForeColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for foreColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

`setForeRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

`setLMS`(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

`setLineRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

`setNVertices`(*nVerts*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

`setOpacity`(*newOpacity*, *operation=''*, *log=None*)

Hard setter for opacity, allows the suppression of log messages and calls the update method

`setOri`(*newOri*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

`setPos`(*newPos*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

`setRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

`setRadius`(*radius*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

`setSize`(*newSize*, *operation=''*, *units=None*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

`setVertices`(*value=None*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

property size

The size (width, height) of the stimulus in the stimulus *units*

Value should be *x,y-pair*, *scalar* (applies to both dimensions) or None (resets to default). *Operations* are supported.

Sizes can be negative (causing a mirror-image reversal) and can extend beyond the window.

Example:

```
stim.size = 0.8  # Set size to (xsize, ysize) = (0.8, 0.8)
print(stim.size)  # Outputs array([0.8, 0.8])
stim.size += (0.5, -0.5)  # make wider and flatter: (1.3, 0.3)
```

Tip: if you can see the actual pixel range this corresponds to by looking at *stim._sizeRendered*

updateColors()

Placeholder method to update colours when set externally, for example updating the *pallette* attribute of a textbox

updateOpacity()

Placeholder method to update colours when set externally, for example updating the *pallette* attribute of a textbox.

property vertices

A list of lists or a numpy array (Nx2) specifying xy positions of each vertex, relative to the center of the field.

Assigning to vertices can be slow if there are many vertices.

Operations supported with *.setVertices()*.

property verticesPix

This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

property win

The `Window` object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1  # stimulus will be drawn in win1
stim.draw()  # stimulus is now drawn to win1
stim.win = win2  # stimulus will be drawn in win2
stim.draw()  # it is now drawn in win2
win1.flip(waitBlanking=False)  # do not wait for next
             # monitor update
win2.flip()  # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

11.4.23 `psychopy.visual.Progress`

Stimulus class for drawing progress bars. This is a lazy-imported class, therefore import using full path *from psychopy.visual.progress import Progress* when inheriting from it.

Overview

Progress(win[, name, progress, direction, ...])	A basic progress bar, consisting of two rectangles: A background and a foreground.
Progress.name	The name (*str*) of the object to be using during logged messages about this stim.
Progress.progress	How far along the progress bar is
Progress.direction	What direction is this progress bar progressing in? Use in combination with "anchor" to control which direction the bar fills up from.
Progress.pos	The position of the center of the stimulus in the stimulus *units*
Progress.size	The size (width, height) of the stimulus in the stimulus *units*
Progress.anchor	
Progress.units	
Progress.barColor	
Progress.backColor	Alternative way of setting fillColor
Progress.borderColor	
Progress.colorSpace	The name of the color space currently being used
Progress.lineWidth	Width of the line in **pixels**.
Progress.opacity	Determines how visible the stimulus is relative to background.

Details

class psychopy.visual.progress.**Progress**(*win*, *name='pb'*, *progress=0*, *direction='horizontal'*, *pos=(-0.5, 0)*, *size=(1, 0.1)*, *anchor='center-left'*, *units=None*, *barColor=False*, *backColor=False*, *borderColor=False*, *colorSpace=None*, *lineWidth=1.5*, *opacity=1.0*, *ori=0.0*, *depth=0*, *autoLog=None*, *autoDraw=False*, *foreColor='white'*, *fillColor=False*, *lineColor='white'*)

A basic progress bar, consisting of two rectangles: A background and a foreground. The foreground rectangle fill the background as progress approaches 1.

Parameters

- **win** (*Window*) – Window this shape is being drawn to. The stimulus instance will allocate its required resources using that Windows context. In many cases, a stimulus instance cannot be drawn on different windows unless those windows share the same OpenGL context, which permits resources to be shared between them.
- **name** (*str*) – Name to refer to this Progress bar by
- **progress** (*float*) – Value between 0 (not started) and 1 (complete) to set the progress bar to.

- **direction** (*str*) – Which dimension the bar should fill along, either "horizontal" (also accepts "horiz", "x" or 0) or "vertical" (also accepts "vert", "y" or 1)
- **pos** (*array_like*) – Initial position (*x*, *y*) of the shape on-screen relative to the origin located at the center of the window or buffer in *units*. This can be updated after initialization by setting the *pos* property. The default value is *(0.0, 0.0)* which results in no translation.
- **size** (*array_like, float, int or None*) – Width and height of the shape as *(w, h)* or *[w, h]*. If a single value is provided, the width and height will be set to the same specified value. If *None* is specified, the *size* will be set with values passed to *width* and *height*.
- **anchor** (*str*) – Point within the shape where size and pos are set from. This also affects where the progress bar fills up from (e.g. if anchor is "left" and direction is "horizontal", then the bar will fill from left to right)
- **units** (*str*) – Units to use when drawing. This will affect how parameters and attributes *pos* and *size* are interpreted.
- **barColor** (array_like, str, *Color* or None) – Color of the full part of the progress bar.
- **foreColor** (array_like, str, *Color* or None) – Color of the full part of the progress bar.
- **backColor** (array_like, str, *Color* or None) – Color of the empty part of the progress bar.
- **fillColor** (array_like, str, *Color* or None) – Color of the empty part of the progress bar.
- **borderColor** (array_like, str, *Color* or None) – Color of the outline around the outside of the progress bar.
- **lineColor** (array_like, str, *Color* or None) – Color of the outline around the outside of the progress bar.
- **colorSpace** (*str*) – Sets the colorspace, changing how values passed to *foreColor*, *lineColor* and *fillColor* are interpreted.
- **lineWidth** (*float*) – Width of the shape outline.
- **opacity** (*float*) – Opacity of the shape. A value of 1.0 indicates fully opaque and 0.0 is fully transparent (therefore invisible). Values between 1.0 and 0.0 will result in colors being blended with objects in the background.

property RGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type

DEPRECATED

static _calcEquilateralVertices(*edges*, *radius=0.5*)

Get vertices for an equilateral shape with a given number of sides, will assume radius is 0.5 (relative) but can be manually specified

_calcPosRendered()

DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix.

_calcSizeRendered()

DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix

static _calculateMinEdges(*lineWidth*, *threshold=180*)

Calculate how many points are needed in an equilateral polygon for the gap between line rects to be < 1px and for corner angles to exceed a threshold.

In other words, how many edges does a polygon need to have to appear smooth?

lineWidth
[int, float, np.ndarray] Width of the line in pixels

threshold
[int] Maximum angle (degrees) for corners of the polygon, useful for drawing a circle. Supply 180 for no maximum angle.

`_drawLegacyGL`(*win*, *keepMatrix*)

Legacy draw the stimulus in the relevant window.

You must call this method after every *win.flip()* if you want the stimulus to appear on that frame and then update the screen again.

`_getDesiredRGB`(*rgb*, *colorSpace*, *contrast*)

Convert color to RGB while adding contrast. Requires self.rgb, self.colorSpace and self.contrast

`_getPolyAsRendered`()

DEPRECATED. Return a list of vertices as rendered.

`_legacyTesselate`(*newVertices*)

Legacy tessellation method for ShapeStim.

static `_sanitizeDirection`(*direction*)

Take a value indicating direction and convert it to a human readable string

`_selectWindow`(*win*)

Switch drawing to the specified window. Calls the window's _setCurrent() method which handles the switch.

`_set`(*attrib*, *val*, *op=''*, *log=None*)

DEPRECATED since 1.80.04 + 1. Use setAttribute() and val2array() instead.

`_tesselate`(*newVertices*)

Set the *.vertices* and *.border* to new values, invoking tessellation.

Parameters
`newVertices` (*array_like*) – Nx2 array of points (eg., *[[-0.5, 0], [0, 0.5], [0.5, 0]*).

`_updateList`()

The user shouldn't need this method since it gets called after every call to .set() Chooses between using and not using shaders each call.

`_updateVertices`()

Sets Stim.verticesPix and ._borderPix from pos, size, ori, flipVert, flipHoriz

`alphaThreshold`

Threshold for alpha values.

If the alpha value of a pixel is below this threshold, the pixel will be rejected (not drawn). This can be useful for creating a mask from an image with an alpha channel. The default value is 0.0, which means that no thresholding will be applied.

`autoDraw`

Determines whether the stimulus should be automatically drawn on every frame flip.

Value should be: *True* or *False*. You do NOT need to set this on every frame flip!

`autoLog`

Whether every change in this stimulus should be auto logged.

Value should be: *True* or *False*. Set to *False* if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren't likely to be useful.

property backColor

Alternative way of setting fillColor

property backColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property backRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type
DEPRECATED

property backgroundColor

Alternative way of setting fillColor

property borderColorSpace

Deprecated, please use colorSpace to set color space for the entire object

property borderRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type
DEPRECATED

`closeShape`

Should the last vertex be automatically connected to the first?

If you're using *Polygon*, *Circle* or *Rect*, *closeShape=True* is assumed and shouldn't be changed.

`color`

Set the color of the shape. Sets both *fillColor* and *lineColor* simultaneously if applicable.

property colorSpace

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                       color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

contains(*x*, *y=None*, *units=None*)

Returns True if a point x,y is inside the stimulus' border.

Can accept variety of input options:

- two separate args, x and y
- one arg (list, tuple or array) containing two vals (x,y)
- **an object with a getPos() method that returns x,y, such**
  as a *Mouse*.

Returns *True* if the point is within the area defined either by its *border* attribute (if one defined), or its *vertices* attribute if there is no .border. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the *contains* method; the extent of the stimulus is determined purely by the size, position (pos), and orientation (ori) settings (and by the vertices for shape stimuli).

See Coder demos: shapeContains.py See Coder demos: shapeContains.py

property contrast

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged).
Operations supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast =  1.0  # unchanged contrast
stim.contrast =  0.5  # decrease contrast
stim.contrast =  0.0  # uniform, no contrast
stim.contrast = -0.5  # slightly inverted
stim.contrast = -1.0  # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast =  1.2  # increases contrast
stim.contrast = -1.2  # inverts with increased contrast
```

depth

DEPRECATED, depth is now controlled simply by drawing order.

direction

What direction is this progress bar progressing in? Use in combination with "anchor" to control which direction the bar fills up from.

Parameters

value (*str*, *int*, *bool*) – Is progress bar horizontal or vertical? Accepts the following values: * horizontal: "horizontal", "horiz", "x", "0", "False", 0, False * vertical: "vertical", "vert", "y", "1", "True", 1, True

doDragging()

If this stimulus is draggable, do the necessary actions on a frame flip to drag it.

`draggable`

Can this stimulus be dragged by a mouse click?

`draw`(*win=None*, *keepMatrix=False*)

Draw the stimulus in the relevant window.

You must call this method after every *win.flip()* if you want the stimulus to appear on that frame and then update the screen again.

Parameters

- **win** (*Window*, optional) – Window to draw the stimulus in. If not specified, the stimulus will be drawn in the window specified at initialization.
- **keepMatrix** (`bool, optional`) – *DEPRECATED* If *True*, the current transformation matrix will be preserved. This is useful when drawing multiple stimuli with the same transformation matrix. Default is *False*.

`property fillColor`

Set the fill color for the shape.

`property fillColorSpace`

Deprecated, please use colorSpace to set color space for the entire object.

`property fillRGB`

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type

DEPRECATED

`property flip`

1x2 array for flipping vertices along each axis; set as True to flip or False to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (*._flip*) will give an array of 1s and -1s with which to multiply vertices.

`property fontColor`

Alternative way of setting *foreColor*.

`property foreColor`

Color of the full part of the progress bar.

`property foreColorSpace`

Deprecated, please use colorSpace to set color space for the entire object.

`property foreRGB`

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type

DEPRECATED

`interpolate`

If *True* the edge of the line will be anti-aliased.

`property lineColor`

Alternative way of setting *borderColor*.

`property lineColorSpace`

Deprecated, please use colorSpace to set color space for the entire object

property lineRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type
DEPRECATED

lineWidth

Width of the line in **pixels**.

Operations supported.

name

The name (*str*) of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = None your stimulus will be called "unnamed <type>", e.g. visual.TextStim(win) will be called "unnamed TextStim" in the logs.

property opacity

Determines how visible the stimulus is relative to background.

The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). *Operations* are supported. Precisely how this is used depends on the *Blend Mode*.

ori

overlaps(*polygon*)

Returns *True* if this stimulus intersects another one.

If *polygon* is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the *overlaps* method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

See coder demo, shapeContains.py

property pos

The position of the center of the stimulus in the stimulus *units*

value should be an *x,y-pair*. *Operations* are also supported.

Example:

```
stim.pos = (0.5, 0)  # Set slightly to the right of center
stim.pos += (0.5, -1)  # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2  # Move stim towards the center.
    Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:

```
from psychopy.tools.monitorunittools import posToPix
posPix = posToPix(stim)
```

progress

How far along the progress bar is

Parameters
`value` (*float*) – Between 0 (not complete) and 1 (fully complete)

`setAlphaThreshold`(*value*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

`setAutoDraw`(*value*, *log=None*)

Sets autoDraw. Usually you can use 'stim.attribute = value' syntax instead, but use this method to suppress the log message.

`setAutoLog`(*value=True*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

`setBackRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

`setBorderColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

`setBorderRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

`setColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Sets both the line and fill to be the same color.

`setContrast`(*newContrast*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

`setDKL`(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

`setDepth`(*newDepth*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

`setFillColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for fillColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

`setFillRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

`setForeColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for foreColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

`setForeRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

`setLMS`(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

`setLineRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setOpacity(*newOpacity*, *operation=''*, *log=None*)

Hard setter for opacity, allows the suppression of log messages and calls the update method

setOri(*newOri*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setPos(*newPos*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setSize(*newSize*, *operation=''*, *units=None*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setVertices(*value=None*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

property size

The size (width, height) of the stimulus in the stimulus *units*

Value should be *x,y-pair*, *scalar* (applies to both dimensions) or None (resets to default). *Operations* are supported.

Sizes can be negative (causing a mirror-image reversal) and can extend beyond the window.

Example:

```
stim.size = 0.8  # Set size to (xsize, ysize) = (0.8, 0.8)
print(stim.size)  # Outputs array([0.8, 0.8])
stim.size += (0.5, -0.5)  # make wider and flatter: (1.3, 0.3)
```

Tip: if you can see the actual pixel range this corresponds to by looking at *stim._sizeRendered*

updateColors()

Placeholder method to update colours when set externally, for example updating the *pallette* attribute of a textbox

updateOpacity()

Placeholder method to update colours when set externally, for example updating the *pallette* attribute of a textbox.

property vertices

A list of lists or a numpy array (Nx2) specifying xy positions of each vertex, relative to the center of the field.

Assigning to vertices can be slow if there are many vertices.

Operations supported with *.setVertices()*.

property verticesPix

This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

property win

The *Window* object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1  # stimulus will be drawn in win1
stim.draw()  # stimulus is now drawn to win1
stim.win = win2  # stimulus will be drawn in win2
stim.draw()  # it is now drawn in win2
win1.flip(waitBlanking=False)  # do not wait for next
             # monitor update
win2.flip()  # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

11.4.24 psychopy.visual.Rect

Stimulus class for drawing rectangles and squares.

Overview

Rect(win[, width, height, units, lineWidth, ...])	Creates a rectangle of given width and height as a special case of a `ShapeStim`.
`Rect.width`	
`Rect.height`	
`Rect.units`	
Rect.lineWidth	Width of the line in **pixels**.
Rect.lineColor	Alternative way of setting *borderColor*.
Rect.lineColorSpace	Deprecated, please use colorSpace to set color space for the entire object
Rect.fillColor	Set the fill color for the shape.
Rect.fillColorSpace	Deprecated, please use colorSpace to set color space for the entire object.
Rect.pos	The position of the center of the stimulus in the stimulus *units*
Rect.size	The size (width, height) of the stimulus in the stimulus *units*
Rect.ori	The orientation of the stimulus (in degrees).
Rect.opacity	Determines how visible the stimulus is relative to background.
Rect.contrast	A value that is simply multiplied by the color.
Rect.depth	DEPRECATED, depth is now controlled simply by drawing order.
Rect.interpolate	If *True* the edge of the line will be anti-aliased.
Rect.lineRGB	Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*
Rect.fillRGB	Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*
Rect.name	The name (*str*) of the object to be using during logged messages about this stim.
Rect.autoLog	Whether every change in this stimulus should be auto logged.
Rect.autoDraw	Determines whether the stimulus should be automatically drawn on every frame flip.
Rect.color	Set the color of the shape.
Rect.colorSpace	The name of the color space currently being used

Details

class psychopy.visual.rect.**Rect**(*win*, *width=0.5*, *height=0.5*, *units=''*, *lineWidth=1.5*, *lineColor=None*, *fillColor='white'*, *colorSpace='rgb'*, *pos=(0, 0)*, *size=None*, *anchor=None*, *ori=0.0*, *opacity=None*, *contrast=1.0*, *depth=0*, *interpolate=True*, *draggable=False*, *name=None*, *autoLog=None*, *autoDraw=False*, *color=undefined*, *lineColorSpace=undefined*, *fillColorSpace=undefined*, *lineRGB=undefined*, *fillRGB=undefined*)

Creates a rectangle of given width and height as a special case of a `ShapeStim`. This is a lazy-imported class, therefore import using full path *from psychopy.visual.rect import Rect* when inheriting from it.

Parameters

- **win** (*Window*) – Window this shape is being drawn to. The stimulus instance will allocate its required resources using that Windows context. In many cases, a stimulus instance cannot be drawn on different windows unless those windows share the same OpenGL context, which permits resources to be shared between them.
- **width** (*float or int*) – The width or height of the shape. *DEPRECATED* use *size* to define the dimensions of the shape on initialization. If *size* is specified the values of *width* and *height* are ignored. This is to provide legacy compatibility for existing applications.
- **height** (*float or int*) – The width or height of the shape. *DEPRECATED* use *size* to define the dimensions of the shape on initialization. If *size* is specified the values of *width* and *height* are ignored. This is to provide legacy compatibility for existing applications.
- **units** (*str*) – Units to use when drawing. This will affect how parameters and attributes *pos*, *size* and *radius* are interpreted.
- **lineWidth** (*float*) – Width of the shape's outline.
- **lineColor** (array_like, str, `Color` or None) – Color of the shape outline and fill. If *None*, a fully transparent color is used which makes the fill or outline invisible.
- **fillColor** (array_like, str, `Color` or None) – Color of the shape outline and fill. If *None*, a fully transparent color is used which makes the fill or outline invisible.
- **pos** (*array_like*) – Initial position (*x*, *y*) of the shape on-screen relative to the origin located at the center of the window or buffer in *units*. This can be updated after initialization by setting the *pos* property. The default value is *(0.0, 0.0)* which results in no translation.
- **size** (*array_like, float, int or None*) – Width and height of the shape as *(w, h)* or *[w, h]*. If a single value is provided, the width and height will be set to the same specified value. If *None* is specified, the *size* will be set with values passed to *width* and *height*.
- **ori** (*float*) – Initial orientation of the shape in degrees about its origin. Positive values will rotate the shape clockwise, while negative values will rotate counterclockwise. The default value for *ori* is 0.0 degrees.
- **opacity** (*float*) – Opacity of the shape. A value of 1.0 indicates fully opaque and 0.0 is fully transparent (therefore invisible). Values between 1.0 and 0.0 will result in colors being blended with objects in the background. This value affects the fill (*fillColor*) and outline (*lineColor*) colors of the shape.
- **contrast** (*float*) – Contrast level of the shape (0.0 to 1.0). This value is used to modulate the contrast of colors passed to *lineColor* and *fillColor*.
- **depth** (*int*) – Depth layer to draw the shape when *autoDraw* is enabled. *DEPRECATED*
- **interpolate** (*bool*) – Enable smoothing (anti-aliasing) when drawing shape outlines. This produces a smoother (less-pixelated) outline of the shape.
- **name** (*str*) – Optional name of the stimuli for logging.
- **autoLog** (*bool*) – Enable auto-logging of events associated with this stimuli. Useful for debugging and to track timing when used in conjunction with *autoDraw*.
- **autoDraw** (*bool*) – Enable auto drawing. When *True*, the stimulus will be drawn every frame without the need to explicitly call the `draw()` method.
- **color** (array_like, str, `Color` or None) – Sets both the initial *lineColor* and *fillColor* of the shape.
- **colorSpace** (*str*) – Sets the colorspace, changing how values passed to *lineColor* and *fillColor* are interpreted.

- **draggable** (*bool*) – Can this stimulus be dragged by a mouse click?

width, height

The width and height of the rectangle. Values are aliased with fields in the *size* attribute. Use these values to adjust the size of the rectangle in a single dimension after initialization.

Type
float or int

property RGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type
DEPRECATED

static _calcEquilateralVertices(*edges*, *radius=0.5*)

Get vertices for an equilateral shape with a given number of sides, will assume radius is 0.5 (relative) but can be manually specified

_calcPosRendered()

DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix.

_calcSizeRendered()

DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix

static _calculateMinEdges(*lineWidth*, *threshold=180*)

Calculate how many points are needed in an equilateral polygon for the gap between line rects to be < 1px and for corner angles to exceed a threshold.

In other words, how many edges does a polygon need to have to appear smooth?

lineWidth
[int, float, np.ndarray] Width of the line in pixels

threshold
[int] Maximum angle (degrees) for corners of the polygon, useful for drawing a circle. Supply 180 for no maximum angle.

_drawLegacyGL(*win*, *keepMatrix*)

Legacy draw the stimulus in its relevant window.

You must call this method after every MyWin.flip() if you want the stimulus to appear on that frame and then update the screen again.

_getDesiredRGB(*rgb*, *colorSpace*, *contrast*)

Convert color to RGB while adding contrast. Requires self.rgb, self.colorSpace and self.contrast

_getPolyAsRendered()

DEPRECATED. Return a list of vertices as rendered.

_selectWindow(*win*)

Switch drawing to the specified window. Calls the window's _setCurrent() method which handles the switch.

_set(*attrib*, *val*, *op=''*, *log=None*)

DEPRECATED since 1.80.04 + 1. Use setAttribute() and val2array() instead.

_updateList()

The user shouldn't need this method since it gets called after every call to .set() Chooses between using and not using shaders each call.

`_updateVertices()`

Sets Stim.verticesPix and ._borderPix from pos, size, ori, flipVert, flipHoriz

`alphaThreshold`

Threshold for alpha values.

If the alpha value of a pixel is below this threshold, the pixel will be rejected (not drawn). This can be useful for creating a mask from an image with an alpha channel. The default value is 0.0, which means that no thresholding will be applied.

`autoDraw`

Determines whether the stimulus should be automatically drawn on every frame flip.

Value should be: *True* or *False*. You do NOT need to set this on every frame flip!

`autoLog`

Whether every change in this stimulus should be auto logged.

Value should be: *True* or *False*. Set to *False* if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren't likely to be useful.

`property backColor`

Alternative way of setting fillColor

`property backColorSpace`

Deprecated, please use colorSpace to set color space for the entire object.

`property backRGB`

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type
DEPRECATED

`property backgroundColor`

Alternative way of setting fillColor

`property borderColorSpace`

Deprecated, please use colorSpace to set color space for the entire object

`property borderRGB`

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type
DEPRECATED

`closeShape`

Should the last vertex be automatically connected to the first?

If you're using *Polygon*, *Circle* or *Rect*, *closeShape=True* is assumed and shouldn't be changed.

`color`

Set the color of the shape. Sets both *fillColor* and *lineColor* simultaneously if applicable.

`property colorSpace`

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                       color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

`contains`(*x*, *y=None*, *units=None*)

Returns True if a point x,y is inside the stimulus' border.

Can accept variety of input options:

- two separate args, x and y
- one arg (list, tuple or array) containing two vals (x,y)
- **an object with a getPos() method that returns x,y, such**
  as a `Mouse`.

Returns *True* if the point is within the area defined either by its *border* attribute (if one defined), or its *vertices* attribute if there is no .border. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the *contains* method; the extent of the stimulus is determined purely by the size, position (pos), and orientation (ori) settings (and by the vertices for shape stimuli).

See Coder demos: shapeContains.py See Coder demos: shapeContains.py

`property contrast`

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged).
Operations supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast =  1.0  # unchanged contrast
stim.contrast =  0.5  # decrease contrast
stim.contrast =  0.0  # uniform, no contrast
stim.contrast = -0.5  # slightly inverted
stim.contrast = -1.0  # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast =  1.2  # increases contrast
stim.contrast = -1.2  # inverts with increased contrast
```

depth

DEPRECATED, depth is now controlled simply by drawing order.

doDragging()

If this stimulus is draggable, do the necessary actions on a frame flip to drag it.

draggable

Can this stimulus be dragged by a mouse click?

draw(*win=None*, *keepMatrix=False*)

Draw the stimulus in its relevant window.

You must call this method after every MyWin.flip() if you want the stimulus to appear on that frame and then update the screen again.

property fillColor

Set the fill color for the shape.

property fillColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property fillRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type
DEPRECATED

property flip

1x2 array for flipping vertices along each axis; set as True to flip or False to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (*._flip*) will give an array of 1s and -1s with which to multiply vertices.

property fontColor

Alternative way of setting *foreColor*.

property foreColor

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names <http://www.w3schools.com/html/html_colornames.asp>* can be used.
- *Colors by hex value*
- **numerically: (scalar or triplet) for DKL, RGB or**
  other *Color spaces*. For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue'  # (the case is actually ignored)
stim.color = '#DDA0DD'  # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0]  # if stim.colorSpace='rgb':
```

(continues on next page)

(continued from previous page)

```
                # a red color in rgb space
stim.color = [0.0, 45.0, 1.0]  # if stim.colorSpace='dkl':
                # DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255]  # if stim.colorSpace='rgb255':
                # a blue stimulus using rgb255 space
stim.color = 255  # interpreted as (255, 255, 255)
                  # which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1]  # increment all guns by 1 value
stim.color *= -1  # multiply the color by -1 (which in this
                    # space inverts the contrast)
stim.color *= [0.5, 0, 1]  # decrease red, remove green, keep blue
```

You can use *setColor* if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

property foreColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property foreRGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type
DEPRECATED

interpolate

If *True* the edge of the line will be anti-aliased.

property lineColor

Alternative way of setting *borderColor*.

property lineColorSpace

Deprecated, please use colorSpace to set color space for the entire object

property lineRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type
DEPRECATED

lineWidth

Width of the line in **pixels**.

Operations supported.

name

The name (*str*) of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = None your stimulus will be called "unnamed <type>", e.g. visual.TextStim(win) will be called "unnamed TextStim" in the logs.

property opacity

Determines how visible the stimulus is relative to background.

The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). *Operations* are supported. Precisely how this is used depends on the *Blend Mode*.

ori

The orientation of the stimulus (in degrees).

Should be a single value (*scalar*). *Operations* are supported.

Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and below zero values wrap appropriately.

overlaps(*polygon*)

Returns *True* if this stimulus intersects another one.

If *polygon* is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the *overlaps* method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

See coder demo, shapeContains.py

property pos

The position of the center of the stimulus in the stimulus *units*

value should be an *x,y-pair*. *Operations* are also supported.

Example:

```
stim.pos = (0.5, 0)  # Set slightly to the right of center
stim.pos += (0.5, -1)  # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2  # Move stim towards the center.
    Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:

```
from psychopy.tools.monitorunittools import posToPix
posPix = posToPix(stim)
```

setAlphaThreshold(*value*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setAutoDraw(*value*, *log=None*)

Sets autoDraw. Usually you can use 'stim.attribute = value' syntax instead, but use this method to suppress the log message.

setAutoLog(*value=True*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

`setBackRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

`setBorderColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

`setBorderRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

`setColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Sets both the line and fill to be the same color.

`setContrast`(*newContrast*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

`setDKL`(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

`setDepth`(*newDepth*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

`setFillColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for fillColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

`setFillRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

`setForeColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for foreColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

`setForeRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

`setHeight`(*height*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

`setLMS`(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

`setLineRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

`setOpacity`(*newOpacity*, *operation=''*, *log=None*)

Hard setter for opacity, allows the suppression of log messages and calls the update method

`setOri`(*newOri*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

`setPos`(*newPos*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setSize(*size*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

Operations supported.

setVertices(*value=None*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setWidth(*width*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

property size

The size (width, height) of the stimulus in the stimulus *units*

Value should be *x,y-pair*, *scalar* (applies to both dimensions) or None (resets to default). *Operations* are supported.

Sizes can be negative (causing a mirror-image reversal) and can extend beyond the window.

Example:

```
stim.size = 0.8  # Set size to (xsize, ysize) = (0.8, 0.8)
print(stim.size)  # Outputs array([0.8, 0.8])
stim.size += (0.5, -0.5)  # make wider and flatter: (1.3, 0.3)
```

Tip: if you can see the actual pixel range this corresponds to by looking at *stim._sizeRendered*

updateColors()

Placeholder method to update colours when set externally, for example updating the *pallette* attribute of a textbox

updateOpacity()

Placeholder method to update colours when set externally, for example updating the *pallette* attribute of a textbox.

property verticesPix

This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

property win

The `Window` object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1  # stimulus will be drawn in win1
stim.draw()  # stimulus is now drawn to win1
stim.win = win2  # stimulus will be drawn in win2
stim.draw()  # it is now drawn in win2
win1.flip(waitBlanking=False)  # do not wait for next
             # monitor update
win2.flip()  # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

11.4.25 `psychopy.visual.Rift`

Overview

Rift([fovType, trackingOriginType, ...])	Class provides a display and peripheral interface for the Oculus Rift (see: https://www.oculus.com/) head-mounted display.
Rift.close()	Close the window and cleanly shutdown the LibOVR session.
Rift.size	Size property to get the dimensions of the view buffer instead of the window.
Rift.setSize(value[, log])	
Rift.perfHudMode([mode])	Set the performance HUD mode.
Rift.hidePerfHud()	Hide the performance HUD.
Rift.stereoDebugHudMode(mode)	Set the debug stereo HUD mode.
Rift.setStereoDebugHudOption(option, value)	Configure stereo debug HUD guides.
Rift.userHeight	Get user height in meters (*float*).
Rift.eyeHeight	Eye height in meters (*float*).
Rift.eyeToNoseDistance	Eye to nose distance in meters (*float*).
Rift.eyeOffset	Eye separation in centimeters (*float*).
Rift.hasPositionTracking	`True` if the HMD is capable of tracking position.
Rift.hasOrientationTracking	`True` if the HMD is capable of tracking orientation.
Rift.hasMagYawCorrection	`True` if this HMD supports yaw drift correction.
Rift.manufacturer	Get the connected HMD's manufacturer (*str*).
Rift.serialNumber	Get the connected HMD's unique serial number (*str*).
Rift.hid	USB human interface device (HID) identifiers (*int*, *int*).
Rift.displayResolution	Get the HMD's raster display size (*int*, *int*).
Rift.displayRefreshRate	Get the HMD's display refresh rate in Hz (*float*).
Rift.pixelsPerTanAngleAtCenter	Horizontal and vertical pixels per tangent angle (=1) at the center of the display.
Rift.tanAngleToNDC(horzTan, vertTan)	Convert tan angles to the normalized device coordinates for the current buffer.
Rift.trackerCount	Number of attached trackers.
Rift.getTrackerInfo(trackerIdx)	Get tracker information.
Rift.headLocked	*True* if head locking is enabled.
Rift.trackingOriginType	Current tracking origin type (*str*).
Rift.recenterTrackingOrigin()	Recenter the tracking origin using the current head position.
Rift.specifyTrackingOrigin(pose)	Specify a tracking origin.
Rift.specifyTrackingOriginPosOri([pos, ori])	Specify a tracking origin using a pose and orientation.
Rift.clearShouldRecenterFlag()	Clear the 'shouldRecenter' status flag at the API level.
Rift.testBoundary(deviceType[, bounadryType])	Test if tracked devices are colliding with the play area boundary.
Rift.sensorSampleTime	Sensor sample time (*float*).
Rift.getDevicePose(deviceName[, absTime, ...])	Get the pose of a tracked device.

continues on next page

Table 11.1 – continued from previous page

Rift.getTrackingState([absTime, latencyMarker])	Get the tracking state of the head and hands.
Rift.calcEyePoses(headPose[, originPose])	Calculate eye poses for rendering.
Rift.eyeRenderPose	Computed eye pose for the current buffer.
Rift.shouldQuit	*True* if the user requested the application should quit through the headset's interface.
Rift.isVisible	*True* if the app has focus in the HMD and is visible to the viewer.
Rift.hmdMounted	*True* if the HMD is mounted on the user's head.
Rift.hmdPresent	*True* if the HMD is present.
Rift.shouldRecenter	*True* if the user requested the origin be re-centered through the headset's interface.
Rift.hasInputFocus	*True* if the application currently has input focus.
Rift.overlayPresent	
Rift.setBuffer(buffer[, clear])	Set the active draw buffer.
Rift.getPredictedDisplayTime()	Get the predicted time the next frame will be displayed on the HMD.
Rift.getTimeInSeconds()	Absolute time in seconds.
Rift.viewMatrix	The view matrix for the current eye buffer.
Rift.nearClip	Distance to the near clipping plane in meters.
Rift.farClip	Distance to the far clipping plane in meters.
Rift.projectionMatrix	Get the projection matrix for the current eye buffer.
Rift.isBoundaryVisible	True if the VR boundary is visible.
Rift.getBoundaryDimensions([boundaryType])	Get boundary dimensions.
Rift.connectedControllers	Connected controller types (*list* of *str*)
Rift.updateInputState([controllers])	Update all connected controller states.
Rift.flip([clearBuffer, drawMirror])	Submit view buffer images to the HMD's compositor for display at next V-SYNC and draw the mirror texture to the on-screen window.
Rift.multiplyViewMatrixGL()	Multiply the local eye pose transformation modelMatrix obtained from the SDK using `glMultMatrixf`.
Rift.multiplyProjectionMatrixGL()	Multiply the current projection modelMatrix obtained from the SDK using `glMultMatrixf`.
Rift.setRiftView([clearDepth])	Set head-mounted display view.
Rift.setDefaultView([clearDepth])	Return to default projection.
Rift.getThumbstickValues([controller, deadzone])	Get controller thumbstick values.
Rift.getIndexTriggerValues([controller, ...])	Get the values of the index triggers.
Rift.getHandTriggerValues([controller, deadzone])	Get the values of the hand triggers.
Rift.getButtons(buttons[, controller, testState])	Get button states from a controller.
Rift.getTouches(touches[, controller, testState])	Get touch states from a controller.
Rift.startHaptics(controller[, frequency, ...])	Start haptic feedback (vibration).
Rift.stopHaptics(controller)	Stop haptic feedback.
Rift.createHapticsBuffer(samples)	Create a new haptics buffer.
Rift.submitControllerVibration(controller, ...)	Submit a haptics buffer to begin controller vibration.
Rift.createPose([pos, ori])	Create a new Rift pose object (`LibOVRPose`).
Rift.createBoundingBox([extents])	Create a new bounding box object (`LibOVRBounds`).
Rift.isPoseVisible(pose)	Check if a pose object if visible to the present eye.

Details

class psychopy.visual.rift.**Rift**(*fovType='recommended'*, *trackingOriginType='floor'*, *texelsPerPixel=1.0*, *headLocked=False*, *highQuality=True*, *monoscopic=False*, *samples=1*, *mirrorMode='default'*, *mirrorRes=None*, *warnAppFrameDropped=True*, *autoUpdateInput=True*, *legacyOpenGL=True*, **args*, ***kwargs*)

Class provides a display and peripheral interface for the Oculus Rift (see: https://www.oculus.com/) head-mounted display. This is a lazy-imported class, therefore import using full path *from psychopy.visual.rift import Rift* when inheriting from it.

Requires PsychXR 0.2.4 to be installed. Setting the *winType='glfw'* is preferred for VR applications.

Parameters

- **fovType** (*str*) – Field-of-view (FOV) configuration type. Using 'recommended' auto-configures the FOV using the recommended parameters computed by the runtime. Using 'symmetric' forces a symmetric FOV using optimal parameters from the SDK, this mode is required for displaying 2D stimuli. Specifying 'max' will use the maximum FOVs supported by the HMD.
- **trackingOriginType** (*str*) – Specify the HMD origin type. If 'floor', the height of the user is added to the head tracker by LibOVR.
- **texelsPerPixel** (*float*) – Texture pixels per display pixel at FOV center. A value of 1.0 results in 1:1 mapping. A fractional value results in a lower resolution draw buffer which may increase performance.
- **headLocked** (*bool*) – Lock the compositor render layer in-place, disabling Asynchronous Space Warp (ASW). Enable this if you plan on computing eye poses using custom or modified head poses.
- **highQuality** (*bool*) – Configure the compositor to use anisotropic texture sampling (4x). This reduces aliasing artifacts resulting from high frequency details particularly in the periphery.
- **nearClip** (*float*) – Location of the near and far clipping plane in GL units (meters by default) from the viewer. These values can be updated after initialization.
- **farClip** (*float*) – Location of the near and far clipping plane in GL units (meters by default) from the viewer. These values can be updated after initialization.
- **monoscopic** (*bool*) – Enable monoscopic rendering mode which presents the same image to both eyes. Eye poses used will be both centered at the HMD origin. Monoscopic mode uses a separate rendering pipeline which reduces VRAM usage. When in monoscopic mode, you do not need to call 'setBuffer' prior to rendering (doing so will do have no effect).
- **samples** (*int or str*) – Specify the number of samples for multi-sample anti-aliasing (MSAA). When >1, multi-sampling logic is enabled in the rendering pipeline. If 'max' is specified, the largest number of samples supported by the platform is used. If floating point textures are used, MSAA sampling is disabled. Must be power of two value.
- **mirrorMode** (*str*) – On-screen mirror mode. Values 'left' and 'right' show rectilinear images of a single eye. Value 'distortion` shows the post-distortion image after being processed by the compositor. Value 'default' displays rectilinear images of both eyes side-by-side.
- **mirrorRes** (*list of int*) – Resolution of the mirror texture. If *None*, the resolution will match the window size. The value of *mirrorRes* is used for to define the resolution of movie frames.
- **warnAppFrameDropped** (*bool*) – Log a warning if the application drops a frame. This occurs when the application fails to submit a frame to the compositor on-time. Application

frame drops can have many causes, such as running routines in your application loop that take too long to complete. However, frame drops can happen sporadically due to driver bugs and running background processes (such as Windows Update). Use the performance HUD to help diagnose the causes of frame drops.

- **autoUpdateInput** (*bool*) – Automatically update controller input states at the start of each frame. If *False*, you must manually call *updateInputState* before getting input values from *LibOVR* managed input devices.
- **legacyOpenGL** (*bool*) – Disable 'immediate mode' OpenGL calls in the rendering pipeline. Specifying False maintains compatibility with existing PsychoPy stimuli drawing routines. Use True when computing transformations using some other method and supplying shaders matrices directly.

`USE_LEGACY_GL = True`

_assignFlipTime(*obj*, *attrib*, *format=<class 'float'>*)

Helper function to assign the time of last flip to the obj.attrib

Parameters

- **obj** (*dict or object*) – A mutable object (usually a dict of class instance).
- **attrib** (*str*) – Key or attribute of `obj` to assign the flip time to.
- **format** (*str, class or None*) – Format in which to return time, see clock.Timestamp.resolve() for more info. Defaults to *float*.

_checkMatchingSizes(*requested*, *actual*)

Checks whether the requested and actual screen sizes differ. If not then a warning is output and the window size is set to actual

_cleanEditables()

Make sure there are no dead refs in the editables list

_clearDepthBuffer()

Clear the depth buffer.

_endOfFlip(*clearBuffer*)

Override end of flip with custom color channel masking if required.

_getFrame(*rect=None*, *buffer='mirror'*)

Return the current HMD view as an image.

Parameters

- **rect** (*array_like*) – Rectangle [x, y, w, h] defining a sub-region of the frame to capture. This should remain *None* for HMD frames.
- **buffer** (*str, optional*) – Buffer to capture. For the HMD, only 'mirror' is available at this time.

Returns

Buffer pixel contents as a PIL/Pillow image object.

Return type

Image

_getPixels(*rect=None*, *buffer='front'*, *includeAlpha=True*, *makeLum=False*)

Return an array of pixel values from the current window buffer or sub-region.

Parameters

- **rect** (*tuple[int], optional*) – The region of the window to capture in pixel coordinates (left, bottom, width, height). If *None*, the whole window is captured.
- **buffer** (*str, optional*) – Buffer to capture.
- **includeAlpha** (*bool, optional*) – Include the alpha channel in the returned array. Default is *True*.
- **makeLum** (*bool, optional*) – Convert the RGB values to luminance values. Values are rounded to the nearest integer. Default is *False*.

Returns

Pixel values as a 3D array of shape (height, width, channels). If *includeAlpha* is *False*, the array will have shape (height, width, 3). If *makeLum* is *True*, the array will have shape (height, width).

Return type

ndarray

Examples

Get the pixel values of the whole window:

```
pix = win._getPixels()
```

Get pixel values and convert to luminance and get average:

```
pix = win._getPixels(makeLum=True)
average = pix.mean()
```

_getRegionOfFrame(*rect=(-1, 1, 1, -1)*, *buffer='front'*, *power2=False*, *squarePower2=False*)

Deprecated function, here for historical reasons. You may now use *:py:attr:`~Window._getFrame()* and specify a rect to get a sub-region, just as used here.

power2 can be useful with older OpenGL versions to avoid interpolation in `PatchStim`. If power2 or squarePower2, it will expand rect dimensions up to next power of two. squarePower2 uses the max dimensions. You need to check what your hardware & OpenGL supports, and call `_getRegionOfFrame()` as appropriate.

_prepareMonoFrame(*clear=True*)

Prepare a frame for monoscopic rendering. This is called automatically after `_startHmdFrame()` if monoscopic rendering is enabled.

_renderFBO()

Perform a warp operation.

(in this case a copy operation without any warping)

_resolveMSAA()

Resolve multisample anti-aliasing (MSAA). If MSAA is enabled, drawing operations are diverted to a special multisample render buffer. Pixel data must be ‘resolved’ by blitting it to the swap chain texture. If not, the texture will be blank.

Notes

You cannot perform operations on the default FBO (at *frameBuffer*) when MSAA is enabled. Any changes will be over-written when ‘flip’ is called.

_setCurrent()

Make this window's OpenGL context current.

If called on a window whose context is current, the function will return immediately. This reduces the number of redundant calls if no context switch is required. If `useFBO=True`, the framebuffer is bound after the context switch.

_setupFrameBuffer()

Override the default framebuffer init code in window.Window to use the HMD swap chain. The HMD's swap texture and render buffer are configured here.

If multisample anti-aliasing (MSAA) is enabled, a secondary render buffer is created. Rendering is diverted to the multi-sample buffer when drawing, which is then resolved into the HMD's swap chain texture prior to committing it to the chain. Consequently, you cannot pass the texture attached to the FBO specified by *frameBuffer* until the MSAA buffer is resolved. Doing so will result in a blank texture.

_setupGL()

Setup OpenGL state for this window.

_setupGamma(*gammaVal*)

A private method to work out how to handle gamma for this Window given that the user might have specified an explicit value, or maybe gave a Monitor.

_startHmdFrame()

Prepare to render an HMD frame. This must be called every frame before flipping or setting the view buffer.

This function will wait until the HMD is ready to begin rendering before continuing. The current frame texture from the swap chain are pulled from the SDK and made available for binding.

_startOfFlip()

Custom *`_startOfFlip`* for HMD rendering. This finalizes the HMD texture before diverting drawing operations back to the on-screen window. This allows *`flip`* to swap the on-screen and HMD buffers when called. This function always returns *True*.

Return type
True

_updatePerfStats()

Update and process performance statistics obtained from LibOVR. This should be called at the beginning of each frame to get the stats of the last frame.

This is called automatically when `_waitToBeginHmdFrame()` is called at the beginning of each frame.

_updateProjectionMatrix()

Update or re-calculate projection matrices based on the current render descriptor configuration.

_updateViewMatrix()

Update the default orthographic view matrix based on the current window settings.

_waitToBeginHmdFrame()

Wait until the HMD surfaces are available for rendering.

addEditable(*editable*)

Adds an editable element to the screen (something to which characters can be sent with meaning from the keyboard).

The current editable object receiving chars is Window.currentEditable

Parameters
editable

Returns

property ambientLight

Ambient light color for the scene [r, g, b, a]. Values range from 0.0 to 1.0. Only applicable if *useLights* is *True*.

Examples

Setting the ambient light color:

```
win.ambientLight = [0.5, 0.5, 0.5]

# don't do this!!!
win.ambientLight[0] = 0.5
win.ambientLight[1] = 0.5
win.ambientLight[2] = 0.5
```

applyEyeTransform(*clearDepth=True*)

Apply the current view and projection matrices.

Matrices specified by attributes `viewMatrix` and `projectionMatrix` are applied using 'immediate mode' OpenGL functions. Subsequent drawing operations will be affected until `flip()` is called.

All transformations in `GL_PROJECTION` and `GL_MODELVIEW` matrix stacks will be cleared (set to identity) prior to applying. After this is called, the current matrix mode will be set to `GL_MODELVIEW`.

Parameters

clearDepth (*bool*) – Clear the depth buffer. This may be required prior to rendering 3D objects.

Examples

Using a custom view and projection matrix:

```
# Must be called every frame since these values are reset after
# `flip()` is called!
win.viewMatrix = viewtools.lookAt( ... )
win.projectionMatrix = viewtools.perspectiveProjectionMatrix( ... )
win.applyEyeTransform()
# draw 3D objects here ...
```

property aspect

Aspect ratio of the current viewport (width / height).

backgroundFit

How should the background image of this window fit? Options are:

None, "None", "none"

No scaling is applied, image is present at its pixel size unaltered.

"cover"

Image is scaled such that it covers the whole screen without changing its aspect ratio. In other words, both dimensions are evenly scaled such that its SHORTEST dimension matches the window's LONGEST dimension.

"contain"

Image is scaled such that it is contained within the screen without changing its aspect ratio. In other words, both dimensions are evenly scaled such that its LONGEST dimension matches the window's SHORTEST dimension.

"scaleDown", "scale-down", "scaledown"

If image is bigger than the window along any dimension, it will behave as if backgroundFit were "contain". Otherwise, it will behave as if backgroundFit were None.

`backgroundImage`

Background image for the window, can be either a visual.ImageStim object or anything which could be passed to visual.ImageStim.image to create one. Will be drawn each time *win.flip()* is called, meaning it is always below all other contents of the window.

`blendMode`

Blend mode to use.

`calcEyePoses`(*headPose*, *originPose=None*)

Calculate eye poses for rendering.

This function calculates the eye poses to define the viewpoint transformation for each eye buffer. Upon starting a new frame, the application loop is halted until this function is called and returns.

Once this function returns, *setBuffer* may be called and frame rendering can commence. The computed eye pose for the selected buffer is accessible through the *`eyeRenderPose`* attribute after calling *`setBuffer()`*. If *monoscopic=True*, the eye poses are set to the head pose.

The source data specified to *headPose* can originate from the tracking state retrieved by calling *`getTrackingState()`*, or from other sources. If a custom head pose is specified (for instance, from a motion tracker), you must ensure *head-locking* is enabled to prevent the ASW feature of the compositor from engaging. Furthermore, you must specify sensor sample time for motion-to-photon calculation derived from the sample time of the custom tracking source.

Parameters

- **headPose** (*`LibOVRPose`*) – Head pose to use.
- **originPose** (*`LibOVRPose, optional`*) – Origin of tracking in the VR scene.

Examples

Get the tracking state and calculate the eye poses:

```python
# get tracking state at predicted mid-frame time
trackingState = hmd.getTrackingState()

# get the head pose from the tracking state
headPose = trackingState.headPose.thePose
hmd.calcEyePoses(headPose)  # compute eye poses

# begin rendering to each eye
for eye in ('left', 'right'):
    hmd.setBuffer(eye)
    hmd.setRiftView()
    # draw stuff here ...
```

Using a custom head pose (make sure `headLocked=True` before doing this):

```python
headPose = createPose((0., 1.75, 0.))
hmd.calcEyePoses(headPose)  # compute eye poses
```

`callOnFlip(function, *args, **kwargs)`

Call a function immediately after the next `flip()` command.

The first argument should be the function to call, the following args should be used exactly as you would for your normal call to the function (can use ordered arguments or keyword arguments as normal).

e.g. If you have a function that you would normally call like this:

```
pingMyDevice(portToPing, channel=2, level=0)
```

then you could call `callOnFlip()` to have the function call synchronized with the frame flip like this:

```
win.callOnFlip(pingMyDevice, portToPing, channel=2, level=0)
```

`clearAutoDraw()`

Remove all autoDraw components, meaning they get autoDraw set to False and are not added to any list (as in .stashAutoDraw)

`clearBuffer(color=True, depth=False, stencil=False)`

Clear the present buffer (to which you are currently drawing) without flipping the window.

Useful if you want to generate movie sequences from the back buffer without actually taking the time to flip the window.

Set *color* prior to clearing to set the color to clear the color buffer to. By default, the depth buffer is cleared to a value of 1.0.

Parameters

- **color** (*bool*) – Buffers to clear.
- **depth** (*bool*) – Buffers to clear.
- **stencil** (*bool*) – Buffers to clear.

Examples

Clear the color buffer to a specified color:

```
win.color = (1, 0, 0)
win.clearBuffer(color=True)
```

Clear only the depth buffer, *depthMask* must be *True* or else this will have no effect. Depth mask is usually *True* by default, but may change:

```
win.depthMask = True
win.clearBuffer(color=False, depth=True, stencil=False)
```

`clearShouldRecenterFlag()`

Clear the 'shouldRecenter' status flag at the API level.

`close()`

Close the window and cleanly shutdown the LibOVR session.

`property color`

Set the color of the window.

This command sets the color that the blank screen will have on the next clear operation. As a result it effectively takes TWO `flip()` operations to become visible (the first uses the color to create the new screen, the second presents that screen to the viewer). For this reason, if you want to changed background color of the window "on the fly", it might be a better idea to draw a `Rect` that fills the whole window with the desired `Rect.fillColor` attribute. That'll show up on first flip.

See other stimuli (e.g. `GratingStim.color`) for more info on the color attribute which essentially works the same on all PsychoPy stimuli.

See *Color spaces* for further information about the ways to specify colors and their various implications.

property colorSpace

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                       color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

property connectedControllers

Connected controller types (*list* of *str*)

property contentScaleFactor

Scaling factor (*float*) to use when drawing to the backbuffer to convert framebuffer to client coordinates.

See also

getContentScaleFactor

property convergeOffset

Convergence offset from monitor in centimeters.

This is value corresponds to the offset from screen plane to set the convergence plane (or point for *toe-in* projections). Positive offsets move the plane farther away from the viewer, while negative offsets nearer. This value is used by *setPerspectiveView* and should be set before calling it to take effect.

Notes

- This value is only applicable for *setToeIn* and *setOffAxisView*.

coordToRay(*screenXY*)

Convert a screen coordinate to a direction vector.

Takes a screen/window coordinate and computes a vector which projects a ray from the viewpoint through it (line-of-sight). Any 3D point touching the ray will appear at the screen coordinate.

Uses the current *viewport* and *projectionMatrix* to calculate the vector. The vector is in eye-space, where the origin of the scene is centered at the viewpoint and the forward direction aligned with the -Z axis. A ray of (0, 0, -1) results from a point at the very center of the screen assuming symmetric frustums.

Note that if you are using a flipped/mirrored view, you must invert your supplied screen coordinates (*screenXY*) prior to passing them to this function.

Parameters
`screenXY` (*array_like*) – X, Y screen coordinate. Must be in units of the window.

Returns
Normalized direction vector [x, y, z].

Return type
ndarray

Examples

Getting the direction vector between the mouse cursor and the eye:

```
mx, my = mouse.getPos()
dir = win.coordToRay((mx, my))
```

Set the position of a 3D stimulus object using the mouse, constrained to a plane. The object origin will always be at the screen coordinate of the mouse cursor:

```
# the eye position in the scene is defined by a rigid body pose
win.viewMatrix = camera.getViewMatrix()
win.applyEyeTransform()

# get the mouse location and calculate the intercept
mx, my = mouse.getPos()
ray = win.coordToRay([mx, my])
result = intersectRayPlane(   # from mathtools
    orig=camera.pos,
    dir=camera.transformNormal(ray),
    planeOrig=(0, 0, -10),
    planeNormal=(0, 1, 0))

# if result is `None`, there is no intercept
if result is not None:
    pos, dist = result
    objModel.thePose.pos = pos
else:
    objModel.thePose.pos = (0, 0, -10)  # plane origin
```

If you don't define the position of the viewer with a *RigidBodyPose*, you can obtain the appropriate eye position and rotate the ray by doing the following:

```
pos = numpy.linalg.inv(win.viewMatrix)[:3, 3]
ray = win.coordToRay([mx, my]).dot(win.viewMatrix[:3, :3])
# then ...
result = intersectRayPlane(
    orig=pos,
    dir=ray,
```

(continues on next page)

(continued from previous page)

```
    planeOrig=(0, 0, -10),
    planeNormal=(0, 1, 0))
```

static createBoundingBox(*extents=None*)

Create a new bounding box object (`LibOVRBounds`).

`LibOVRBounds` represents an axis-aligned bounding box with dimensions defined by *extents*. Bounding boxes are primarily used for visibility testing and culling by *PsychXR*. The dimensions of the bounding box can be specified explicitly, or fitted to meshes by passing vertices to the `fit()` method after initialization.

This function exposes the `LibOVRBounds` class so you don't need to access it by importing *psychxr*.

Parameters

extents (*array_like or None*) – Extents of the bounding box as (*mins*, *maxs*). Where *mins* (x, y, z) is the minimum and *maxs* (x, y, z) is the maximum extents of the bounding box in world units. If *None* is specified, the returned bounding box will be invalid. The bounding box can be later constructed using the `fit()` method or the `extents` attribute.

Returns

Object representing a bounding box.

Return type

~psychxr.libovr.LibOVRBounds

Examples

Add a bounding box to a pose:

```
# create a 1 meter cube bounding box centered with the pose
bbox = Rift.createBoundingBox(((-.5, -.5, -.5), (.5, .5, .5)))

# create a pose and attach the bounding box
modelPose = Rift.createPose()
modelPose.boundingBox = bbox
```

Perform visibility culling on the pose using the bounding box by using the `isVisible()` method:

```
if hmd.isPoseVisible(modelPose):
    modelPose.draw()
```

static createHapticsBuffer(*samples*)

Create a new haptics buffer.

A haptics buffer is object which stores vibration amplitude samples for playback through the Touch controllers. To play a haptics buffer, pass it to `submitHapticsBuffer()`.

Parameters

samples (*array_like*) – 1-D array of amplitude samples, ranging from 0 to 1. Values outside of this range will be clipped. The buffer must not exceed *HAPTICS_BUFFER_SAMPLES_MAX* samples, any additional samples will be dropped.

Returns

Haptics buffer object.

Return type

LibOVRHapticsBuffer

Notes

Methods *startHaptics* and *stopHaptics* cannot be used interchangeably with this function.

Examples

Create a haptics buffer where vibration amplitude ramps down over the course of playback:

```
samples = np.linspace(
    1.0, 0.0, num=HAPTICS_BUFFER_SAMPLES_MAX-1, dtype=np.float32)
hbuff = Rift.createHapticsBuffer(samples)

# vibrate right Touch controller
hmd.submitControllerVibration(CONTROLLER_TYPE_RTOUCH, hbuff)
```

`static` **`createPose`**(*pos=(0.0, 0.0, 0.0)*, *ori=(0.0, 0.0, 0.0, 1.0)*)

Create a new Rift pose object (`LibOVRPose`).

`LibOVRPose` is used to represent a rigid body pose mainly for use with the PsychXR's LibOVR module. There are several methods associated with the object to manipulate the pose.

This function exposes the `LibOVRPose` class so you don't need to access it by importing *psychxr*.

Parameters

- **pos** (*tuple, list, or ndarray of float*) – Position vector/coordinate (x, y, z).
- **ori** (*tuple, list, or ndarray of float*) – Orientation quaternion (x, y, z, w).

Returns

Object representing a rigid body pose for use with LibOVR.

Return type

`LibOVRPose`

`property` **`cullFace`**

True if face culling is enabled.`

`property` **`cullFaceMode`**

Face culling mode, either *back*, *front* or *both*.

`property` **`currentEditable`**

The editable (Text?) object that currently has key focus

`property` **`depthFunc`**

Depth test comparison function for rendering.

`property` **`depthMask`**

True if depth masking is enabled. Writing to the depth buffer will be disabled.

`property` **`depthTest`**

True if depth testing is enabled.

`classmethod` **`dispatchAllWindowEvents`**()

Dispatches events for all pyglet windows. Used by iohub 2.0 psychopy kb event integration.

`property` **`displayRefreshRate`**

Get the HMD's display refresh rate in Hz (*float*).

property displayResolution

Get the HMD's raster display size (*int*, *int*).

property draw3d

True if 3D drawing is enabled on this window.

property eyeHeight

Eye height in meters (*float*).

property eyeOffset

Eye separation in centimeters (*float*).

property eyeRenderPose

Computed eye pose for the current buffer. Only valid after calling *calcEyePoses()*.

property eyeToNoseDistance

Eye to nose distance in meters (*float*).

Examples

Generate your own eye poses. These are used when *calcEyePoses()* is called:

```
leftEyePose = Rift.createPose((-self.eyeToNoseDistance, 0., 0.))
rightEyePose = Rift.createPose((self.eyeToNoseDistance, 0., 0.))
```

Get the inter-axial separation (IAS) reported by *LibOVR*:

```
iad = self.eyeToNoseDistance * 2.0
```

property farClip

Distance to the far clipping plane in meters.

property firmwareVersion

Get the firmware version of the active HMD (*int*, *int*).

flip(*clearBuffer=True*, *drawMirror=True*)

Submit view buffer images to the HMD's compositor for display at next V-SYNC and draw the mirror texture to the on-screen window. This must be called every frame.

Parameters

- **clearBuffer** (*bool*) – Clear the frame after flipping.
- **drawMirror** (*bool*) – Draw the HMD mirror texture from the compositor to the window.

Returns

Absolute time in seconds when control was given back to the application. The difference between the current and previous values should be very close to 1 / refreshRate of the HMD.

Return type

float

Notes

- The HMD compositor and application are asynchronous, therefore there is no guarantee that the timestamp returned by 'flip' corresponds to the exact vertical retrace time of the HMD.

`fps()`

Report the frames per second since the last call to this function (or since the window was created if this is first call)

`property frameBufferSize`

Size of the framebuffer in pixels (w, h).

`property frontFace`

Face winding order to define front, either *ccw* or *cw*.

`property fullscr`

Return whether the window is in fullscreen mode.

`gamma`

Set the monitor gamma for linearization.

> **Warning**
>
> Don't use this if using a Bits++ or Bits#, as it overrides monitor settings.

`gammaRamp`

Sets the hardware CLUT using a specified 3xN array of floats ranging between 0.0 and 1.0.

Array must have a number of rows equal to 2 ^ max(bpc).

`getActualFrameRate`(*nIdentical=10*, *nMaxFrames=100*, *nWarmUpFrames=10*, *threshold=1*, *infoMsg=None*)

Measures the actual frames-per-second (FPS) for the screen.

This is done by waiting (for a max of *nMaxFrames*) until *nIdentical* frames in a row have identical frame times (std dev below *threshold* ms).

Parameters

- **`nIdentical`** (`int`, `optional`) – The number of consecutive frames that will be evaluated. Higher –> greater precision. Lower –> faster.
- **`nMaxFrames`** (`int`, `optional`) – The maximum number of frames to wait for a matching set of nIdentical.
- **`nWarmUpFrames`** (`int`, `optional`) – The number of frames to display before starting the test (this is in place to allow the system to settle after opening the *Window* for the first time.
- **`threshold`** (`int or float`, `optional`) – The threshold for the std deviation (in ms) before the set are considered a match.

Returns

Frame rate (FPS) in seconds. If there is no such sequence of identical frames a warning is logged and *None* will be returned.

Return type

float or None

`getBoundaryDimensions`(*boundaryType='PlayArea'*)

Get boundary dimensions.

Parameters

`boundaryType` (`str`) – Boundary type, can be 'PlayArea' or 'Outer'.

Returns
Dimensions of the boundary meters [x, y, z].

Return type
ndarray

getButtons(*buttons*, *controller='Xbox'*, *testState='continuous'*)

Get button states from a controller.

Returns *True* if any names specified to *buttons* reflect *testState* since the last `updateInputState` or *flip* call. If multiple button names are specified as a *list* or *tuple* to *buttons*, multiple button states are tested, returning *True* if all the buttons presently satisfy the *testState*. Note that not all controllers available share the same buttons. If a button is not available, this function will always return *False*.

Parameters

- **buttons** (*list* of *str* or *str*) – Buttons to test. Valid *buttons* names are 'A', 'B', 'RThumb', 'RShoulder' 'X', 'Y', 'LThumb', 'LShoulder', 'Up', 'Down', 'Left', 'Right', 'Enter', 'Back', 'VolUp', 'VolDown', and 'Home'. Names can be passed as a *list* to test multiple button states.
- **controller** (*str*) – Controller name.
- **testState** (*str*) – State to test. Valid values are:
  - **continuous** - Button is presently being held down.
  - **rising** or **pressed** - Button has been *pressed* since the last update.
  - **falling** or **released** - Button has been *released* since the last update.

Returns
Button state and timestamp in seconds the controller was polled.

Return type
tuple of bool, float

Examples

Check if the 'Enter' button on the Oculus remote was released:

```
isPressed, tsec = hmd.getButtons(['Enter'], 'Remote', 'falling')
```

Check if the 'A' button was pressed on the touch controller:

```
isPressed, tsec = hmd.getButtons(['A'], 'Touch', 'pressed')
```

getContentScaleFactor()

Get the scaling factor required for scaling correctly on high-DPI displays.

If the returned value is 1.0, no scaling needs to be applied to objects drawn on the backbuffer. A value >1.0 indicates that the backbuffer is larger than the reported client area, requiring points to be scaled to maintain constant size across similarly sized displays. In other words, the scaling required to convert framebuffer to client coordinates.

Returns
Scaling factor to be applied along both horizontal and vertical dimensions.

Return type
float

Examples

Get the size of the client area:

```
clientSize = win.frameBufferSize / win.getContentScaleFactor()
```

Get the framebuffer size from the client size:

```
frameBufferSize = win.clientSize * win.getContentScaleFactor()
```

Convert client (window) to framebuffer pixel coordinates (eg., a mouse coordinate, vertices, etc.):

```
# `mousePosXY` is an array ...
frameBufferXY = mousePosXY * win.getContentScaleFactor()
# you can also use the attribute ...
frameBufferXY = mousePosXY * win.contentScaleFactor
```

Notes

- This value is only valid after the window has been fully realized.

getDevicePose(*deviceName*, *absTime=None*, *latencyMarker=False*)

Get the pose of a tracked device. For head (HMD) and hand poses (Touch controllers) it is better to use *getTrackingState()* instead.

Parameters

- **deviceName** (*str*) – Name of the device. Valid device names are: 'HMD', 'LTouch', 'RTouch', 'Touch', 'Object0', 'Object1', 'Object2', and 'Object3'.
- **absTime** (*float, optional*) – Absolute time in seconds the device pose refers to. If not specified, the predicted time is used.
- **latencyMarker** (*bool*) – Insert a marker for motion-to-photon latency calculation. Should only be *True* if the HMD pose is being used to compute eye poses.

Returns

Pose state object. *None* if device tracking was lost.

Return type

LibOVRPoseState or *None*

getFutureFlipTime(*targetTime=0*, *clock=None*)

The expected time of the next screen refresh. This is currently calculated as win._lastFrameTime + refreshInterval

Parameters

- **targetTime** (*float*) – The delay *from now* for which you want the flip time. 0 will give the because that the earliest we can achieve. 0.15 will give the schedule flip time that gets as close to 150 ms as possible
- **clock** (*None, 'ptb', 'now' or any Clock object*) – If True then the time returned is compatible with ptb.GetSecs()
- **verbose** (*bool*) – Set to True to view the calculations along the way

getHandTriggerValues(*controller='Touch'*, *deadzone=False*)

Get the values of the hand triggers.

Parameters

- **controller** (*str*) – Name of the controller to get hand trigger values. Possible values for *controller* are 'Touch', 'RTouch', 'LTouch', 'Object0', 'Object1', 'Object2', and 'Object3'; the only devices with hand triggers the SDK manages. For additional controllers, use PsychPy's built-in event or hardware support.
- **deadzone** (*bool*) – Apply the deadzone to hand trigger values. This pre-filters stick input to apply a dead-zone where 0.0 will be returned if the trigger returns a displacement within 0.2746.

Returns

Left and right hand trigger values. Displacements are represented as *tuple* of two float representing the left anr right displacement values, which range from 0.0 to 1.0. The returned values reflect the controller state since the last `updateInputState` or `flip` call.

Return type

tuple

`getIndexTriggerValues`(*controller='Xbox'*, *deadzone=False*)

Get the values of the index triggers.

Parameters

- **controller** (*str*) – Name of the controller to get index trigger values. Possible values for *controller* are 'Xbox', 'Touch', 'RTouch', 'LTouch', 'Object0', 'Object1', 'Object2', and 'Object3'; the only devices with index triggers the SDK manages. For additional controllers, use PsychPy's built-in event or hardware support.
- **deadzone** (*bool*) – Apply the deadzone to index trigger values. This pre-filters stick input to apply a dead-zone where 0.0 will be returned if the trigger returns a displacement within 0.2746.

Returns

Left and right index trigger values. Displacements are represented as *tuple* of two float representing the left anr right displacement values, which range from 0.0 to 1.0. The returned values reflect the controller state since the last `updateInputState` or `flip` call.

Return type

tuple of float

`getMovieFrame`(*buffer='mirror'*)

Capture the current HMD frame as an image.

Saves to stack for `saveMovieFrames()`. As of v1.81.00 this also returns the frame as a PIL image.

This can be done at any time (usually after a `flip()` command).

Frames are stored in memory until a `saveMovieFrames()` command is issued. You can issue `getMovieFrame()` as often as you like and then save them all in one go when finished.

For HMD frames, you should call *getMovieFrame* after calling *flip* to ensure that the mirror texture saved reflects what is presently being shown on the HMD. Note, that this function is somewhat slow and may impact performance. Only call this function when you're not collecting experimental data.

Parameters

buffer (*str, optional*) – Buffer to capture. For the HMD, only 'mirror' is available at this time.

Returns

Buffer pixel contents as a PIL/Pillow image object.

Return type

Image

`getMsPerFrame`(*nFrames=60*, *showVisual=False*, *msg=''*, *msDelay=0.0*)

Assesses the monitor refresh rate (average, median, SD) under current conditions, over at least 60 frames.

Records time for each refresh (frame) for n frames (at least 60), while displaying an optional visual. The visual is just eye-candy to show that something is happening when assessing many frames. You can also give it text to display instead of a visual, e.g., `msg='(testing refresh rate...)'`; setting msg implies `showVisual == False`.

To simulate refresh rate under cpu load, you can specify a time to wait within the loop prior to doing the `flip()`. If 0 < msDelay < 100, wait for that long in ms.

Returns timing stats (in ms) of:

- average time per frame, for all frames
- standard deviation of all frames
- median, as the average of 12 frame times around the median (~monitor refresh rate)

Author

- 2010 written by Jeremy Gray

`getPredictedDisplayTime`()

Get the predicted time the next frame will be displayed on the HMD. The returned time is referenced to the clock *LibOVR* is using.

Returns

Absolute frame mid-point time for the given frame index in seconds.

Return type

float

`getThumbstickValues`(*controller='Xbox'*, *deadzone=False*)

Get controller thumbstick values.

Parameters

- **controller** (*str*) – Name of the controller to get thumbstick values. Possible values for *controller* are 'Xbox', 'Touch', 'RTouch', 'LTouch', 'Object0', 'Object1', 'Object2', and 'Object3'; the only devices with thumbsticks the SDK manages. For additional controllers, use PsychPy's built-in event or hardware support.
- **deadzone** (*bool*) – Apply the deadzone to thumbstick values. This pre-filters stick input to apply a dead-zone where 0.0 will be returned if the sticks return a displacement within -0.2746 to 0.2746.

Returns

Left and right, X and Y thumbstick values. Axis displacements are represented in each tuple by floats ranging from -1.0 (full left/down) to 1.0 (full right/up). The returned values reflect the controller state since the last *updateInputState* or *flip* call.

Return type

tuple

`getTimeInSeconds`()

Absolute time in seconds. The returned time is referenced to the clock *LibOVR* is using.

Returns

Time in seconds.

Return type
float

getTouches(*touches*, *controller='Touch'*, *testState='continuous'*)

Get touch states from a controller.

Returns *True* if any names specified to *touches* reflect *testState* since the last `updateInputState` or `flip` call. If multiple button names are specified as a *list* or *tuple* to *touches*, multiple button states are tested, returning *True* if all the touches presently satisfy the *testState*. Note that not all controllers available support touches. If a touch is not supported or available, this function will always return *False*.

Special states can be used for basic gesture recognition, such as 'LThumbUp', 'RThumbUp', 'LIndexPointing', and 'RIndexPointing'.

Parameters

- **touches** (*list* of *str* or *str*) – Buttons to test. Valid *touches* names are 'A', 'B', 'RThumb', 'RThumbRest' 'RThumbUp', 'RIndexPointing', 'LThumb', 'LThumbRest', 'LThumbUp', 'LIndexPointing', 'X', and 'Y'. Names can be passed as a *list* to test multiple button states.
- **controller** (*str*) – Controller name.
- **testState** (*str*) – State to test. Valid values are:
  - **continuous** - User is touching something on the controller.
  - **rising** or **pressed** - User began touching something since the last call to *updateInputState*.
  - **falling** or **released** - User stopped touching something since the last call to *updateInputState*.

Returns
Touch state and timestamp in seconds the controller was polled.

Return type
tuple of bool, float

Examples

Check if the 'Enter' button on the Oculus remote was released:

```
isPressed, tsec = hmd.getButtons(['Enter'], 'Remote', 'falling')
```

Check if the 'A' button was pressed on the touch controller:

```
isPressed, tsec = hmd.getButtons(['A'], 'Touch', 'pressed')
```

getTrackerInfo(*trackerIdx*)

Get tracker information.

Parameters
trackerIdx (*int*) – Tracker index, ranging from 0 to `trackerCount`.

Returns
Object containing tracker information.

Return type
`LibOVRTrackerInfo`

Raises
IndexError – Raised when *trackerIdx* out of range.

getTrackingState(*absTime=None*, *latencyMarker=True*)

Get the tracking state of the head and hands.

Calling this function retrieves the tracking state of the head (HMD) and hands at *absTime* from the *LibOVR* runtime. The returned object is a `LibOVRTrackingState` instance with poses, motion derivatives (i.e. linear and angular velocity/acceleration), and tracking status flags accessible through its attributes.

The pose states of the head and hands are available by accessing the *headPose* and *handPoses* attributes, respectively.

Parameters

- **absTime** (*float, optional*) – Absolute time the tracking state refers to. If not specified, the predicted display time is used.
- **latencyMarker** (*bool, optional*) – Set a latency marker upon getting the tracking state. This is used for motion-to-photon calculations.

Returns

Tracking state object. For more information about this type see:

Return type

`LibOVRTrackingState`

See also

getPredictedDisplayTime
Time at mid-frame for the current frame index.

Examples

Get the tracked head pose and use it to calculate render eye poses:

```
# get tracking state at predicted mid-frame time
absTime = getPredictedDisplayTime()
trackingState = hmd.getTrackingState(absTime)

# get the head pose from the tracking state
headPose = trackingState.headPose.thePose
hmd.calcEyePoses(headPose)  # compute eye poses
```

Get linear/angular velocity and acceleration vectors of the right touch controller:

```
# right hand is the second value (index 1) at `handPoses`
rightHandState = trackingState.handPoses[1]  # is `LibOVRPoseState`

# access `LibOVRPoseState` fields to get the data
linearVel = rightHandState.linearVelocity  # m/s
angularVel = rightHandState.angularVelocity  # rad/s
linearAcc = rightHandState.linearAcceleration  # m/s^2
angularAcc = rightHandState.angularAcceleration  # rad/s^2

# extract components like this if desired
vx, vy, vz = linearVel
ax, ay, az = angularVel
```

Above is useful for physics simulations, where one can compute the magnitude and direction of a force applied to a virtual object.

It's often the case that object tracking becomes unreliable for some reason, for instance, if it becomes occluded and is no longer visible to the sensors. In such cases, the reported pose state is invalid and may not be useful. You can check if the position and orientation of a tracked object is invalid using flags associated with the tracking state. This shows how to check if head position and orientation tracking was valid when sampled:

```
if trackingState.positionValid and trackingState.orientationValid:
    print('Tracking valid.')
```

It's up to the programmer to determine what to do in such cases. Note that tracking may still be valid even if

Get the calibrated origin used for tracking during the sample period of the tracking state:

```
calibratedOrigin = trackingState.calibratedOrigin
calibPos, calibOri = calibratedOrigin.posOri
```

Time integrate a tracking state. This extrapolates the pose over time given the present computed motion derivatives. The contrived example below shows how to implement head pose forward prediction:

```
# get current system time
absTime = getTimeInSeconds()

# get the elapsed time from `absTime` to predicted v-sync time,
# again this is an example, you would usually pass predicted time to
# `getTrackingState` directly.
dt = getPredictedDisplayTime() - absTime

# get the tracking state for the current time, poses will lag where
# they are expected at predicted time by `dt` seconds
trackingState = hmd.getTrackingState(absTime)

# time integrate a pose by `dt`
headPoseState = trackingState.headPose
headPosePredicted = headPoseState.timeIntegrate(dt)

# calc eye poses with predicted head pose, this is a custom pose to
# head-locking should be enabled!
hmd.calcEyePoses(headPosePredicted)
```

The resulting head pose is usually very close to what *getTrackingState* would return if the predicted time was used. Simple forward prediction with time integration becomes increasingly unstable as the prediction interval increases. Under normal circumstances, let the runtime handle forward prediction by using the pose states returned at the predicted display time. If you plan on doing your own forward prediction, you need enable head-locking, clamp the prediction interval, and apply some sort of smoothing to keep the image as stable as possible.

property hasInputFocus

True if the application currently has input focus.

property hasMagYawCorrection

`True` if this HMD supports yaw drift correction.

property hasOrientationTracking

`True` if the HMD is capable of tracking orientation.

property hasPositionTracking

`True` if the HMD is capable of tracking position.

property headLocked

True if head locking is enabled.

property hid

USB human interface device (HID) identifiers (*int*, *int*).

hideMessage()

Remove any message that is currently being displayed.

hidePerfHud()

Hide the performance HUD.

hidePilotingIndicator()

Hide the visual indicator which shows we are in piloting mode.

property hmdMounted

True if the HMD is mounted on the user's head.

property hmdPresent

True if the HMD is present.

property isBoundaryVisible

True if the VR boundary is visible.

isPoseVisible(*pose*)

Check if a pose object if visible to the present eye. This method can be used to perform visibility culling to avoid executing draw commands for objects that fall outside the FOV for the current eye buffer.

If `boundingBox` has a valid bounding box object, this function will return *False* if all the box points fall completely to one side of the view frustum. If `boundingBox` is *None*, the point at `pos` is checked, returning *False* if it falls outside of the frustum. If the present buffer is not 'left' or 'right', this function will always return *False*.

Parameters

pose (`LibOVRPose`) – Pose to test for visibility.

Returns

True if pose's bounding box or origin is outside of the view frustum.

Return type

bool

property isVisible

True if the app has focus in the HMD and is visible to the viewer.

property lights

Scene lights.

This is specified as an array of *~psychopy.visual.LightSource* objects. If a single value is given, it will be converted to a *list* before setting. Set *useLights* to *True* before rendering to enable lighting/shading on subsequent objects. If *lights* is *None* or an empty *list*, no lights will be enabled if *useLights=True*, however, the scene ambient light set with *ambientLight* will be still be used.

Examples

Create a directional light source and add it to scene lights:

```
dirLight = gltools.LightSource((0., 1., 0.), lightType='directional')
win.lights = dirLight  # `win.lights` will be a list when accessed!
```

Multiple lights can be specified by passing values as a list:

```
myLights = [gltools.LightSource((0., 5., 0.)),
            gltools.LightSource((-2., -2., 0.))
win.lights = myLights
```

logOnFlip(*msg*, *level*, *obj=None*)

Send a log message that should be time-stamped at the next `flip()` command.

Parameters

- **msg** (*str*) – The message to be logged.
- **level** (*int*) – The level of importance for the message.
- **obj** (*object, optional*) – The python object that might be associated with this message if desired.

property manufacturer

Get the connected HMD's manufacturer (*str*).

property mouseVisible

Returns the visibility of the mouse cursor.

multiFlip(*flips=1*, *clearBuffer=True*)

Flip multiple times while maintaining the display constant. Use this method for precise timing.

WARNING: This function should not be used. See the *Notes* section for details.

Parameters

- **flips** (*int, optional*) – The number of monitor frames to flip. Floats will be rounded to integers, and a warning will be emitted. `Window.multiFlip(flips=1)` is equivalent to `Window.flip()`. Defaults to *1*.
- **clearBuffer** (*bool, optional*) – Whether to clear the screen after the last flip. Defaults to *True*.

Notes

- This function can behave unpredictably, and the PsychoPy authors recommend against using it. See https://github.com/psychopy/psychopy/issues/867 for more information.

Examples

Example of using `multiFlip`:

```
# Draws myStim1 to buffer
myStim1.draw()
# Show stimulus for 4 frames (90 ms at 60Hz)
myWin.multiFlip(clearBuffer=False, flips=6)
# Draw myStim2 "on top of" myStim1
```

(continues on next page)

(continued from previous page)

```
# (because buffer was not cleared above)
myStim2.draw()
# Show this for 2 frames (30 ms at 60Hz)
myWin.multiFlip(flips=2)
# Show blank screen for 3 frames (buffer was cleared above)
myWin.multiFlip(flips=3)
```

multiplyProjectionMatrixGL()

Multiply the current projection modelMatrix obtained from the SDK using `glMultMatrixf`. The projection matrix used depends on the current eye buffer set by *setBuffer()*.

multiplyViewMatrixGL()

Multiply the local eye pose transformation modelMatrix obtained from the SDK using `glMultMatrixf`. The modelMatrix used depends on the current eye buffer set by *setBuffer()*.

Return type
None

property **nearClip**

Distance to the near clipping plane in meters.

nextEditable()

Moves focus of the cursor to the next editable window

onResize(*width*, *height*)

A default resize event handler.

This default handler updates the GL viewport to cover the entire window and sets the `GL_PROJECTION` matrix to be orthogonal in window space. The bottom-left corner is (0, 0) and the top-right corner is the width and height of the *Window* in pixels.

Override this event handler with your own to create another projection, for example in perspective.

property **overlayPresent**

perfHudMode(*mode='Off'*)

Set the performance HUD mode.

Parameters
mode (*str*) – HUD mode to use.

property **pixelsPerTanAngleAtCenter**

Horizontal and vertical pixels per tangent angle (=1) at the center of the display.

This can be used to compute pixels-per-degree for the display.

property **productName**

Get the HMD's product name (*str*).

property **projectionMatrix**

Get the projection matrix for the current eye buffer. Note that setting *projectionMatrix* manually will break visibility culling.

recenterTrackingOrigin()

Recenter the tracking origin using the current head position.

recordFrameIntervals

Record time elapsed per frame.

Provides accurate measures of frame intervals to determine whether frames are being dropped. The intervals are the times between calls to `flip()`. Set to *True* only during the time-critical parts of the script. Set this to *False* while the screen is not being updated, i.e., during any slow, non-frame-time-critical sections of your code, including inter-trial-intervals, `event.waitkeys()`, `core.wait()`, or `image.setImage()`.

Examples

Enable frame interval recording, successive frame intervals will be stored:

```
win.recordFrameIntervals = True
```

Frame intervals can be saved by calling the `saveFrameIntervals` method:

```
win.saveFrameIntervals()
```

removeEditable(*editable*)

resetEyeTransform(*clearDepth=True*)

Restore the default projection and view settings to PsychoPy defaults. Call this prior to drawing 2D stimuli objects (i.e. GratingStim, ImageStim, Rect, etc.) if any eye transformations were applied for the stimuli to be drawn correctly.

Parameters

clearDepth (*bool*) – Clear the depth buffer upon reset. This ensures successive draw commands are not affected by previous data written to the depth buffer. Default is *True*.

Notes

- Calling `flip()` automatically resets the view and projection to defaults. So you don't need to call this unless you are mixing 3D and 2D stimuli.

Examples

Going between 3D and 2D stimuli:

```
# 2D stimuli can be drawn before setting a perspective projection
win.setPerspectiveView()
# draw 3D stimuli here ...
win.resetEyeTransform()
# 2D stimuli can be drawn here again ...
win.flip()
```

resetViewport()

Reset the viewport to cover the whole framebuffer.

Set the viewport to match the dimensions of the back buffer or framebuffer (if *useFBO=True*). The scissor rectangle is also set to match the dimensions of the viewport.

retrieveAutoDraw()

Add all stimuli which are on 'hold' back into the autoDraw list, and clear the hold list.

property rgb

`saveFrameIntervals`(*fileName=None*, *clear=True*)

Save recorded screen frame intervals to disk, as comma-separated values.

Parameters

- **fileName** (*None* or str) – *None* or the filename (including path if necessary) in which to store the data. If None then 'lastFrameIntervals.log' will be used.
- **clear** (*bool*) – Clear buffer frames intervals were stored after saving. Default is *True*.

`saveMovieFrames`(*fileName*, *codec='libx264'*, *fps=30*, *clearFrames=True*)

Writes any captured frames to disk.

Will write any format that is understood by PIL (tif, jpg, png, . . .)

Parameters

- **filename** (*str*) – Name of file, including path. The extension at the end of the file determines the type of file(s) created. If an image type (e.g. .png) is given, then multiple static frames are created. If it is .gif then an animated GIF image is created (although you will get higher quality GIF by saving PNG files and then combining them in dedicated image manipulation software, such as GIMP). On Windows and Linux *.mpeg* files can be created if *pymedia* is installed. On macOS *.mov* files can be created if the pyobjc-frameworks-QTKit is installed. Unfortunately the libs used for movie generation can be flaky and poor quality. As for animated GIFs, better results can be achieved by saving as individual .png frames and then combining them into a movie using software like ffmpeg.
- **codec** (*str, optional*) – The codec to be used **by moviepy** for mp4/mpg/mov files. If *None* then the default will depend on file extension. Can be one of `libx264`, `mpeg4` for mp4/mov files. Can be `rawvideo`, `png` for avi files (not recommended). Can be `libvorbis` for ogv files. Default is `libx264`.
- **fps** (*int, optional*) – The frame rate to be used throughout the movie. **Only for quicktime (.mov) movies.**. Default is *30*.
- **clearFrames** (*bool, optional*) – Set this to *False* if you want the frames to be kept for additional calls to `saveMovieFrames`. Default is *True*.

Examples

Writes a series of static frames as frame001.tif, frame002.tif etc.:

```
myWin.saveMovieFrames('frame.tif')
```

As of PsychoPy 1.84.1 the following are written with moviepy:

```
myWin.saveMovieFrames('stimuli.mp4') # codec = 'libx264' or 'mpeg4'
myWin.saveMovieFrames('stimuli.mov')
myWin.saveMovieFrames('stimuli.gif')
```

`property scissor`

Scissor rectangle (x, y, w, h) for the current draw buffer.

Values *x* and *y* define the origin, and *w* and *h* the size of the rectangle in pixels. The scissor operation is only active if *scissorTest=True*.

Usually, the scissor and viewport are set to the same rectangle to prevent drawing operations from *spilling* into other regions of the screen. For instance, calling *clearBuffer* will only clear within the scissor rectangle.

Setting the scissor rectangle but not the viewport will restrict drawing within the defined region (like a rectangular aperture), not changing the positions of stimuli.

property scissorTest

True if scissor testing is enabled.

property screenshot

property sensorSampleTime

Sensor sample time (*float*). This value corresponds to the time the head (HMD) position was sampled, which is required for computing motion-to-photon latency. This does not need to be specified if *getTrackingState* was called with *latencyMarker=True*.

property serialNumber

Get the connected HMD's unique serial number (*str*).

Use this to identify a particular unit if you own many.

setBlendMode(*blendMode*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setBuffer(*buffer*, *clear=True*)

Set the active draw buffer.

> ⚠ **Warning**
>
> The window.Window.size property will return the buffer's dimensions in pixels instead of the window's when *setBuffer* is set to 'left' or 'right'.

Parameters

- **buffer** (`str`) – View buffer to divert successive drawing operations to, can be either 'left' or 'right'.
- **clear** (`boolean`) – Clear the color, stencil and depth buffer.

setColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Usually you can use `stim.attribute = value` syntax instead, but use this method if you want to set color and colorSpace simultaneously.

See `color` for documentation on colors.

setDefaultView(*clearDepth=True*)

Return to default projection. Call this before drawing PsychoPy's 2D stimuli after a stereo projection change.

Note: This only has an effect if using Rift in legacy immediate mode OpenGL.

Parameters
clearDepth (`bool`) – Clear the depth buffer prior after configuring the view parameters.

setGamma(*gamma*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setMouseType(*name='arrow'*)

Change the appearance of the cursor for this window. Cursor types provide contextual hints about how to interact with on-screen objects.

The graphics used 'standard cursors' provided by the operating system. They may vary in appearance and hot spot location across platforms. The following names are valid on most platforms:

- `arrow` : Default pointer.
- `ibeam` : Indicates text can be edited.
- `crosshair` : Crosshair with hot-spot at center.
- `hand` : A pointing hand.
- `hresize` : Double arrows pointing horizontally.
- `vresize` : Double arrows pointing vertically.

Parameters

name (*str*) – Type of standard cursor to use (see above). Default is `arrow`.

Notes

- On Windows the `crosshair` option is negated with the background color. It will not be visible when placed over 50% grey fields.

`setMouseVisible`(*visibility*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

`setOffAxisView`(*applyTransform=True*, *clearDepth=True*)

Set an off-axis projection.

Create an off-axis projection for subsequent rendering calls. Sets the *viewMatrix* and *projectionMatrix* accordingly so the scene origin is on the screen plane. If *eyeOffset* is correct and the view distance and screen size is defined in the monitor configuration, the resulting view will approximate *ortho-stereo* viewing.

The convergence plane can be adjusted by setting *convergeOffset*. By default, the convergence plane is set to the screen plane. Any points on the screen plane will have zero disparity.

Parameters

- **applyTransform** (*bool*) – Apply transformations after computing them in immediate mode. Same as calling `applyEyeTransform()` afterwards.
- **clearDepth** (*bool, optional*) – Clear the depth buffer.

`setOrthographicView`(*applyTransform=True*, *clearDepth=True*)

Set the projection and view matrix to render with orthographic view.

Orthographic projection is used to render 3D objects without perspective distortion. The scene origin is centered on the screen plane. The frustum is defined by the size of the window in pixels, with the origin at the center of the window. 2D stimuli are typically drawn using this projection.

Note that the values of `projectionMatrix` and `viewMatrix` will be replaced when calling this function.

Parameters

- **applyTransform** (*bool*) – Apply transformations after computing them in immediate mode. Same as calling `applyEyeTransform()` afterwards if *False*.
- **clearDepth** (*bool, optional*) – Clear the depth buffer.

`setPerspectiveView`(*applyTransform=True*, *clearDepth=True*)

Set the projection and view matrix to render with perspective.

Matrices are computed using values specified in the monitor configuration with the scene origin on the screen plane. Calculations assume units are in meters. If *eyeOffset != 0*, the view will be transformed laterally, however the frustum shape will remain the same.

Note that the values of `projectionMatrix` and `viewMatrix` will be replaced when calling this function.

Parameters

- **applyTransform** (`bool`) – Apply transformations after computing them in immediate mode. Same as calling `applyEyeTransform()` afterwards if *False*.
- **clearDepth** (`bool, optional`) – Clear the depth buffer.

`setRGB`(*newRGB*)

Deprecated: As of v1.61.00 please use *setColor()* instead

`setRecordFrameIntervals`(*value=True*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

`setRiftView`(*clearDepth=True*)

Set head-mounted display view. Gets the projection and view matrices from the HMD and applies them.

Note: This only has an effect if using Rift in legacy immediate mode OpenGL.

Parameters

clearDepth (`bool`) – Clear the depth buffer prior after configuring the view parameters.

`setScale`(*units*, *font='dummyFont'*, *prevScale=(1.0, 1.0)*)

DEPRECATED: this method used to be used to switch between units for stimulus drawing but this is now handled by the stimuli themselves and the window should always be left in units of 'pix'

`setSize`(*value*, *log=True*)

`setStereoDebugHudOption`(*option*, *value*)

Configure stereo debug HUD guides.

Parameters

- **option** (`str`) – Option to set. Valid options are *InfoEnable*, *Size*, *Position*, *YawPitchRoll*, and *Color*.
- **value** (`array_like or bool`) – Value to set for a given *option*. Appropriate types for each option are:
  - *InfoEnable* - bool, *True* to show, *False* to hide.
  - *Size* - array_like, [w, h] in meters.
  - *Position* - array_like, [x, y, z] in meters.
  - *YawPitchRoll* - array_like, [pitch, yaw, roll] in degrees.
  - *Color* - array_like, [r, g, b] as floats ranging 0.0 to 1.0.

Returns

`True` if the option was successfully set.

Return type

bool

Examples

Configuring a stereo debug HUD guide:

```
# show a quad with a crosshair
hmd.stereoDebugHudMode('QuadWithCrosshair')
# enable displaying guide information
hmd.setStereoDebugHudOption('InfoEnable', True)
# set the position of the guide quad in the scene
hmd.setStereoDebugHudOption('Position', [0.0, 1.7, -2.0])
```

`setToeInView`(*applyTransform=True*, *clearDepth=True*)

Set toe-in projection.

Create a toe-in projection for subsequent rendering calls. Sets the *viewMatrix* and *projectionMatrix* accordingly so the scene origin is on the screen plane. The value of *convergeOffset* will define the convergence point of the view, which is offset perpendicular to the center of the screen plane. Points falling on a vertical line at the convergence point will have zero disparity.

Parameters

- **applyTransform** (*bool*) – Apply transformations after computing them in immediate mode. Same as calling `applyEyeTransform()` afterwards.
- **clearDepth** (*bool, optional*) – Clear the depth buffer.

Notes

- This projection mode is only 'correct' if the viewer's eyes are converged at the convergence point. Due to perspective, this projection introduces vertical disparities which increase in magnitude with eccentricity. Use *setOffAxisView* if you want to display something the viewer can look around the screen comfortably.

`setUnits`(*value*, *log=True*)

`setViewOri`(*value*, *log=True*)

`setViewPos`(*value*, *log=True*)

`setViewScale`(*value*, *log=True*)

`property shouldQuit`

True if the user requested the application should quit through the headset's interface.

`property shouldRecenter`

True if the user requested the origin be re-centered through the headset's interface.

`showMessage`(*msg*)

Show a message in the window. This can be used to show information to the participant.

This creates a TextBox2 object that is displayed in the window. The text can be updated by calling this method again with a new message. The updated text will appear the next time *draw()* is called.

Parameters

msg (*str or None*) – Message text to display. If None, then any existing message is removed.

`showPilotingIndicator`()

Show the visual indicator which shows we are in piloting mode.

property size

Size property to get the dimensions of the view buffer instead of the window. If there are no view buffers, always return the dims of the window.

specifyTrackingOrigin(*pose*)

Specify a tracking origin. If *trackingOriginType='floor'*, this function sets the origin of the scene in the ground plane. If *trackingOriginType='eye'*, the scene origin is set to the known eye height.

Parameters

pose (*LibOVRPose*) – Tracking origin pose.

specifyTrackingOriginPosOri(*pos=(0.0, 0.0, 0.0)*, *ori=(0.0, 0.0, 0.0, 1.0)*)

Specify a tracking origin using a pose and orientation. This is the same as *specifyTrackingOrigin*, but accepts a position vector [x, y, z] and orientation quaternion [x, y, z, w].

Parameters

- **pos** (*tuple or list of float, or ndarray*) – Position coordinate of origin (x, y, z).
- **ori** (*tuple or list of float, or ndarray*) – Quaternion specifying orientation (x, y, z, w).

startHaptics(*controller*, *frequency='low'*, *amplitude=1.0*)

Start haptic feedback (vibration).

Vibration is constant at fixed frequency and amplitude. Vibration lasts 2.5 seconds, so this function needs to be called more often than that for sustained vibration. Only controllers which support vibration can be used here.

There are only two frequencies permitted 'high' and 'low', however, amplitude can vary from 0.0 to 1.0. Specifying `frequency`='off' stops vibration if in progress.

Parameters

- **controller** (*str*) – Name of the controller to vibrate.
- **frequency** (*str*) – Vibration frequency. Valid values are: 'off', 'low', or 'high'.
- **amplitude** (*float*) – Vibration amplitude in the range of [0.0 and 1.0]. Values outside this range are clamped.

stashAutoDraw()

Put autoDraw components on 'hold', meaning they get autoDraw set to False but are added to an internal list to be 'released' when .releaseAutoDraw is called.

property stencilTest

True if stencil testing is enabled.

stereoDebugHudMode(*mode*)

Set the debug stereo HUD mode.

This makes the compositor add stereoscopic reference guides to the scene. You can configure the HUD can be configured using other methods.

Parameters

mode (*str*) – Stereo debug mode to use. Valid options are *Off*, *Quad*, *QuadWithCrosshair*, and *CrosshairAtInfinity*.

Examples

Enable a stereo debugging guide:

```
hmd.stereoDebugHudMode('CrosshairAtInfinity')
```

Hide the debugging guide. Should be called before exiting the application since it's persistent until the Oculus service is restarted:

```
hmd.stereoDebugHudMode('Off')
```

`stopHaptics`(*controller*)

Stop haptic feedback.

Convenience function to stop controller vibration initiated by the last `vibrateController` call. This is the same as calling `vibrateController(controller, frequency='off')`.

Parameters
`controller` (*`str`*) – Name of the controller to stop vibrating.

`submitControllerVibration`(*controller*, *hapticsBuffer*)

Submit a haptics buffer to begin controller vibration.

Parameters

- **`controller`** (*`str`*) – Name of controller to vibrate.
- **`hapticsBuffer`** (*`LibOVRHapticsBuffer`*) – Haptics buffer to playback.

Notes

Methods *startHaptics* and *stopHaptics* cannot be used interchangeably with this function.

`tanAngleToNDC`(*horzTan*, *vertTan*)

Convert tan angles to the normalized device coordinates for the current buffer.

Parameters

- **`horzTan`** (*`float`*) – Horizontal tan angle.
- **`vertTan`** (*`float`*) – Vertical tan angle.

Returns
Normalized device coordinates X, Y. Coordinates range between -1.0 and 1.0. Returns *None* if an invalid buffer is selected.

Return type
tuple of float

`testBoundary`(*deviceType*, *bounadryType='PlayArea'*)

Test if tracked devices are colliding with the play area boundary.

This returns an object containing test result data.

Parameters

- **`deviceType`** (*`str, list or tuple`*) – The device to check for boundary collision. If a list of names is provided, they will be combined and all tested.
- **`boundaryType`** (*`str`*) – Boundary type to test.

timeOnFlip(*obj*, *attrib*, *format=<class 'float'>*)

Retrieves the time on the next flip and assigns it to the *attrib* for this *obj*.

Parameters

- **obj** (*dict or object*) – A mutable object (usually a dict of class instance).
- **attrib** (*str*) – Key or attribute of *obj* to assign the flip time to.
- **format** (*str, class or None*) – Format in which to return time, see clock.Timestamp.resolve() for more info. Defaults to *float*.

Examples

Assign time on flip to the `tStartRefresh` key of `myTimingDict`:

```
win.getTimeOnFlip(myTimingDict, 'tStartRefresh')
```

title

property trackerCount

Number of attached trackers.

property trackingOriginType

Current tracking origin type (*str*).

Valid tracking origin types are 'floor' and 'eye'.

units

None, 'height' (of the window), 'norm', 'deg', 'cm', 'pix' Defines the default units of stimuli initialized in the window. I.e. if you change units, already initialized stimuli won't change their units.

Can be overridden by each stimulus, if units is specified on initialization.

See *Units for the window and stimuli* for explanation of options.

update()

Deprecated: use Window.flip() instead

updateInputState(*controllers=None*)

Update all connected controller states. This updates controller input states for an input device managed by *LibOVR*.

The polling time for each device is accessible through the *controllerPollTimes* attribute. This attribute returns a dictionary where the polling time from the last *updateInputState* call for a given controller can be retrieved by using the name as a key.

Parameters

controllers (*tuple or list, optional*) – List of controllers to poll. If *None*, all available controllers will be polled.

Examples

Poll the state of specific controllers by name:

```
controllers = ['XBox', 'Touch']
updateInputState(controllers)
```

updateLights(*index=None*)

Explicitly update scene lights if they were modified.

This is required if modifications to objects referenced in *lights* have been changed since assignment. If you removed or added items of *lights* you must refresh all of them.

Parameters

index (*int, optional*) – Index of light source in *lights* to update. If *None*, all lights will be refreshed.

Examples

Call *updateLights* if you modified lights directly like this:

```
win.lights[1].diffuseColor = [1., 0., 0.]
win.updateLights(1)
```

property useLights

Enable scene lighting.

Lights will be enabled if using legacy OpenGL lighting. Stimuli using shaders for lighting should check if *useLights* is *True* since this will have no effect on them, and disable or use a no lighting shader instead. Lights will be transformed to the current view matrix upon setting to *True*.

Lights are transformed by the present *GL_MODELVIEW* matrix. Setting *useLights* will result in their positions being transformed by it. If you want lights to appear at the specified positions in world space, make sure the current matrix defines the view/eye transformation when setting *useLights=True*.

This flag is reset to *False* at the beginning of each frame. Should be *False* if rendering 2D stimuli or else the colors will be incorrect.

property userHeight

Get user height in meters (*float*).

property viewMatrix

The view matrix for the current eye buffer. Only valid after a `calcEyePoses()` call. Note that setting *viewMatrix* manually will break visibility culling.

viewOri

Set the rotation of the view in degrees.

The rotation is applied around the origin of the window, which is defined by the viewPos attribute. The rotation is applied after scaling but before translation.

viewPos

The origin of the window onto which stimulus-objects are drawn.

The value should be given in the units defined for the window. NB: Never change a single component (x or y) of the origin, instead replace the viewPos-attribute in one shot, e.g.:

```
win.viewPos = [new_xval, new_yval]  # This is the way to do it
win.viewPos[0] = new_xval  # DO NOT DO THIS! Errors will result.
```

viewScale

Set the scale factors for the view.

The scaling is applied around the origin of the window, which is defined by the viewPos attribute. The scaling is applied before translation and rotation.

property viewport

Viewport rectangle (x, y, w, h) for the current draw buffer.

Values *x* and *y* define the origin, and *w* and *h* the size of the rectangle in pixels.

This is typically set to cover the whole buffer, however it can be changed for applications like multi-view rendering. Stimuli will draw according to the new shape of the viewport, for instance and stimulus with position (0, 0) will be drawn at the center of the viewport, not the window.

Examples

Constrain drawing to the left and right halves of the screen, where stimuli will be drawn centered on the new rectangle. Note that you need to set both the *viewport* and the *scissor* rectangle:

```
x, y, w, h = win.frameBufferSize  # size of the framebuffer
win.viewport = win.scissor = [x, y, w / 2.0, h]
# draw left stimuli ...

win.viewport = win.scissor = [x + (w / 2.0), y, w / 2.0, h]
# draw right stimuli ...

# restore drawing to the whole screen
win.viewport = win.scissor = [x, y, w, h]
```

waitBlanking

After a call to `flip()` should we wait for the blank before the script continues.

property windowedSize

Size of the window to use when not fullscreen (w, h).

11.4.26 `RigidBodyPose`

Attributes

RigidBodyPose([pos, ori, dtype])	Class for representing rigid body poses.

Details

class psychopy.visual.**RigidBodyPose**(*pos=(0.0, 0.0, 0.0)*, *ori=(0.0, 0.0, 0.0, 1.0)*, *dtype=None*)

Class for representing rigid body poses.

This class is an abstract representation of a rigid body pose, where the position of the body in a scene is represented by a vector/coordinate and the orientation with a quaternion. Pose can be manipulated and interacted with using class methods and attributes. Rigid body poses assume a right-handed coordinate system (-Z is forward and +Y is up).

Poses can be converted to 4x4 transformation matrices with *getModelMatrix*. One can use these matrices when rendering to transform the vertices of a model associated with the pose by passing them to OpenGL. Matrices are cached internally to avoid recomputing them if *pos* and *ori* attributes have not been updated.

Operators * and ~ can be used on *RigidBodyPose* objects to combine and invert poses. For instance, you can multiply (*) poses to get a new pose which is the combination of both orientations and translations by:

```
newPose = rb1 * rb2
```

Likewise, a pose can be inverted by using the ~ operator:

```
invPose = ~rb
```

Multiplying a pose by its inverse will result in an identity pose with no translation and default orientation where *pos=[0, 0, 0]* and *ori=[0, 0, 0, 1]*:

```
identityPose = ~rb * rb
```

> ⚠ **Warning**
>
> This class is experimental and may result in undefined behavior.

Parameters

- **pos** (*array_like*) – Position vector *[x, y, z]* for the origin of the rigid body.
- **ori** (*array_like*) – Orientation quaternion *[x, y, z, w]* where *x*, *y*, *z* are imaginary and *w* is real.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. Default is *None* which uses the default data configured by *setDefaultPrecision*.

alignTo(*alignTo*)

Align this pose to another point or pose.

This sets the orientation of this pose to one which orients the forward axis towards *alignTo*.

Parameters
alignTo (*array_like or* RigidBodyPose) – Position vector [x, y, z] or pose to align to.

property at

Vector defining the forward direction (-Z) of this pose.

property bounds

Bounding box associated with this pose.

clear()

Clear the pose, setting position and orientation to zero.

copy()

Get a new *RigidBodyPose* object which copies the position and orientation of this one. Copies are independent and do not reference each others data.

Returns
Copy of this pose.

Return type
RigidBodyPose

distanceTo(*v*)

Get the distance to a pose or point in scene units.

Parameters
v (RigidBodyPose *or array_like*) – Pose or point [x, y, z] to compute distance to.

Returns
Distance to *v* from this pose's origin.

Return type
float

property dtype

Data type used for computations and arrays (*numpy.dtype*).

Cannot be changed after object creation.

getModelMatrix(*inverse=False*, *out=None*)

Get the present rigid body transformation as a 4x4 matrix.

Matrices are computed only if the *pos* and *ori* attributes have been updated since the last call to *getModelMatrix*. The returned matrix is an *ndarray* and row-major.

Parameters

- **inverse** (*bool, optional*) – Return the inverse of the model matrix.
- **out** (*ndarray or None*) – Optional 4x4 array to write values to. Values written are computed using 32-bit float precision regardless of the data type of *out*.

Returns
4x4 transformation matrix.

Return type
ndarray

Examples

Using a rigid body pose to transform something in OpenGL:

```
rb = RigidBodyPose((0, 0, -2))  # 2 meters away from origin

# Use `array2pointer` from `psychopy.tools.arraytools` to convert
# array to something OpenGL accepts.
mv = array2pointer(rb.modelMatrix)

# use the matrix to transform the scene
glMatrixMode(GL_MODELVIEW)
glPushMatrix()
glLoadIdentity()
glMultTransposeMatrixf(mv)

# draw the thing here ...

glPopMatrix()
```

getNormalMatrix(*out=None*)

Get the present normal matrix.

Parameters
out (*ndarray or None*) – Optional 4x4 array to write values to. Values written are computed using 32-bit float precision regardless of the data type of *out*.

Returns
4x4 normal transformation matrix.

Return type
ndarray

`getOriAxisAngle`(*degrees=True*)

Get the axis and angle of rotation for the rigid body. Converts the orientation defined by the *ori* quaternion to and axis-angle representation.

Parameters

degrees (`bool, optional`) – Specify `True` if *angle* is in degrees, or else it will be treated as radians. Default is `True`.

Returns

Axis [rx, ry, rz] and angle.

Return type

tuple

`getViewMatrix`(*inverse=False*, *out=None*)

Convert this pose into a view matrix.

Creates a view matrix which transforms points into eye space using the current pose as the eye position in the scene. Furthermore, you can use view matrices for rendering shadows if light positions are defined as *RigidBodyPose* objects.

Parameters

- **inverse** (`bool`) – Return the inverse of the view matrix. Default is *False*.
- **out** (`ndarray or None`) – Optional 4x4 array to write values to. Values written are computed using 32-bit float precision regardless of the data type of *out*.

Returns

4x4 transformation matrix.

Return type

ndarray

`getYawPitchRoll`(*degrees=True*)

Get the yaw, pitch and roll angles for this pose relative to the -Z world axis.

Parameters

degrees (`bool, optional`) – Specify `True` if *angle* is in degrees, or else it will be treated as radians. Default is `True`.

`interp`(*end*, *s*)

Interpolate between poses.

Linear interpolation is used on position (Lerp) while the orientation has spherical linear interpolation (Slerp) applied taking the shortest arc on the hypersphere.

Parameters

- **end** (`RigidBodyPose`) – End pose.
- **s** (`float`) – Interpolation factor between interval 0.0 and 1.0.

Returns

Rigid body pose whose position and orientation is at *s* between this pose and *end*.

Return type

RigidBodyPose

`property inverseModelMatrix`

Inverse of the pose as a 4x4 model matrix (read-only).

property inverseViewMatrix

The inverse of the 4x4 view matrix for this pose (read-only).

invert()

Invert this pose.

inverted()

Get a pose which is the inverse of this one.

Returns
This pose inverted.

Return type
RigidBodyPose

isEqual(*other*)

Check if poses have similar orientation and position.

Parameters
other (*RigidBodyPose*) – Other pose to compare.

Returns
Returns *True* is poses are effectively equal.

Return type
bool

property modelMatrix

Pose as a 4x4 model matrix (read-only).

property normalMatrix

The 4x4 normal transformation matrix (read-only).

property ori

Orientation quaternion (X, Y, Z, W).

property pos

Position vector (X, Y, Z).

property posOri

The position (x, y, z) and orientation (x, y, z, w).

setIdentity()

Clear rigid body transformations (alias for *clear*).

setOriAxisAngle(*axis*, *angle*, *degrees=True*)

Set the orientation of the rigid body using an *axis* and *angle*. This sets the quaternion at *ori*.

Parameters

- **axis** (*array_like*) – Axis of rotation [rx, ry, rz].
- **angle** (*float*) – Angle of rotation.
- **degrees** (*bool, optional*) – Specify `True` if *angle* is in degrees, or else it will be treated as radians. Default is `True`.

transform(*v*, *out=None*)

Transform a vector using this pose.

Parameters

- **v** (*array_like*) – Vector to transform [x, y, z].
- **out** (*ndarray or None, optional*) – Optional array to write values to. Must have the same shape as *v*.

Returns
Transformed points.

Return type
ndarray

transformNormal(*n*)

Rotate a normal vector with respect to this pose.

Rotates a normal vector *n* using the orientation quaternion at *ori*.

Parameters
n (*array_like*) – Normal to rotate (1-D with length 3).

Returns
Rotated normal *n*.

Return type
ndarray

property up

Vector defining the up direction (+Y) of this pose.

property viewMatrix

The 4x4 view matrix for this pose (read-only).

11.4.27 SceneSkybox

Attributes

SceneSkybox(win[, tex, ori, axis])	Class to render scene skyboxes.

Details

class psychopy.visual.**SceneSkybox**(*win*, *tex=()*, *ori=0.0*, *axis=(0, 1, 0)*)

Class to render scene skyboxes. This is a lazy-imported class, therefore import using full path *from psychopy.visual.stim3d import SceneSkybox* when inheriting from it.

A skybox provides background imagery to serve as a visual reference for the scene. Background images are projected onto faces of a cube centered about the viewpoint regardless of any viewpoint translations, giving the illusion that the background is very far away. Usually, only one skybox can be rendered per buffer each frame. Render targets must have a depth buffer associated with them.

Background images are specified as a set of image paths passed to *faceTextures*:

```
sky = SceneSkybox(
    win, ('rt.jpg', 'lf.jpg', 'up.jpg', 'dn.jpg', 'bk.jpg', 'ft.jpg'))
```

The skybox is rendered by calling *draw()* after drawing all other 3D stimuli.

Skyboxes are not affected by lighting, however, their colors can be modulated by setting the window's *sceneAmbient* value. Skyboxes should be drawn after all other 3D stimuli, but before any successive call that clears the depth buffer (eg. *setPerspectiveView*, *resetEyeTransform*, etc.)

Parameters

- **win** (*~psychopy.visual.Window*) – Window this skybox is associated with.
- **tex** (`list or tuple or TexCubeMap`) – List of files paths to images to use for each face. Images are assigned to faces depending on their index within the list ([+X, -X, +Y, -Y, +Z, -Z] or [right, left, top, bottom, back, front]). If *None* is specified, the cube map may be specified later by setting the *cubemap* attribute. Alternatively, you can specify a *TexCubeMap* object to set the cube map directly.
- **ori** (`float`) – Rotation of the skybox about *axis* in degrees.
- **axis** (`array_like`) – Axis [ax, ay, az] to rotate about, default is (0, 1, 0).

draw(*win=None*)

Draw the skybox.

This should be called last after drawing other 3D stimuli for performance reasons.

Parameters

win (*~psychopy.visual.Window*, optional) – Window to draw the skybox to. If *None*, the window set when initializing this object will be used. The window must share a context with the window which this objects was initialized with.

property skyCubeMap

Cubemap for the sky.

11.4.28 EnvelopeGrating

Attributes

Details

class psychopy.visual.**EnvelopeGrating**(**args*, ***kwargs*)

psychopy.visual.secondorder is now located within the psychopy-visionscience plugin.

When initialised, rather than creating an object, will log an error.

11.4.29 psychopy.visual.ShapeStim

ShapeStim is the base class for drawing lines and polygons.

Overview

Details

class psychopy.visual.shape.**ShapeStim**(*win*, *units=''*, *colorSpace='rgb'*, *fillColor=False*, *lineColor=False*, *lineWidth=1.5*, *vertices=((-0.5, 0), (0, 0.5), (0.5, 0))*, *windingRule=None*, *closeShape=True*, *pos=(0, 0)*, *size=1*, *anchor=None*, *ori=0.0*, *opacity=1.0*, *contrast=1.0*, *depth=0*, *interpolate=True*, *draggable=False*, *name=None*, *autoLog=None*, *autoDraw=False*, *color=undefined*, *lineRGB=undefined*, *fillRGB=undefined*, *fillColorSpace=undefined*, *lineColorSpace=undefined*)

A class for arbitrary shapes defined as lists of vertices (x,y). This is a lazy-imported class, therefore import using full path *from psychopy.visual.shape import ShapeStim* when inheriting from it.

Shapes can be lines, polygons (concave, convex, self-crossing), or have holes or multiple regions.

vertices is typically a list of points (x,y). By default, these are assumed to define a closed figure (polygon); set *closeShape=False* for a line. *closeShape* cannot be changed dynamically, but individual vertices can be changed on a frame-by-frame basis. The stimulus as a whole can be rotated, translated, or scaled dynamically (using .ori, .pos, .size).

Vertices can be a string, giving the name of a known set of vertices, although "cross" is the only named shape available at present.

Advanced shapes: *vertices* can also be a list of loops, where each loop is a list of points (x,y), e.g., to define a shape with a hole. Borders and contains() are not supported for multi-loop stimuli.

windingRule is an advanced feature to allow control over the GLU tessellator winding rule (default: GLU_TESS_WINDING_ODD). This is relevant only for self-crossing or multi-loop shapes. Cannot be set dynamically.

See Coder demo > stimuli > shapes.py

Changed Nov 2015: v1.84.00. Now allows filling of complex shapes. This is almost completely backwards compatible (see changelog). The old version is accessible as *psychopy.visual.BaseShapeStim*.

Parameters

- **win** (*Window*) – Window this shape is being drawn to. The stimulus instance will allocate its required resources using that Windows context. In many cases, a stimulus instance cannot be drawn on different windows unless those windows share the same OpenGL context, which permits resources to be shared between them.
- **units** (*str*) – Units to use when drawing. This will affect how parameters and attributes *pos*, *size* and *radius* are interpreted.
- **colorSpace** (*str*) – Sets the colorspace, changing how values passed to *lineColor* and *fillColor* are interpreted.
- **lineWidth** (*float*) – Width of the shape outline.
- **lineColor** (array_like, str, *Color* or None) – Color of the shape outline and fill. If *None*, a fully transparent color is used which makes the fill or outline invisible.
- **fillColor** (array_like, str, *Color* or None) – Color of the shape outline and fill. If *None*, a fully transparent color is used which makes the fill or outline invisible.
- **vertices** (*array_like*) – Nx2 array of points (eg., *[[-0.5, 0], [0, 0.5], [0.5, 0]*).
- **windingRule** (GLenum or None) – Winding rule to use for tesselation, default is *GLU_TESS_WINDING_ODD* if *None* is specified.
- **closeShape** (*bool*) – Close the shape's outline. If *True* the first and last vertex will be joined by an edge. Must be *True* to use tesselation. Default is *True*.
- **pos** (*array_like*) – Initial position (*x*, *y*) of the shape on-screen relative to the origin located at the center of the window or buffer in *units*. This can be updated after initialization by setting the *pos* property. The default value is *(0.0, 0.0)* which results in no translation.
- **size** (*array_like, float, int or None*) – Width and height of the shape as *(w, h)* or *[w, h]*. If a single value is provided, the width and height will be set to the same specified value. If *None* is specified, the *size* will be set with values passed to *width* and *height*.
- **ori** (*float*) – Initial orientation of the shape in degrees about its origin. Positive values will rotate the shape clockwise, while negative values will rotate counterclockwise. The default value for *ori* is 0.0 degrees.
- **opacity** (*float*) – Opacity of the shape. A value of 1.0 indicates fully opaque and 0.0 is fully transparent (therefore invisible). Values between 1.0 and 0.0 will result in colors being

blended with objects in the background. This value affects the fill (*fillColor*) and outline (*lineColor*) colors of the shape.

- **contrast** (*float*) – Contrast level of the shape (0.0 to 1.0). This value is used to modulate the contrast of colors passed to *lineColor* and *fillColor*.
- **depth** (*int*) – Depth layer to draw the shape when *autoDraw* is enabled. *DEPRECATED*
- **interpolate** (*bool*) – Enable smoothing (anti-aliasing) when drawing shape outlines. This produces a smoother (less-pixelated) outline of the shape.
- **draggable** (*bool*) – Can this stimulus be dragged by a mouse click?
- **name** (*str*) – Optional name of the stimuli for logging.
- **autoLog** (*bool*) – Enable auto-logging of events associated with this stimuli. Useful for debugging and to track timing when used in conjunction with *autoDraw*.
- **autoDraw** (*bool*) – Enable auto drawing. When *True*, the stimulus will be drawn every frame without the need to explicitly call the `draw()` method.
- **color** (array_like, str, `Color` or None) – Synonymous with *fillColor*

`property RGB`

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type
DEPRECATED

`static _calcEquilateralVertices`(*edges*, *radius=0.5*)

Get vertices for an equilateral shape with a given number of sides, will assume radius is 0.5 (relative) but can be manually specified

`_calcPosRendered()`

DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix.

`_calcSizeRendered()`

DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix

`static _calculateMinEdges`(*lineWidth*, *threshold=180*)

Calculate how many points are needed in an equilateral polygon for the gap between line rects to be < 1px and for corner angles to exceed a threshold.

In other words, how many edges does a polygon need to have to appear smooth?

lineWidth
[int, float, np.ndarray] Width of the line in pixels

threshold
[int] Maximum angle (degrees) for corners of the polygon, useful for drawing a circle. Supply 180 for no maximum angle.

`_drawLegacyGL`(*win*, *keepMatrix*)

Legacy draw the stimulus in the relevant window.

You must call this method after every *win.flip()* if you want the stimulus to appear on that frame and then update the screen again.

`_getDesiredRGB`(*rgb*, *colorSpace*, *contrast*)

Convert color to RGB while adding contrast. Requires self.rgb, self.colorSpace and self.contrast `_getPolyAsRendered()`

DEPRECATED. Return a list of vertices as rendered.

`_legacyTesselate(`*newVertices*`)`

Legacy tessellation method for ShapeStim.

`_selectWindow(`*win*`)`

Switch drawing to the specified window. Calls the window's _setCurrent() method which handles the switch.

`_set(`*attrib*, *val*, *op=''*, *log=None*`)`

DEPRECATED since 1.80.04 + 1. Use setAttribute() and val2array() instead.

`_tesselate(`*newVertices*`)`

Set the *.vertices* and *.border* to new values, invoking tessellation.

Parameters

`newVertices` (*array_like*) – Nx2 array of points (eg., *[[-0.5, 0], [0, 0.5], [0.5, 0]*).

`_updateList()`

The user shouldn't need this method since it gets called after every call to .set() Chooses between using and not using shaders each call.

`_updateVertices()`

Sets Stim.verticesPix and ._borderPix from pos, size, ori, flipVert, flipHoriz

`alphaThreshold`

Threshold for alpha values.

If the alpha value of a pixel is below this threshold, the pixel will be rejected (not drawn). This can be useful for creating a mask from an image with an alpha channel. The default value is 0.0, which means that no thresholding will be applied.

`autoDraw`

Determines whether the stimulus should be automatically drawn on every frame flip.

Value should be: *True* or *False*. You do NOT need to set this on every frame flip!

`autoLog`

Whether every change in this stimulus should be auto logged.

Value should be: *True* or *False*. Set to *False* if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren't likely to be useful.

`property backColor`

Alternative way of setting fillColor

`property backColorSpace`

Deprecated, please use colorSpace to set color space for the entire object.

`property backRGB`

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type

DEPRECATED

`property backgroundColor`

Alternative way of setting fillColor

property borderColorSpace

Deprecated, please use colorSpace to set color space for the entire object

property borderRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type
DEPRECATED

closeShape

Should the last vertex be automatically connected to the first?

If you're using *Polygon*, *Circle* or *Rect*, *closeShape=True* is assumed and shouldn't be changed.

color

Set the color of the shape. Sets both *fillColor* and *lineColor* simultaneously if applicable.

property colorSpace

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                       color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

contains(*x*, *y=None*, *units=None*)

Returns True if a point x,y is inside the stimulus' border.

Can accept variety of input options:

- two separate args, x and y
- one arg (list, tuple or array) containing two vals (x,y)
- **an object with a getPos() method that returns x,y, such**
  as a `Mouse`.

Returns *True* if the point is within the area defined either by its *border* attribute (if one defined), or its *vertices* attribute if there is no .border. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the *contains* method; the extent of the stimulus is determined purely by the size, position (pos), and orientation (ori) settings (and by the vertices for shape stimuli).

See Coder demos: shapeContains.py See Coder demos: shapeContains.py

property contrast

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged).
Operations supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast =  1.0  # unchanged contrast
stim.contrast =  0.5  # decrease contrast
stim.contrast =  0.0  # uniform, no contrast
stim.contrast = -0.5  # slightly inverted
stim.contrast = -1.0  # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast =  1.2  # increases contrast
stim.contrast = -1.2  # inverts with increased contrast
```

depth

DEPRECATED, depth is now controlled simply by drawing order.

doDragging()

If this stimulus is draggable, do the necessary actions on a frame flip to drag it.

draggable

Can this stimulus be dragged by a mouse click?

draw(*win=None*, *keepMatrix=False*)

Draw the stimulus in the relevant window.

You must call this method after every *win.flip()* if you want the stimulus to appear on that frame and then update the screen again.

Parameters

- **win** (*Window*, optional) – Window to draw the stimulus in. If not specified, the stimulus will be drawn in the window specified at initialization.
- **keepMatrix** (*bool, optional*) – *DEPRECATED* If *True*, the current transformation matrix will be preserved. This is useful when drawing multiple stimuli with the same transformation matrix. Default is *False*.

property fillColor

Set the fill color for the shape.

property fillColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property fillRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type

DEPRECATED

property flip

1x2 array for flipping vertices along each axis; set as True to flip or False to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (*._flip*) will give an array of 1s and -1s with which to multiply vertices.

property fontColor

Alternative way of setting *foreColor*.

property foreColor

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names <http://www.w3schools.com/html/html_colornames.asp>* can be used.
- *Colors by hex value*
- **numerically: (scalar or triplet) for DKL, RGB or**
  other *Color spaces*. For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue'  # (the case is actually ignored)
stim.color = '#DDA0DD'  # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0]  # if stim.colorSpace='rgb':
                # a red color in rgb space
stim.color = [0.0, 45.0, 1.0]  # if stim.colorSpace='dkl':
                # DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255]  # if stim.colorSpace='rgb255':
                # a blue stimulus using rgb255 space
stim.color = 255  # interpreted as (255, 255, 255)
                  # which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1]  # increment all guns by 1 value
stim.color *= -1  # multiply the color by -1 (which in this
                    # space inverts the contrast)
stim.color *= [0.5, 0, 1]  # decrease red, remove green, keep blue
```

You can use *setColor* if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

property foreColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

`property foreRGB`

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type
DEPRECATED

`interpolate`

If *True* the edge of the line will be anti-aliased.

`property lineColor`

Alternative way of setting *borderColor*.

`property lineColorSpace`

Deprecated, please use colorSpace to set color space for the entire object

`property lineRGB`

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type
DEPRECATED

`lineWidth`

Width of the line in **pixels**.

Operations supported.

`name`

The name (*str*) of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = None your stimulus will be called "unnamed <type>", e.g. visual.TextStim(win) will be called "unnamed TextStim" in the logs.

`property opacity`

Determines how visible the stimulus is relative to background.

The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). *Operations* are supported. Precisely how this is used depends on the *Blend Mode*.

`ori`

The orientation of the stimulus (in degrees).

Should be a single value (*scalar*). *Operations* are supported.

Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and below zero values wrap appropriately.

`overlaps`(*polygon*)

Returns *True* if this stimulus intersects another one.

If *polygon* is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the *overlaps* method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

See coder demo, shapeContains.py

property pos

The position of the center of the stimulus in the stimulus *units*

value should be an *x,y-pair*. *Operations* are also supported.

Example:

```
stim.pos = (0.5, 0)  # Set slightly to the right of center
stim.pos += (0.5, -1)  # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2  # Move stim towards the center.
    Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:

```
from psychopy.tools.monitorunittools import posToPix
posPix = posToPix(stim)
```

setAlphaThreshold(*value*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setAutoDraw(*value*, *log=None*)

Sets autoDraw. Usually you can use 'stim.attribute = value' syntax instead, but use this method to suppress the log message.

setAutoLog(*value=True*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setBackRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setBorderColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setBorderRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Sets both the line and fill to be the same color.

setContrast(*newContrast*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setDKL(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setDepth(*newDepth*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setFillColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for fillColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setFillRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setForeColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for foreColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setForeRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setLMS(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setLineRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setOpacity(*newOpacity*, *operation=''*, *log=None*)

Hard setter for opacity, allows the suppression of log messages and calls the update method

setOri(*newOri*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setPos(*newPos*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setSize(*newSize*, *operation=''*, *units=None*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setVertices(*value=None*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

property size

The size (width, height) of the stimulus in the stimulus *units*

Value should be *x,y-pair*, *scalar* (applies to both dimensions) or None (resets to default). *Operations* are supported.

Sizes can be negative (causing a mirror-image reversal) and can extend beyond the window.

Example:

```
stim.size = 0.8  # Set size to (xsize, ysize) = (0.8, 0.8)
print(stim.size)  # Outputs array([0.8, 0.8])
stim.size += (0.5, -0.5)  # make wider and flatter: (1.3, 0.3)
```

Tip: if you can see the actual pixel range this corresponds to by looking at *stim._sizeRendered*

updateColors()

Placeholder method to update colours when set externally, for example updating the *pallette* attribute of a textbox

updateOpacity()

Placeholder method to update colours when set externally, for example updating the *pallette* attribute of a textbox.

property vertices

A list of lists or a numpy array (Nx2) specifying xy positions of each vertex, relative to the center of the field.

Assigning to vertices can be slow if there are many vertices.

Operations supported with *.setVertices()*.

property verticesPix

This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

property win

The `Window` object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1  # stimulus will be drawn in win1
stim.draw()  # stimulus is now drawn to win1
stim.win = win2  # stimulus will be drawn in win2
stim.draw()  # it is now drawn in win2
win1.flip(waitBlanking=False)  # do not wait for next
             # monitor update
win2.flip()  # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

11.4.30 SimpleImageStim

class psychopy.visual.**SimpleImageStim**(**args*, ***kwargs*)

A simple stimulus for loading images from a file and presenting at exactly the resolution and color in the file (subject to gamma correction if set). This is a lazy-imported class, therefore import using full path *from psychopy.visual.simpleimage import SimpleImageStim* when inheriting from it.

Unlike the ImageStim, this type of stimulus cannot be rescaled, rotated or masked (although flipping horizontally or vertically is possible). Drawing will also tend to be marginally slower, because the image isn't preloaded to the graphics card. The slight advantage, however is that the stimulus will always be in its original aspect ratio, with no interplotation or other transformation, and it is slightly faster to load into PsychoPy.

11.4.31 Slider

A class for obtaining ratings, e.g., on a 1-to-7 or categorical scale. This is a lazy-imported class, therefore import using full path *from psychopy.visual.slider import Slider* when inheriting from it.

Attributes

Slider(win[, ticks, labels, startValue, ...])	A class for obtaining ratings, e.g., on a 1-to-7 or categorical scale.
Slider.getRating()	Get the current value of rating (or None if no response yet)
Slider.getRT()	Get the RT for most recent rating (or None if no response yet)
Slider.markerPos	The position on the scale where the marker should be.
Slider.setReadOnly([value, log])	When the rating scale is read only no responses can be made and the scale contrast is reduced
Slider.contrast	Set all elements of the Slider (labels, ticks, line) to a contrast
Slider.style	
Slider.getHistory()	Return a list of the subject's history as (rating, time) tuples.
Slider.getMouseResponses()	Instructs the rating scale to check for valid mouse responses.
Slider.reset()	Resets the slider to its starting state (so that it can be restarted on each trial with a new stimulus)

Details

`class psychopy.visual.Slider`(*win*, *ticks=(1, 2, 3, 4, 5)*, *labels=None*, *startValue=None*, *pos=(0, 0)*, *size=None*, *units=None*, *flip=False*, *ori=0*, *style='rating'*, *styleTweaks=[]*, *granularity=0*, *readOnly=False*, *labelColor='White'*, *markerColor='Red'*, *lineColor='White'*, *colorSpace='rgb'*, *opacity=None*, *font='Helvetica Bold'*, *depth=0*, *name=None*, *labelHeight=None*, *labelWrapWidth=None*, *autoDraw=False*, *autoLog=True*, *color=False*, *fillColor=False*, *borderColor=False*)

A class for obtaining ratings, e.g., on a 1-to-7 or categorical scale.

A simpler alternative to RatingScale, to be customised with code rather than with arguments.

A Slider instance is a re-usable visual object having a `draw()` method, with customizable appearance and response options. `draw()` displays the rating scale, handles the subject’s mouse or key responses, and updates the display. As soon as a rating is supplied, `.rating`

> will go from `None` to selected item

You can call the `getRating()` method anytime to get a rating, `getRT()` to get the decision time, or `getHistory()` to obtain the entire set of (rating, RT) pairs.

For other examples see Coder Demos -> stimuli -> ratingsNew.py.

Authors

- 2018: Jon Peirce

Parameters

- **win** (`psychopy.visual.Window`) – Into which the scale will be rendered
- **ticks** (*list or tuple, optional*) – A set of values for tick locations. If given a list of numbers then these determine the locations of the ticks (the first and last determine the endpoints and the rest are spaced according to their values between these endpoints.

- **labels** (*a list or tuple, optional*) – The text to go with each tick (or spaced evenly across the ticks). If you give 3 labels but 5 tick locations then the end and middle ticks will be given labels. If the labels can't be distributed across the ticks then an error will be raised. If you want an uneven distribution you should include a list matching the length of ticks but with some values set to None
- **startValue** (*int or float, optional*) – The initial position of the marker on the slider. If not specified, the marker will start at the mid-point of the scale.
- **pos** (*tuple, list, or array, optional*) – The (x, y) position of the slider on the screen.
- **size** (*w,h pair (tuple, array or list)*) – The size for the scale defines the area taken up by the line and the ticks. This also controls whether the scale is horizontal or vertical.
- **units** (*str, optional*) – The units to interpret the *pos* and *size* parameters. Can be any of the standard PsychoPy units (e.g., 'pix', 'cm', 'norm').
- **flip** (*bool, optional*) – If *True*, the labels will be placed above (for horizontal sliders) or to the right (for vertical sliders) of the slider line. Default is *False*.
- **ori** (*int or float, optional*) – The orientation of the slider in degrees. A value of 0 means no rotation, positive values rotate the slider clockwise.

  style : str or list of str, optional

  The style of the slider, e.g., 'rating', 'slider', 'radio'. Multiple styles can be combined in a list.

  styleTweaks : list of str, optional

  Additional styling tweaks, e.g., 'triangleMarker', 'labels45'.
- **granularity** (*int or float*) – The smallest valid increments for the scale. 0 gives a continuous (e.g. "VAS") scale. 1 gives a traditional likert scale. Something like 0.1 gives a limited fine-grained scale.
- **readOnly** (*bool, optional*) – If *True*, the slider is displayed but does not accept input.
- **labelColor** (*color, optional*) – The color of the labels in the specified color space.
- **markerColor** (*color, optional*) – The color of the marker in the specified color space.
- **lineColor** (*color, optional*) – The color of the slider line and ticks in the specified color space.
- **colorSpace** (*str, optional*) – The color space for defining *labelColor*, *markerColor*, and *lineColor* (e.g., 'rgb', 'rgb255', 'hex').
- **opacity** (*float, optional*) – The opacity of the slider, ranging from 0 (completely transparent) to 1 (completely opaque).
- **font** (*str, optional*) – The font used for the labels.
- **depth** (*int, optional*) – The depth layer for rendering. Layers with lower numbers are rendered first (behind).
- **name** (*str, optional*) – An optional name for the slider, useful for logging.
- **labelHeight** (*float, optional*) – The height of the label text. If *None*, a default value based on the slider size is used.
- **labelWrapWidth** (*float, optional*) – The maximum width for text labels before wrapping. If *None*, labels are not wrapped.

- **autoDraw** (*bool, optional*) – If *True*, the slider will be automatically drawn every frame.
- **autoLog** (*bool, optional*) – If *True*, a log message is automatically generated each time the slider is updated. This can be useful for debugging or analysis.
- **color** (*color, optional*) – Synonym for *labelColor*.
- **fillColor** (*color, optional*) – Synonym for *markerColor*.
- **borderColor** (*color, optional*) – Synonym for *lineColor*.

_getHitboxParams()

Calculates hitbox size and pos from own size and pos

_getLineParams()

Calculates location and size of the line based on own location and size

_getMarkerParams()

Calculates location and size of marker based on own location and size

_getTickParams()

Calculates the locations of the line, tickLines and labels from the rating info

_granularRating(*rating*)

Handle granularity for the rating

property **borderColor**

property **categorical**

(readonly) determines from labels and ticks whether the slider is categorical

contrast

Set all elements of the Slider (labels, ticks, line) to a contrast

Parameters
contrast

draw()

Draw the Slider, with all its constituent elements on this frame

property **extent**

The distance from the leftmost point on the slider to the rightmost point, and from the highest point to the lowest.

property **fillColor**

Set the fill color for the shape.

property **flip**

1x2 array for flipping vertices along each axis; set as True to flip or False to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (*._flip*) will give an array of 1s and -1s with which to multiply vertices.

property **foreColor**

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names <http://www.w3schools.com/html/html_colornames.asp>* can be used.
- *Colors by hex value*

- **numerically: (scalar or triplet) for DKL, RGB or**
  other *Color spaces*. For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue'  # (the case is actually ignored)
stim.color = '#DDA0DD'  # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0]  # if stim.colorSpace='rgb':
                # a red color in rgb space
stim.color = [0.0, 45.0, 1.0]  # if stim.colorSpace='dkl':
                # DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255]  # if stim.colorSpace='rgb255':
                # a blue stimulus using rgb255 space
stim.color = 255  # interpreted as (255, 255, 255)
                  # which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1]  # increment all guns by 1 value
stim.color *= -1  # multiply the color by -1 (which in this
                    # space inverts the contrast)
stim.color *= [0.5, 0, 1]  # decrease red, remove green, keep blue
```

You can use *setColor* if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

getAnchor()

getBackColor()

getBackColorSpace()

getBackRGB()

getBackgroundColor()

getBorderColor()

getBorderColorSpace()

getBorderRGB()

getBorderWidth()

getCategorical()

getColor()

getColorSpace()

getContrast()

getExtent()

getFillColor()

getFillColorSpace()

getFillRGB()

getFlip()

getFlipHoriz()

getFlipVert()

getFontColor()

getForeColor()

getForeColorSpace()

getForeRGB()

getHeight()

getHistory()

Return a list of the subject's history as (rating, time) tuples.

The history can be retrieved at any time, allowing for continuous ratings to be obtained in real-time. Both numerical and categorical choices are stored automatically in the history.

getHoriz()

getLabelColor()

getLabelHeight()

getLabelWrapWidth()

getLineColor()

getLineColorSpace()

getLineRGB()

getLineWidth()

getMarkerColor()

getMarkerPos()

Get the current marker position (or None if no response yet) `getMouseResponses()`

Instructs the rating scale to check for valid mouse responses.

This is usually done during the draw() method but can be done by the user as well at any point in time. The rating will be returned but will ALSO automatically be set as the current rating response.

While the mouse button is down we will alter self.markerPos but don't set a value for self.rating until button comes up

Return type

A rating value or None

`getOpacity()`

`getPos()`

`getRGB()`

`getRT()`

Get the RT for most recent rating (or None if no response yet)

`getRating()`

Get the current value of rating (or None if no response yet)

`getSize()`

`getStyle()`

`getStyleTweaks()`

`getTicks()`

`getUnits()`

`getValue()`

`getVertices()`

`getWidth()`

`getWin()`

`get_tickL()`

`property horiz`

(readonly) determines from self.size whether the scale is horizontal

`knownStyleTweaks = ['labels45', 'triangleMarker']`

`knownStyles = ['slider', 'rating', 'radio', 'scrollbar', 'choice']`

`property labelColor`

Synonym of Slider.foreColor

`property labelHeight`

`property labelWrapWidth`

`legacyStyleTweaks = ['whiteOnBlack']`

`legacyStyles = []`

`property markerColor`

Synonym of Slider.fillColor

`markerPos`

The position on the scale where the marker should be. Note that this does not alter the value of the reported rating, only its visible display. Also note that this position is in scale units, not in coordinates

`property opacity`

`property pos`

`property rating`

`recordRating`(*rating*, *rt=None*, *log=None*)

Sets the current rating value

`reset`()

Resets the slider to its starting state (so that it can be restarted on each trial with a new stimulus)

`setBackColorSpace`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setBorderColorSpace`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setColorSpace`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setExtent`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setFillColorSpace`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setFlip`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setFlipHoriz`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setFlipVert`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setForeColorSpace`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setHeight`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setLabelColor`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setLabelHeight`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setLabelWrapWidth`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setLineColorSpace`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setMarkerColor`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setMarkerPos`(*rating*)

Set the current marker position (or None if no response yet)

Parameters

`rating` (*int or float*) – The rating on the scale where we want to set the marker

`setOpacity`(*newOpacity*, *operation=''*, *log=None*)

`setOri`(*newOri*, *operation=''*, *log=None*)

`setPos`(*newPos*, *operation=''*, *log=None*)

setRating(*value*, *log=None*, *operation=False*, *stealth=False*)

setReadOnly(*value=True*, *log=None*)

When the rating scale is read only no responses can be made and the scale contrast is reduced

Parameters

- **value** (`bool (True)`) – The value to which we should set the readOnly flag
- **log** (`bool or None`) – Force the autologging to occur or leave as default

setSize(*newSize*, *operation=''*, *units=None*, *log=None*)

setStyle(*value*, *log=None*, *operation=False*, *stealth=False*)

setStyleTweaks(*value*, *log=None*, *operation=False*, *stealth=False*)

setTicks(*value*, *log=None*, *operation=False*, *stealth=False*)

setUnits(*value*, *log=None*, *operation=False*, *stealth=False*)

setValue(*value*, *log=None*, *operation=False*, *stealth=False*)

setVertices(*value*, *log=None*, *operation=False*, *stealth=False*)

setWidth(*value*, *log=None*, *operation=False*, *stealth=False*)

setWin(*value*, *log=None*, *operation=False*, *stealth=False*)

property size

property style

styleTweaks

Sets some predefined style tweaks or use these to create your own.

If you fancy creating and including your own style tweaks that would be great!

Parameters

styleTweaks (`list of strings`) – Known style tweaks currently include:

’triangleMarker’: the marker is a triangle ‘labels45’: the text is rotated by 45 degrees

Legacy style tweaks include:

’whiteOnBlack’: a sort of color-inverse rating scale

Legacy style tweaks will work if set in code, but are not exposed in Builder as they are redundant

Style tweaks can be combined in a list e.g. *[‘labels45’]*

ticks

property units

updateOpacity()

property value

Synonymous with .rating

11.4.32 `SphereStim`

Attributes

SphereStim(win[, radius, subdiv, flipFaces, ...])	Class for drawing a UV sphere.

Details

class psychopy.visual.**SphereStim**(*win*, *radius=0.5*, *subdiv=(32, 32)*, *flipFaces=False*, *pos=(0.0, 0.0, 0.0)*, *ori=(0.0, 0.0, 0.0, 1.0)*, *color=(0.0, 0.0, 0.0)*, *colorSpace='rgb'*, *contrast=1.0*, *opacity=1.0*, *useMaterial=None*, *name=''*, *autoLog=True*)

Class for drawing a UV sphere. This is a lazy-imported class, therefore import using full path *from psychopy.visual.stim3d import SphereStim* when inheriting from it.

The resolution of the sphere mesh can be controlled by setting *sectors* and *stacks* which controls the number of latitudinal and longitudinal subdivisions, respectively. The radius of the sphere is defined by setting *radius* expressed in scene units (meters if using a perspective projection).

Calling the *draw* method will render the sphere to the current buffer. The render target (FBO or back buffer) must have a depth buffer attached to it for the object to be rendered correctly. Shading is used if the current window has light sources defined and lighting is enabled (by setting *useLights=True* before drawing the stimulus).

> ⚠ **Warning**
>
> This class is experimental and may result in undefined behavior.

Examples

Creating a red sphere 1.5 meters away from the viewer with radius 0.25:

```
redSphere = SphereStim(win,
                       pos=(0., 0., -1.5),
                       radius=0.25,
                       color=(1, 0, 0))
```

Parameters

- **win** (*~psychopy.visual.Window*) – Window this stimulus is associated with. Stimuli cannot be shared across windows unless they share the same context.
- **radius** (*float*) – Radius of the sphere in scene units.
- **subdiv** (*array_like*) – Number of latitudinal and longitudinal subdivisions *(lat, long)* for the sphere mesh. The greater the number, the smoother the sphere will appear.
- **flipFaces** (*bool, optional*) – If *True*, normals and face windings will be set to point inward towards the center of the sphere. Texture coordinates will remain the same. Default is *False*.
- **pos** (*array_like*) – Position vector *[x, y, z]* for the origin of the rigid body.
- **ori** (*array_like*) – Orientation quaternion *[x, y, z, w]* where *x*, *y*, *z* are imaginary and *w* is real. If you prefer specifying rotations in axis-angle format, call *setOriAxisAngle* after initialization.

- **useMaterial** (*PhongMaterial, optional*) – Material to use. The material can be configured by accessing the *material* attribute after initialization. If not material is specified, the diffuse and ambient color of the shape will be set by *color*.
- **color** (*array_like*) – Diffuse and ambient color of the stimulus if *useMaterial* is not specified. Values are with respect to *colorSpace*.
- **colorSpace** (*str*) – Colorspace of *color* to use.
- **contrast** (*float*) – Contrast of the stimulus, value modulates the *color*.
- **opacity** (*float*) – Opacity of the stimulus ranging from 0.0 to 1.0. Note that transparent objects look best when rendered from farthest to nearest.
- **name** (*str*) – Name of this object for logging purposes.
- **autoLog** (*bool*) – Enable automatic logging on attribute changes.

property RGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type

DEPRECATED

_createVAO(*vertices*, *textureCoords*, *normals*, *faces*)

Create a vertex array object for handling vertex attribute data.

_getDesiredRGB(*rgb*, *colorSpace*, *contrast*)

Convert color to RGB while adding contrast. Requires self.rgb, self.colorSpace and self.contrast

_selectWindow(*win*)

Switch drawing to the specified window. Calls the window's _setCurrent() method which handles the switch.

_updateList()

The user shouldn't need this method since it gets called after every call to .set() Chooses between using and not using shaders each call.

property anchor

property backColor

Alternative way of setting fillColor

property backColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property backRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type

DEPRECATED

property backgroundColor

Alternative way of setting fillColor

property borderColor

property borderColorSpace

Deprecated, please use colorSpace to set color space for the entire object

property borderRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type
DEPRECATED

borderWidth

property color

Alternative way of setting *foreColor*.

property colorSpace

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                       color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

property contrast

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged).
Operations supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast =  1.0  # unchanged contrast
stim.contrast =  0.5  # decrease contrast
stim.contrast =  0.0  # uniform, no contrast
stim.contrast = -0.5  # slightly inverted
stim.contrast = -1.0  # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast =  1.2  # increases contrast
stim.contrast = -1.2  # inverts with increased contrast
```

draw(*win=None*)

Draw the stimulus.

This should work for stimuli using a single VAO and material. More complex stimuli with multiple materials should override this method to correctly handle that case.

Parameters

`win` (*~psychopy.visual.Window*) – Window this stimulus is associated with. Stimuli cannot be shared across windows unless they share the same context.

`property fillColor`

Set the fill color for the shape.

`property fillColorSpace`

Deprecated, please use colorSpace to set color space for the entire object.

`property fillRGB`

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type

DEPRECATED

`property flip`

1x2 array for flipping vertices along each axis; set as True to flip or False to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (*._flip*) will give an array of 1s and -1s with which to multiply vertices.

`property flipHoriz`

`property flipVert`

`property fontColor`

Alternative way of setting *foreColor*.

`property foreColor`

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names <http://www.w3schools.com/html/html_colornames.asp>* can be used.
- *Colors by hex value*
- **numerically: (scalar or triplet) for DKL, RGB or**
  other *Color spaces*. For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue'  # (the case is actually ignored)
stim.color = '#DDA0DD'  # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0]  # if stim.colorSpace='rgb':
                # a red color in rgb space
stim.color = [0.0, 45.0, 1.0]  # if stim.colorSpace='dkl':
                # DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255]  # if stim.colorSpace='rgb255':
```

(continues on next page)

(continued from previous page)

```
                # a blue stimulus using rgb255 space
stim.color = 255  # interpreted as (255, 255, 255)
                  # which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1]  # increment all guns by 1 value
stim.color *= -1  # multiply the color by -1 (which in this
                    # space inverts the contrast)
stim.color *= [0.5, 0, 1]  # decrease red, remove green, keep blue
```

You can use *setColor* if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

property foreColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property foreRGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type

DEPRECATED

getOri()

getOriAxisAngle(*degrees=True*)

Get the axis and angle of rotation for the 3D stimulus. Converts the orientation defined by the *ori* quaternion to and axis-angle representation.

Parameters

degrees (*bool, optional*) – Specify `True` if *angle* is in degrees, or else it will be treated as radians. Default is `True`.

Returns

Axis *[rx, ry, rz]* and angle.

Return type

tuple

getPos()

getRayIntersectBounds(*rayOrig*, *rayDir*)

Get the point which a ray intersects the bounding box of this mesh.

Parameters

- **rayOrig** (*array_like*) – Origin of the ray in space [x, y, z].
- **rayDir** (*array_like*) – Direction vector of the ray [x, y, z], should be normalized.

Returns

Coordinate in world space of the intersection and distance in scene units from *rayOrig*. Returns *None* if there is no intersection.

Return type

tuple

getRayIntersectSphere(*rayOrig*, *rayDir*)

Get the point which a ray intersects the sphere.

Parameters

- **rayOrig** (*array_like*) – Origin of the ray in space [x, y, z].
- **rayDir** (*array_like*) – Direction vector of the ray [x, y, z], should be normalized.

Returns

Coordinate in world space of the intersection and distance in scene units from *rayOrig*. Returns *None* if there is no intersection.

Return type

tuple

property height

isVisible()

Check if the object is visible to the observer.

Test if a pose's bounding box or position falls outside of an eye's view frustum.

Poses can be assigned bounding boxes which enclose any 3D models associated with them. A model is not visible if all the corners of the bounding box fall outside the viewing frustum. Therefore any primitives (i.e. triangles) associated with the pose can be culled during rendering to reduce CPU/GPU workload.

Returns

True if the object's bounding box is visible.

Return type

bool

Examples

You can avoid running draw commands if the object is not visible by doing a visibility test first:

```
if myStim.isVisible():
    myStim.draw()
```

property lineColor

Alternative way of setting *borderColor*.

property lineColorSpace

Deprecated, please use colorSpace to set color space for the entire object

property lineRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type

DEPRECATED

lineWidth

property ori

Orientation quaternion (X, Y, Z, W).

property pos

Position vector (X, Y, Z).

setAnchor(*value*, *log=None*)

setBackColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setBackRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setBackgroundColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setBorderColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setBorderRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setBorderWidth(*newWidth*, *operation=''*, *log=None*)

setColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setContrast(*newContrast*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setDKL(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setFillColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for fillColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setFillRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setFontColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setForeColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for foreColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setForeRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setLMS(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setLineColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

setLineRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setLineWidth(*newWidth*, *operation=''*, *log=None*)

setOri(*ori*)

setOriAxisAngle(*axis*, *angle*, *degrees=True*)

Set the orientation of the 3D stimulus using an *axis* and *angle*. This sets the quaternion at *ori*.

Parameters

- **axis** (*array_like*) – Axis of rotation [rx, ry, rz].
- **angle** (*float*) – Angle of rotation.
- **degrees** (*bool, optional*) – Specify True if *angle* is in degrees, or else it will be treated as radians. Default is True.

setPos(*pos*)

setRGB(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

property size

property thePose

The pose of the rigid body. This is a class which has *pos* and *ori* attributes.

units

None, 'norm', 'cm', 'deg', 'degFlat', 'degFlatPos', or 'pix'

If None then the current units of the *Window* will be used. See *Units for the window and stimuli* for explanation of other options.

Note that when you change units, you don't change the stimulus parameters and it is likely to change appearance. Example:

```
# This stimulus is 20% wide and 50% tall with respect to window
stim = visual.PatchStim(win, units='norm', size=(0.2, 0.5)

# This stimulus is 0.2 degrees wide and 0.5 degrees tall.
stim.units = 'deg'
```

updateColors()

Placeholder method to update colours when set externally, for example updating the *pallette* attribute of a textbox

property vertices

property width

property win

The *Window* object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1  # stimulus will be drawn in win1
stim.draw()  # stimulus is now drawn to win1
stim.win = win2  # stimulus will be drawn in win2
stim.draw()  # it is now drawn in win2
win1.flip(waitBlanking=False)  # do not wait for next
             # monitor update
win2.flip()  # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

11.4.33 TextBox

Warning

TextBox is deprecated. Please use TextBox2 instead which supports similar editable high-performance rendering of text but also supports non-monospaced fonts and a wider range of formatting and alignment options. This is a lazy-imported class, therefore import using full path *from psychopy.visual.textbox import TextBox* when inheriting from it.

Attributes

TextBox([window, text, font_name, bold, ...])	Similar to the visual.TextStim component, TextBox can be used to display text within a psychopy window.

The following `set______()` attributes all have equivalent `get______()` attributes:

TextBox.setText(text_source)	Set the text to be displayed within the Textbox.
TextBox.setPosition(pos)	Set the (x,y) position of the TextBox on the Monitor.
TextBox.setHorzAlign(v)	Specify how the horizontal (x) component of the TextBox position is to be interpreted.
TextBox.setVertAlign(v)	Specify how the vertical (y) component of the TextBox position is to be interpreted.
TextBox.setHorzJust(v)	Specify how text within the TextBox should be aligned horizontally.
TextBox.setVertJust(v)	Specify how text within the TextBox should be aligned vertically.
TextBox.setFontColor(c)	Set the color to use when drawing text glyphs within the TextBox.
TextBox.setBorderColor(c)	Set the color to use for the border of the textBox.
TextBox.setBackgroundColor(c)	Set the fill color used to fill the rectangular area of the TextBox stim.
TextBox.setTextGridLineColor(c)	Set the color used when drawing text grid lines.
TextBox.setTextGridLineWidth(c)	Set the stroke width (in pixels) to use for the text grid character bounding boxes.
TextBox.setInterpolated(interpolate)	Specify whether interpolation should be enabled for the TextBox when it is drawn.
TextBox.setOpacity(o)	Sets the TextBox transparency level to use for color related attributes of the Textbox.
TextBox.setAutoLog(v)	Specify if changes to textBox attribute values should be logged automatically by PsychoPy.
TextBox.draw()	Draws the TextBox to the back buffer of the graphics card.

TextBox also provides the following read-only functions:

TextBox.getSize()	Return the width,height of the TextBox, using the unit type being used by the stimulus.
TextBox.getName()	Same as the GetLabel method.
TextBox.getDisplayedText()	Return the text that fits within the TextBox and therefore is actually seen. This is equal to::.
TextBox.getValidStrokeWidths()	Returns the stroke width range supported by the graphics card being used.
TextBox.getLineSpacing()	Return the additional spacing being applied between rows of text.
TextBox.getGlyphPositionForTextIndex(char_ind	For the provided char_index, which is the index of one character in
TextBox.getTextGridCellPlacement()	Returns a 3D numpy array containing position information for each text grid cell in the TextBox.

Helper Functions

getFontManager()

FontManager provides a simple API for finding and loading font files (.ttf) via the FreeType library.

The FontManager finds supported font files on the computer and initially creates a dictionary containing the information about available fonts. This can be used to quickly determine what font family names are available on the computer and what styles (bold, italic) are supported for each family.

This font information can then be used to create the resources necessary to display text using a given font family, style, size, color, and dpi.

The *FontManager* is currently used by the psychopy.visual.TextBox stim type. A user script can access the *FontManager* via:

```
font_mngr=visual.textbox.getFontManager()
```

Once a font of a given size and dpi has been created; it is cached by the *FontManager* and can be used by all *TextBox* instances created within the experiment.

Details

class psychopy.visual.**TextBox**(*window=None*, *text='Default Test Text.'*, *font_name=None*, *bold=False*, *italic=False*, *font_size=32*, *font_color=(0, 0, 0, 1)*, *dpi=72*, *line_spacing=0*, *line_spacing_units='pix'*, *background_color=None*, *border_color=None*, *border_stroke_width=1*, *size=None*, *textgrid_shape=None*, *pos=(0.0, 0.0)*, *align_horz='center'*, *align_vert='center'*, *units='norm'*, *grid_color=None*, *grid_stroke_width=1*, *color_space='rgb'*, *opacity=1.0*, *grid_horz_justification='left'*, *grid_vert_justification='top'*, *autoLog=True*, *interpolate=False*, *name=None*)

Similar to the visual.TextStim component, TextBox can be used to display text within a psychopy window. TextBox and TextStim each have different strengths and weaknesses. You should select the most appropriate text component type based on how it will be used within the experiment.

NOTE: As of PsychoPy 1.79, TextBox should be considered experimental. The two TextBox demo scripts provided have been tested on all PsychoPy supported OS's and run without exceptions. However there are very likely bugs in the existing TextBox code and the TextBox API will be further enhanced and improved (i.e. changed) over the next couple months.

TextBox Features

- Text character placement is very well defined, useful when the exact positioning of each letter needs to be known.
- The text string that is displayed can be changed (setText()) and drawn (win.draw()) **very** quickly. See the TextBox vs. TextStim comparison table for details.
- Built-in font manager; providing easy access to the font family names and styles that are available on the computer being used.
- TextBox is a composite stimulus type, with the following graphical elements, many of which can be changed to control many aspects of how the TextBox is displayed.:
  - TextBox Border / Outline
  - TextBox Fill Area
  - Text Grid Cell Lines
  - Text Glyphs
- When using 'rgb' or 'rgb255' color spaces, colors can be specified as a list/tuple of 3 elements (red, green, blue), or with four elements (reg, green, blue, alpha) which allows different elements of the TextBox to use different opacity settings if desired. For colors that include the alpha channel value, it will be applied instead of the opacity setting of the TextBox, effectively overriding the stimulus defined opacity for that part of the textbox graphics. Colors that do not include an alpha channel use the opacity setting as normal.
- Text Line Spacing can be controlled.

Textbox Limitations

- Only Monospace Fonts are supported.
- TextBox component is not a completely **standard** psychopy visual stim and has the following functional difference:
  - TextBox attributes are never accessed directly; get* and set* methods are always used (this will be changed to use class properties in the future).
  - Setting an attribute of a TextBox only supports value replacement, (textbox.setFontColor([1.0,1.0,1.0])) and does not support specifying operators.
- Some key word arguments supported by other stimulus types in general, or by TextStim itself, are not supported by TextBox. See the TextBox class definition for the arguments that are supported.
- When a new font, style, and size are used it takes about 1 second to load and process the font. This is a one time delay for a given font name, style, and size. After first being loaded, the same font style can be used or re-applied to multiple TextBox components with no significant delay.
- Auto logging or auto drawing is not currently supported.

TextStim and TextBox Comparison:

Feature	TextBox	TextStim
Change text + redraw time^	1.513 msec	28.537 msec
No change + redraw time^	0.240 msec	0.931 msec
Initial Creation time^	0.927 msec	0.194 msec
MonoSpace Font Support	Yes	Yes
Non MonoSpace Font Support	No	Yes
Adjustable Line Spacing	Yes	No
Precise Text Pos. Info	Yes	No
Auto logging Support	No	Yes
Rotation Support	No	Yes
Word Wrapping Support	Yes	Yes

^ Times are in msec.usec format. Tested using the textstim_vs_textbox.py

demo script provided with the PsychoPy distribution. Results are dependent on text length, video card, and OS. Displayed results are based on 120 character string with an average of 24 words. Test computer used Windows 7 64 bit, PsychoPy 1.79, with a i7 3.4 Ghz CPU, 8 GB RAM, and NVIDIA 480 GTX 2GB graphics card.

Example:

```
from psychopy import visual

win=visual.Window(...)

# A Textbox stim that will look similar to a TextStim component

textstimlike=visual.TextBox(
    window=win,
    text="This textbox looks most like a textstim.",
    font_size=18,
    font_color=[-1,-1,1],
    color_space='rgb',
    size=(1.8,.1),
    pos=(0.0,.5),
    units='norm')

# A Textbox stim that uses more of the supported graphical features
#
textboxloaded=visual.TextBox(
    window=win
    text='TextBox showing all supported graphical elements',
    font_size=32,
    font_color=[1,1,1],
    border_color=[-1,-1,1], # draw a blue border around stim
    border_stroke_width=4, # border width of 4 pix.
    background_color=[-1,-1,-1], # fill the stim background
    grid_color=[1,-1,-1,0.5], # draw a red line around each
                              # possible letter area,
                              # 50% transparent
    grid_stroke_width=1,  # with a width of 1 pix
    textgrid_shape=[20,2],  # specify area of text box
```

(continues on next page)

(continued from previous page)

```
                            # by the number of cols x
                            # number of rows of text to support
                            # instead of by a screen
                            # units width x height.
    pos=(0.0,-.5),
    # If the text string length < num rows * num cols in
    # textgrid_shape, how should text be justified?
    #
    grid_horz_justification='center',
    grid_vert_justification='center')

textstimlike.draw()
textboxloaded.draw()
win.flip()
```

draw()

Draws the TextBox to the back buffer of the graphics card. Then call win.flip() to display the changes drawn. If draw() is not called prior to a call to win.flip(), the textBox will not be displayed for that retrace.

getAutoLog()

Indicates if changes to textBox attribute values should be logged automatically by PsychoPy. *Currently not supported by TextBox.*

getBackgroundColor()

Get the color used to fill the rectangular area of the TextBox stim. All other graphical elements of the TextBox are drawn on top of the background.

getBorderColor()

A border can be drawn around the perimeter of the TextBox. This method sets the color of that border.

getBorderWidth()

Get the stroke width of the optional TextBox area outline. This is always given in pixel units.

getColorSpace()

Returns the psychopy color space used when specifying colors for the TextBox. Supported values are:

- 'rgb'
- 'rbg255'
- 'norm'
- hex (implicit)
- html name (implicit)

See the Color Space section of the PsychoPy docs for details.

getDisplayedText()

Return the text that fits within the TextBox and therefore is actually seen. This is equal to:

```
text_length=len(self.getText())
cols,rows=self.getTextGridShape()

displayed_text=self.getText()[0:min(text_length,rows*cols]
```

`getFontColor()`

Return the color used when drawing text glyphs.

`getFontSize()`

`getGlyphPositionForTextIndex(`*char_index*`)`

For the provided char_index, which is the index of one character in

the current text being displayed by the TextBox (getDisplayedText()), return the bounding box position, width, and height for the associated glyph drawn to the screen. This factors in the glyphs position within the textgrid cell it is being drawn in, so the returned bounding box is for the actual glyph itself, not the textgrid cell. For textgrid cell placement information, see the getTextGridCellPlacement() method.

The glyph position for the given text index is returned as a tuple (x,y,width,height), where x,y is the top left hand corner of the bounding box.

Special Cases:

- If the index provided is out of bounds for the currently displayed text, None is returned.
- For u' ' (space) characters, the full textgrid cell bounding box is returned.
- For u'

' (new line) characters,the textgrid cell bounding box

is returned, but with the box width set to 0.

`getHorzAlign()`

Return what textbox x position should be interpreted as. Valid options are 'left', 'center', or 'right' .

`getHorzJust()`

Return how text should laid out horizontally when the number of columns of each text grid row is greater than the number needed to display the text for that text row.

`getInterpolated()`

Returns whether interpolation is enabled for the TextBox when it is drawn. When True, GL_LINE_SMOOTH and GL_POLYGON_SMOOTH are enabled within OpenGL; otherwise they are disabled.

`getLabel()`

Return the label / name assigned to the textbox. This does not impact how the stimulus looks when drawn, and instead is used for internal purposes only.

`getLineSpacing()`

Return the additional spacing being applied between rows of text. The value is in units specified by the textbox getUnits() method.

`getName()`

Same as the GetLabel method.

`getOpacity()`

Get the default TextBox transparency level used for color related attributes. 0.0 equals fully transparent, 1.0 equals fully opaque.

`getPosition()`

Return the x,y position of the textbox, in getUnitType() coord space.

`getSize()`

Return the width,height of the TextBox, using the unit type being used by the stimulus.

getText()

Return the text to display.

getTextGridCellForCharIndex(*char_index*)

getTextGridCellPlacement()

Returns a 3D numpy array containing position information for each text grid cell in the TextBox. The array has the shape (*num_cols*, *num_rows*, *cell_bounds*), where num_cols is the number of *textgrid* columns in the TextBox. *num_rows* is the number of *textgrid* rows in the *TextBox*. *cell_bounds* is a 4 element array containing the (x pos, y pos, width, height) data for the given cell. Position fields are for the top left hand corner of the cell box. Column and Row indices start at 0.

To get the shape of the textgrid in terms of columns and rows, use:

```
cell_pos_array=textbox.getTextGridCellPlacement()
col_row_count=cell_pos_array.shape[:2]
```

To access the position, width, and height for textgrid cell at column 0 and row 0 (so the top left cell in the textgrid):

```
cell00=cell_pos_array[0,0,:]
```

For the cell at col 3, row 1 (so 4th cell on second row):

```
cell41=cell_pos_array[4,1,:]
```

getTextGridLineColor()

Return the color used when drawing the outline of the text grid cells. Each letter displayed in a TextBox populates one of the text cells defined by the shape of the TextBox text grid. Color value must be valid for the color space being used by the TextBox.

A value of None indicates drawing of the textgrid lines is disabled.

getTextGridLineWidth()

Return the stroke width (in pixels) of the optional lines drawn around the text grid cell areas.

getUnitType()

Returns which of the psychopy coordinate systems are used by the TextBox. Position and size related attributes mush be specified relative to the unit type being used. Valid options are:

- pix
- norm
- cm

getValidStrokeWidths()

Returns the stroke width range supported by the graphics card being used. If the TextBox is Interpolated, a tuple is returns using float values, with the following structure:

((min_line_width, max_line_width), line_width_granularity)

If Interpolation is disabled for the TextBox, the returned tuple elements are int values, with the following structure:

(min_line_width, max_line_width)

getVertAlign()

Return what textbox y position should be interpreted as. Valid options are 'top', 'center', or 'bottom' .

`getVertJust()`

Return how text should laid out vertically when the number of text grid rows is greater than the number needed to display the current text

`getWindow()`

Returns the psychopy window that the textBox is associated with.

`setAutoLog(v)`

Specify if changes to textBox attribute values should be logged automatically by PsychoPy. True enables auto logging; False disables it. *Currently not supported by TextBox.*

`setBackgroundColor(c)`

Set the fill color used to fill the rectangular area of the TextBox stim. Color value must be valid for the color space being used by the TextBox.

A value of None will disable drawing of the TextBox background.

`setBorderColor(c)`

Set the color to use for the border of the textBox. The TextBox border is a rectangular outline drawn around the edges of the TextBox stim. Color value must be valid for the color space being used by the TextBox.

A value of None will disable drawing of the border.

`setBorderWidth(c)`

Set the stroke width (in pixels) to use for the border of the TextBox stim. Border values must be within the range of stroke widths supported by the OpenGL driver used by the graphics. Setting the width outside the valid range will result in the stroke width being clamped to the nearest end of the valid range.

Use the TextBox.getValidStrokeWidths() to access the minimum - maximum range of valid line widths.

`setFontColor(c)`

Set the color to use when drawing text glyphs within the TextBox. Color value must be valid for the color space being used by the TextBox. For 'rgb', 'rgb255', and 'norm' based colors, three or four element lists are valid. Three element colors use the TextBox getOpacity() value to determine the alpha channel for the color. Four element colors use the value of the fourth element to set the alpha value for the color.

`setHorzAlign(v)`

Specify how the horizontal (x) component of the TextBox position is to be interpreted. left = x position is the left edge, right = x position is the right edge x position, and center = the x position is used to center the stim horizontally.

`setHorzJust(v)`

Specify how text within the TextBox should be aligned horizontally. For example, if a text grid has 10 columns, and the text being displayed is 6 characters in length, the horizontal justification determines if the text should be draw starting at the left of the text columns (left), or should be centered on the columns ('center', in this example there would be two empty text cells to the left and right of the text.), or should be drawn such that the last letter of text is drawn in the last column of the text row ('right').

`setInterpolated(interpolate)`

Specify whether interpolation should be enabled for the TextBox when it is drawn. When interpolate == True, GL_LINE_SMOOTH and GL_POLYGON_SMOOTH are enabled within OpenGL. When interpolate is set to False, GL_POLYGON_SMOOTH and GL_LINE_SMOOTH are disabled.

`setOpacity(o)`

Sets the TextBox transparency level to use for color related attributes of the Textbox. 0.0 equals fully transparent, 1.0 equals fully opaque.

If opacity is set to None, it is assumed to have a default value of 1.0.

When a color is defined with a 4th element in the colors element list, then this opacity value is ignored and the alpha value provided in the color itself is used for that TextGrid element instead.

`setPosition`(*pos*)

Set the (x,y) position of the TextBox on the Monitor. The position must be given using the unit coord type used by the stim.

The TextBox position is interpreted differently depending on the Horizontal and Vertical Alignment settings of the stim. See getHorzAlignment() and getVertAlignment() for more information.

For example, if the TextBox alignment is specified as left, top, then the position specifies the top left hand corner of where the stim will be drawn. An alignment of bottom,right indicates that the position value will define where the bottom right corner of the TextBox will be drawn. A horz., vert. alignment of center, center will place the center of the TextBox at pos.

`setText`(*text_source*)

Set the text to be displayed within the Textbox.

Note that once a TextBox has been created, the number of character rows and columns is static. To change the size of a TextBox, a new TextBox stim must be created to replace the current Textbox stim. Therefore ensure that the textbox is large enough to display the largest length string to be presented in the TextBox. Characters that do not fit within the TextBox will not be displayed.

Color value must be valid for the color space being used by the TextBox.

`setTextGridLineColor`(*c*)

Set the color used when drawing text grid lines. These are lines that can be drawn which mark the bounding box for each character within the TextBox text grid. Color value must be valid for the color space being used by the TextBox.

Provide a value of None to disable drawing of textgrid lines.

`setTextGridLineWidth`(*c*)

Set the stroke width (in pixels) to use for the text grid character bounding boxes. Border values must be within the range of stroke widths supported by the OpenGL driver used by the computer graphics card. Setting the width outside the valid range will result in the stroke width being clamped to the nearest end of the valid range.

Use the TextBox.getGLineRanges() to access a dict containing some OpenGL parameters which provide the minimum, maximum, and resolution of valid line widths.

`setVertAlign`(*v*)

Specify how the vertical (y) component of the TextBox position is to be interpreted. top = y position is the top edge, bottom = y position is the bottom edge y position, and center = the y position is used to center the stim vertically.

`setVertJust`(*v*)

Specify how text within the TextBox should be aligned vertically. For example, if a text grid has 3 rows for text, and the text being displayed all fits on one row, the vertical justification determines if the text should be draw on the top row of the text grid (top), or should be centered on the rows (‘center’, in this example there would be one row above and below the row used to draw the text), or should be drawn on the last row of the text grid, (‘bottom’).

11.4.34 TextBox2

Attributes

TextBox2(win, text[, font, pos, units, ...])	

The following `set______()` attributes all have equivalent `get______()` attributes:

TextBox2.text	
TextBox2.alignment	
TextBox2.hasFocus	
TextBox2.overlaps(polygon[, tight])	Returns *True* if this stimulus intersects another one.
TextBox2.contains(x[, y, units, tight])	Returns True if a point x,y is inside the stimulus' border.
TextBox2.clear()	Resets the TextBox2 to a blank string
TextBox2.reset()	Resets the TextBox2 to hold **whatever it was given on initialisation**
TextBox2.font	
TextBox2.height	
TextBox2.anchor	
TextBox2.editable	Determines whether or not the TextBox2 instance can receive typed text
TextBox2.padding	
TextBox2.size	The (requested) size of the TextBox (w,h) in whatever units the stimulus is using
TextBox2.pos	The position of the center of the TextBox in the stimulus *units*
TextBox2.units	
TextBox2.draw()	Draw the text to the back buffer

Details

class psychopy.visual.textbox2.**TextBox2**(*win*, *text*, *font='Noto Sans'*, *pos=(0, 0)*, *units=None*, *letterHeight=None*, *ori=0*, *size=None*, *color=(1.0, 1.0, 1.0)*, *colorSpace='rgb'*, *fillColor=None*, *fillColorSpace=None*, *borderWidth=2*, *borderColor=None*, *borderColorSpace=None*, *contrast=1*, *opacity=None*, *bold=False*, *italic=False*, *placeholder='Type here...'*, *lineSpacing=1.0*, *letterSpacing=None*, *padding=None*, *speechPoint=None*, *anchor='center'*, *alignment='left'*, *flipHoriz=False*, *flipVert=False*, *languageStyle='LTR'*, *editable=False*, *overflow='visible'*, *lineBreaking='default'*, *draggable=False*, *name=''*, *autoLog=None*, *autoDraw=False*, *depth=0*, *onTextCallback=None*, *clickable=True*)

Parameters

- **win** (Window) – The window this stimulus is associated with.
- **text** (*str*) – The text to display in the TextBox.
- **font**
- **pos**
- **units**
- **letterHeight**
- **size** (*Specifying None gets the default size for this type of unit.*) – Specifying [None, None] gets a TextBox that's expandable in both dimensions. Specifying [0.75, None] gets a textbox that expands in the length but fixed at 0.75 units in the width
- **color**
- **colorSpace**
- **contrast**
- **opacity**
- **bold**
- **italic**
- **lineSpacing**
- **padding**
- **speechPoint** (*list, tuple, np.ndarray or None*) – Location of the end of a speech bubble tail on the textbox, in the same units as this textbox. If the point sits within the textbox, the tail will be inverted. Use *None* for no tail.
- **anchor**
- **alignment**
- **fillColor**
- **borderWidth**
- **borderColor**
- **flipHoriz**
- **flipVert**
- **editable**
- **lineBreaking** (*Specifying 'default', text will be broken at a set of*) – characters defined in the module. Specifying 'uax14', text will be broken in accordance with UAX#14 (Unicode Line Breaking Algorithm).
- **draggable** (*bool*) – Can this stimulus be dragged by a mouse click?
- **name**
- **autoLog**

property RGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type

DEPRECATED

_addRenderOnlyChar(*i*, *x*, *y*, *vertices*, *glyph*, *alphaCorrection=1*)

Add a character at index i which is drawn but not actually part of the text

_calcPosRendered()

DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix.

_calcSizeRendered()

DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix

_drawLegacyGL()

Legacy draw routine for older GL versions.

_getDesiredRGB(*rgb*, *colorSpace*, *contrast*)

Convert color to RGB while adding contrast. Requires self.rgb, self.colorSpace and self.contrast

_getPolyAsRendered()

DEPRECATED. Return a list of vertices as rendered.

_layout()

Layout the text, calculating the vertex locations

_onCursorKeys(*key*)

Called by the window when cursor/del/backspace… are received

_onMouse()

Called by the window when the mouse is inside the stimulus.

_onText(*chr*)

Called by the window when characters are received

_selectWindow(*win*)

Switch drawing to the specified window. Calls the window's _setCurrent() method which handles the switch.

_set(*attrib*, *val*, *op=''*, *log=None*)

DEPRECATED since 1.80.04 + 1. Use setAttribute() and val2array() instead.

_updateList()

The user shouldn't need this method since it gets called after every call to .set() Chooses between using and not using shaders each call.

_updateVertices()

Sets Stim.verticesPix and ._borderPix from pos, size, ori, flipVert, flipHoriz

addCharAtCaret(*char*)

Allows a character to be added programmatically at the current caret

property alignment

alphaThreshold

Threshold for alpha values.

If the alpha value of a pixel is below this threshold, the pixel will be rejected (not drawn). This can be useful for creating a mask from an image with an alpha channel. The default value is 0.0, which means that no thresholding will be applied.

property anchor

`autoDraw`

Determines whether the stimulus should be automatically drawn on every frame flip.

Value should be: *True* or *False*. You do NOT need to set this on every frame flip!

`autoLog`

Whether every change in this stimulus should be auto logged.

Value should be: *True* or *False*. Set to *False* if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren't likely to be useful.

`property backColor`

Alternative way of setting fillColor

`property backColorSpace`

Deprecated, please use colorSpace to set color space for the entire object.

`property backRGB`

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type
DEPRECATED

`property backgroundColor`

Alternative way of setting fillColor

`property borderColor`

`property borderColorSpace`

Deprecated, please use colorSpace to set color space for the entire object

`property borderRGB`

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type
DEPRECATED

`borderWidth`

`clear()`

Resets the TextBox2 to a blank string

`clickable`

Whether the stimulus can be clicked on.

If set to *True*, the stimulus will be checked for mouse clicks and the *_onMouse* method will be called if the stimulus is clicked.

`property color`

Alternative way of setting *foreColor*.

`property colorSpace`

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                       color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

`contains`(*x*, *y=None*, *units=None*, *tight=False*)

Returns True if a point x,y is inside the stimulus' border.

Can accept variety of input options:

- two separate args, x and y
- one arg (list, tuple or array) containing two vals (x,y)
- **an object with a getPos() method that returns x,y, such**
  as a `Mouse`.

Returns *True* if the point is within the area defined either by its *border* attribute (if one defined), or its *vertices* attribute if there is no .border. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the *contains* method; the extent of the stimulus is determined purely by the size, position (pos), and orientation (ori) settings (and by the vertices for shape stimuli).

See Coder demos: shapeContains.py See Coder demos: shapeContains.py

`containsPointer`()

Check if the mouse is within the stimulus boundaries.

Returns
Whether the mouse is within the stimulus.

Return type
bool

`property contrast`

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged).
Operations supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast =  1.0  # unchanged contrast
stim.contrast =  0.5  # decrease contrast
stim.contrast =  0.0  # uniform, no contrast
stim.contrast = -0.5  # slightly inverted
stim.contrast = -1.0  # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast =  1.2  # increases contrast
stim.contrast = -1.2  # inverts with increased contrast
```

deleteCaretLeft()

Deletes 1 character to the left of the caret

deleteCaretRight()

Deletes 1 character to the right of the caret

depth

DEPRECATED, depth is now controlled simply by drawing order.

doDragging()

If this stimulus is draggable, do the necessary actions on a frame flip to drag it.

doPointerActions()

Handle mouse interaction with the stimulus.

This method should be called each frame to update the stimulus based on mouse interactions.

draggable

Can this stimulus be dragged by a mouse click?

draw()

Draw the text to the back buffer

property editable

Determines whether or not the TextBox2 instance can receive typed text

property fillColor

Set the fill color for the shape.

property fillColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property fillRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type

DEPRECATED

property flip

1x2 array for flipping vertices along each axis; set as True to flip or False to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (*._flip*) will give an array of 1s and -1s with which to multiply vertices.

property flipHoriz

property flipVert

font

property fontColor

Alternative way of setting *foreColor*.

property fontMGR

property foreColor

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names <http://www.w3schools.com/html/html_colornames.asp>* can be used.
- *Colors by hex value*
- **numerically: (scalar or triplet) for DKL, RGB or**
  other *Color spaces*. For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue'  # (the case is actually ignored)
stim.color = '#DDA0DD'  # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0]  # if stim.colorSpace='rgb':
                # a red color in rgb space
stim.color = [0.0, 45.0, 1.0]  # if stim.colorSpace='dkl':
                # DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255]  # if stim.colorSpace='rgb255':
                # a blue stimulus using rgb255 space
stim.color = 255  # interpreted as (255, 255, 255)
                  # which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1]  # increment all guns by 1 value
stim.color *= -1  # multiply the color by -1 (which in this
                    # space inverts the contrast)
stim.color *= [0.5, 0, 1]  # decrease red, remove green, keep blue
```

You can use *setColor* if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

property foreColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property foreRGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type
DEPRECATED

getAlignment()

getAlphaThreshold()

getAnchor()

getAutoDraw()

getAutoLog()

getBackColor()

getBackColorSpace()

getBackRGB()

getBackgroundColor()

getBorderColor()

getBorderColorSpace()

getBorderRGB()

getBorderWidth()

getCharAtPos(*pos*)

Get the character index at a given position.

This can be used to determine what character is under the specified position in the stimulus.

Parameters

pos (*list*, *tuple*) – Position in stimulus units.

Returns

Index of character at the given position. Returns None if no character is at the given position or if the position is outside the stimulus bounds.

Return type

int or None

getClickable()

getColor()

getColorSpace()

getContrast()

getDepth()

getDraggable()

getEditable()

getFillColor()

getFillColorSpace()

getFillRGB()

getFlip()

getFlipHoriz()

getFlipVert()

getFont()

getFontColor()

getFontMGR()

getForeColor()

getForeColorSpace()

getForeRGB()

getHasFocus()

getHeight()

getLanguageStyle()

getLetterHeight()

getLetterHeightPix()

getLetterSpacing()

getLineColor()

getLineColorSpace()

getLineRGB()

getLineWidth()

getName()

getOpacity()

getOri()

getOverflow()

getPadding()

getPalette()

getPallette()

getPlaceholder()

getPos()

getRGB()

getSize()

getSpeechPoint()

`getText()`

Returns the current text in the box, including formatting tokens.

`getUnits()`

`getVertices()`

`getVerticesPix()`

`getVisibleText()`

Returns the current visible text in the box

`getWidth()`

`getWin()`

`get_borderPix()`

`property hasFocus`

`property height`

`isDragging = False`

`property languageStyle`

How is text laid out? Left to right (LTR), right to left (RTL) or using Arabic layout rules?

`property letterHeight`

`property letterHeightPix`

Convenience function to get self._letterHeight.pix and be guaranteed a return that is a single integer

`letterSpacing`

Distance between letters, relative to the current font's default. Set as None or 1 to use font default unchanged.

`property lineColor`

Alternative way of setting *borderColor*.

`property lineColorSpace`

Deprecated, please use colorSpace to set color space for the entire object

`property lineRGB`

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type
DEPRECATED

`lineWidth`

`name`

The name (*str*) of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = None your stimulus will be called "unnamed <type>", e.g. visual.TextStim(win) will be called "unnamed TextStim" in the logs.

`property opacity`

Determines how visible the stimulus is relative to background.

The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). *Operations* are supported. Precisely how this is used depends on the *Blend Mode*.

ori

overflow

overlaps(*polygon*, *tight=False*)

Returns *True* if this stimulus intersects another one.

If *polygon* is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the *overlaps* method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

Parameters

See coder demo, shapeContains.py

property padding

property palette

Describes the current visual properties of the TextBox in a dict

property pallette

Disambiguation for palette.

placeholder

Text to display when textbox is editable and has no content.

property pos

The position of the center of the TextBox in the stimulus *units*

value should be an *x,y-pair*. *Operations* are also supported.

Example:

```
stim.pos = (0.5, 0)  # Set slightly to the right of center
stim.pos += (0.5, -1)  # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2  # Move stim towards the center.
    Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:

myTextbox._pos.pix

reset()

Resets the TextBox2 to hold **whatever it was given on initialisation**

setAlignment(*value*, *log=None*, *operation=False*, *stealth=False*)

setAlphaThreshold(*value*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setAnchor(*value*, *log=None*)

`setAutoDraw`(*value*, *log=None*)

Sets autoDraw. Usually you can use 'stim.attribute = value' syntax instead, but use this method to suppress the log message.

`setAutoLog`(*value=True*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

`setBackColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

`setBackColorSpace`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setBackRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

`setBackgroundColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

`setBorderColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

`setBorderColorSpace`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setBorderRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

`setBorderWidth`(*newWidth*, *operation=''*, *log=None*)

`setClickable`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

`setColorSpace`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setContrast`(*newContrast*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

`setDKL`(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

`setDepth`(*newDepth*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

`setDraggable`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setEditable`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setFillColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for fillColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

`setFillColorSpace`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setFillRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

`setFlip`(*value*, *log=None*, *operation=False*, *stealth=False*) `setFlipHoriz`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setFlipVert`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setFont`(*font*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

`setFontColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

`setFontMGR`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setForeColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for foreColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

`setForeColorSpace`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setForeRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

`setHasFocus`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setHeight`(*height*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

`setLMS`(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

`setLanguageStyle`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setLetterHeight`(*value*, *log=None*)

`setLetterSpacing`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setLineColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

`setLineColorSpace`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setLineRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

`setLineWidth`(*newWidth*, *operation=''*, *log=None*)

`setName`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setOpacity`(*newOpacity*, *operation=''*, *log=None*)

Hard setter for opacity, allows the suppression of log messages and calls the update method

`setOri`(*newOri*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

`setOverflow`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setPadding`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setPalette`(*value*, *log=None*, *operation=False*, *stealth=False*) `setPallette`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setPlaceholder`(*value*, *log=False*)

Set text to display when textbox is editable and has no content.

`setPos`(*newPos*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

`setRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

`setSize`(*newSize*, *operation=''*, *units=None*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

`setSpeechPoint`(*value*, *log=None*)

`setText`(*text=None*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

`setUnits`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setVertices`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setWidth`(*value*, *log=None*, *operation=False*, *stealth=False*)

`setWin`(*value*, *log=None*, *operation=False*, *stealth=False*)

`property size`

The (requested) size of the TextBox (w,h) in whatever units the stimulus is using

This determines the outer extent of the area.

If the width is set to None then the text will continue extending and not wrap. If the height is set to None then the text will continue to grow downwards as needed.

`speechPoint`

`property text`

`property units`

`updateColors`()

Placeholder method to update colours when set externally, for example updating the *pallette* attribute of a textbox

`updateOpacity`()

Placeholder method to update colours when set externally, for example updating the *pallette* attribute of a textbox.

`property vertices`

`property verticesPix`

This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

`property visibleText`

Returns the current visible text in the box

property width

property win

The *Window* object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1  # stimulus will be drawn in win1
stim.draw()  # stimulus is now drawn to win1
stim.win = win2  # stimulus will be drawn in win2
stim.draw()  # it is now drawn in win2
win1.flip(waitBlanking=False)  # do not wait for next
             # monitor update
win2.flip()  # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

11.4.35 TextStim

class psychopy.visual.**TextStim**(*win*, *text='Hello World'*, *font=''*, *pos=(0.0, 0.0)*, *depth=0*, *rgb=None*, *color=(1.0, 1.0, 1.0)*, *colorSpace='rgb'*, *opacity=1.0*, *contrast=1.0*, *units=''*, *ori=0.0*, *height=None*, *antialias=True*, *bold=False*, *italic=False*, *alignHoriz=None*, *alignVert=None*, *alignText='center'*, *anchorHoriz='center'*, *anchorVert='center'*, *fontFiles=()*, *wrapWidth=None*, *flipHoriz=False*, *flipVert=False*, *languageStyle='LTR'*, *draggable=False*, *name=None*, *autoLog=None*, *autoDraw=False*)

Class of text stimuli to be displayed in a *Window*

Performance OBS: in general, TextStim is slower than many other visual stimuli, i.e. it takes longer to change some attributes. In general, it's the attributes that affect the shapes of the letters: `text`, `height`, `font`, `bold` etc. These make the next .draw() slower because that sets the text again. You can make the draw() quick by calling re-setting the text (`myTextStim.text = myTextStim.text`) when you've changed the parameters.

In general, other attributes which merely affect the presentation of unchanged shapes are as fast as usual. This includes `pos`, `opacity` etc.

The following attribute can only be set at initialization (see further down for a list of attributes which can be changed after initialization):

languageStyle

Apply settings to correctly display content from some languages that are written right-to-left. Currently there are three (case- insensitive) values for this parameter:

- **`'LTR'` is the default, for typical left-to-right, Latin-style**
  languages.
- **`'RTL'` will correctly display text in right-to-left languages**
  such as Hebrew. By applying the bidirectional algorithm, it allows mixing portions of left-to-right content (such as numbers or Latin script) within the string.
- **`'Arabic'` applies the bidirectional algorithm but additionally**
  will _reshape_ Arabic characters so they appear in the cursive, linked form that depends on neigh-

bouring characters, rather than in their isolated form. May also be applied in other scripts, such as Farsi or Urdu, that use Arabic-style alphabets.

Parameters

`property RGB`

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type
DEPRECATED

`_calcPosRendered()`

DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix.

`_calcSizeRendered()`

DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix

`_getDesiredRGB`(*rgb*, *colorSpace*, *contrast*)

Convert color to RGB while adding contrast. Requires self.rgb, self.colorSpace and self.contrast

`_getPolyAsRendered()`

DEPRECATED. Return a list of vertices as rendered.

`_selectWindow`(*win*)

Switch drawing to the specified window. Calls the window's _setCurrent() method which handles the switch.

`_set`(*attrib*, *val*, *op=''*, *log=None*)

DEPRECATED since 1.80.04 + 1. Use setAttribute() and val2array() instead.

`_setTextShaders`(*value=None*)

Set the text to be rendered using the current font

`_updateList()`

The user shouldn't need this method since it gets called after every call to .set() Chooses between using and not using shaders each call.

`_updateListShaders()`

Only used with pygame text - pyglet handles all from the draw()

`_updateVertices()`

Sets Stim.verticesPix and ._borderPix from pos, size, ori, flipVert, flipHoriz

`alignHoriz`

Deprecated in PsychoPy 3.3. Use *alignText* and *anchorHoriz* instead

`alignText`

Aligns the text content within the bounding box ('left', 'right' or 'center') See also *anchorX* to set alignment of the box itself relative to pos

`alignVert`

Deprecated in PsychoPy 3.3. Use *anchorVert*

`alphaThreshold`

Threshold for alpha values.

If the alpha value of a pixel is below this threshold, the pixel will be rejected (not drawn). This can be useful for creating a mask from an image with an alpha channel. The default value is 0.0, which means that no thresholding will be applied.

`anchorHoriz`

The horizontal alignment ('left', 'right' or 'center')

`anchorVert`

The vertical alignment ('top', 'bottom' or 'center') of the box relative to the text *pos*.

`antialias`

Allow antialiasing the text (True or False). Sets text, slow.

`autoDraw`

Determines whether the stimulus should be automatically drawn on every frame flip.

Value should be: *True* or *False*. You do NOT need to set this on every frame flip!

`autoLog`

Whether every change in this stimulus should be auto logged.

Value should be: *True* or *False*. Set to *False* if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren't likely to be useful.

`bold`

Make the text bold (True, False) (better to use a bold font name).

`property boundingBox`

(read only) attribute representing the bounding box of the text (w,h). This differs from *width* in that the width represents the width of the margins, which might differ from the width of the text within them.

NOTE: currently always returns the size in pixels (this will change to return in stimulus units)

`property color`

Alternative way of setting *foreColor*.

`property colorSpace`

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                       color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

`contains`(*x*, *y=None*, *units=None*)

Returns True if a point x,y is inside the stimulus' border.

Can accept variety of input options:

- two separate args, x and y

- one arg (list, tuple or array) containing two vals (x,y)
- **an object with a getPos() method that returns x,y, such**
  as a `Mouse`.

Returns *True* if the point is within the area defined either by its *border* attribute (if one defined), or its *vertices* attribute if there is no .border. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the *contains* method; the extent of the stimulus is determined purely by the size, position (pos), and orientation (ori) settings (and by the vertices for shape stimuli).

See Coder demos: shapeContains.py See Coder demos: shapeContains.py

`property contrast`

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged).
Operations supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast =  1.0  # unchanged contrast
stim.contrast =  0.5  # decrease contrast
stim.contrast =  0.0  # uniform, no contrast
stim.contrast = -0.5  # slightly inverted
stim.contrast = -1.0  # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast =  1.2  # increases contrast
stim.contrast = -1.2  # inverts with increased contrast
```

`depth`

DEPRECATED, depth is now controlled simply by drawing order.

`doDragging()`

If this stimulus is draggable, do the necessary actions on a frame flip to drag it.

`draggable`

Can this stimulus be dragged by a mouse click?

`draw(`*win=None*`)`

Draw the stimulus in its relevant window. You must call this method after every MyWin.flip() if you want the stimulus to appear on that frame and then update the screen again.

If win is specified then override the normal window of this stimulus.

`property flip`

1x2 array for flipping vertices along each axis; set as True to flip or False to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (*._flip*) will give an array of 1s and -1s with which to multiply vertices.

flipHoriz

If set to True then the text will be flipped left-to-right. The flip is relative to the original, not relative to the current state.

flipVert

If set to True then the text will be flipped top-to-bottom. The flip is relative to the original, not relative to the current state.

font

String. Set the font to be used for text rendering. font should be a string specifying the name of the font (in system resources).

property fontColor

Alternative way of setting *foreColor*.

fontFiles

A list of additional files if the font is not in the standard system location (include the full path).

OBS: fonts are added every time this value is set. Previous are not deleted.

E.g.:

```
stim.fontFiles = ['SpringRage.ttf']  # load file(s)
stim.font = 'SpringRage'  # set to font
```

property foreColor

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names <http://www.w3schools.com/html/html_colornames.asp>* can be used.
- *Colors by hex value*
- **numerically: (scalar or triplet) for DKL, RGB or**
  other *Color spaces*. For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue'  # (the case is actually ignored)
stim.color = '#DDA0DD'  # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0]  # if stim.colorSpace='rgb':
                # a red color in rgb space
stim.color = [0.0, 45.0, 1.0]  # if stim.colorSpace='dkl':
                # DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255]  # if stim.colorSpace='rgb255':
                # a blue stimulus using rgb255 space
stim.color = 255  # interpreted as (255, 255, 255)
                  # which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1]  # increment all guns by 1 value
stim.color *= -1  # multiply the color by -1 (which in this
                    # space inverts the contrast)
stim.color *= [0.5, 0, 1]  # decrease red, remove green, keep blue
```

You can use *setColor* if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

property foreColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property foreRGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type

DEPRECATED

height

The height of the letters (Float/int or None = set default).

Height includes the entire box that surrounds the letters in the font. The width of the letters is then defined by the font.

Operations supported.

italic

True/False. Make the text italic (better to use a italic font name).

name

The name (*str*) of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = None your stimulus will be called “unnamed <type>”, e.g. visual.TextStim(win) will be called “unnamed TextStim” in the logs.

property opacity

Determines how visible the stimulus is relative to background.

The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). *Operations* are supported. Precisely how this is used depends on the *Blend Mode*.

ori

The orientation of the stimulus (in degrees).

Should be a single value (*scalar*). *Operations* are supported.

Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and below zero values wrap appropriately.

overlaps(*polygon*)

Returns *True* if this stimulus intersects another one.

If *polygon* is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the *overlaps* method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

See coder demo, shapeContains.py

property pos

The position of the center of the stimulus in the stimulus *units*

value should be an *x,y-pair*. *Operations* are also supported.

Example:

```
stim.pos = (0.5, 0)  # Set slightly to the right of center
stim.pos += (0.5, -1)  # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2  # Move stim towards the center.
    Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:

```
from psychopy.tools.monitorunittools import posToPix
posPix = posToPix(stim)
```

property posPix

This determines the coordinates in pixels of the position for the current stimulus, accounting for pos and units. This property should automatically update if *pos* is changed

setAlphaThreshold(*value*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setAutoDraw(*value*, *log=None*)

Sets autoDraw. Usually you can use 'stim.attribute = value' syntax instead, but use this method to suppress the log message.

setAutoLog(*value=True*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setContrast(*newContrast*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setDKL(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setDepth(*newDepth*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setFlip(*direction*, *log=None*)

(used by Builder to simplify the dialog)

setFlipHoriz(*newVal=True*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

`setFlipVert`(*newVal=True*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

`setFont`(*font*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

`setForeColor`(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Hard setter for foreColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

`setForeRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

`setHeight`(*height*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

`setLMS`(*color*, *operation=''*)

DEPRECATED since v1.60.05: Please use the *color* attribute

`setLetterHeight`(*height*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

`setOpacity`(*newOpacity*, *operation=''*, *log=None*)

Hard setter for opacity, allows the suppression of log messages and calls the update method

`setOri`(*newOri*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

`setPos`(*newPos*, *operation=''*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

`setRGB`(*color*, *operation=''*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

`setSize`(*newSize*, *operation=''*, *units=None*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

`setText`(*text=None*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

`property size`

The size (width, height) of the stimulus in the stimulus *units*

Value should be *x,y-pair*, *scalar* (applies to both dimensions) or None (resets to default). *Operations* are supported.

Sizes can be negative (causing a mirror-image reversal) and can extend beyond the window.

Example:

```
stim.size = 0.8  # Set size to (xsize, ysize) = (0.8, 0.8)
print(stim.size)  # Outputs array([0.8, 0.8])
stim.size += (0.5, -0.5)  # make wider and flatter: (1.3, 0.3)
```

Tip: if you can see the actual pixel range this corresponds to by looking at *stim._sizeRendered*

text

The text to be rendered. Use \n to make new lines.

Issues: May be slow, and pyglet has a memory leak when setting text. For these reasons, this function checks so that it only updates the text if it has changed. So scripts can safely set the text on every frame, with no need to check if it has actually altered.

updateColors()

Placeholder method to update colours when set externally, for example updating the *pallette* attribute of a textbox

updateOpacity()

Placeholder method to update colours when set externally, for example updating the *pallette* attribute of a textbox.

property verticesPix

This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

property win

The `Window` object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1  # stimulus will be drawn in win1
stim.draw()  # stimulus is now drawn to win1
stim.win = win2  # stimulus will be drawn in win2
stim.draw()  # it is now drawn in win2
win1.flip(waitBlanking=False)  # do not wait for next
             # monitor update
win2.flip()  # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

wrapWidth

Int/float or None (set default). The width the text should run before wrapping.

Operations supported.

11.4.36 psychopy.visual.VisualSystemHD

Classes for using NordicNeuralLab's VisualSystemHD in-scanner display for presenting visual stimuli. Support is preliminary so users must empirically verify whether the default settings for barrel distortion and FOV are correct. Support may be good enough at this point for studies that do not require precise stereoscopy or stimulus sizes.

Overview

VisualSystemHD([monoscopic, diopters, ...])	Class provides support for NordicNeuralLab's VisualSystemHD(tm) fMRI display hardware.
VisualSystemHD.monoscopic	*True* if using monoscopic mode.
VisualSystemHD.lensCorrection	*True* if using lens correction.
VisualSystemHD.distCoef	Distortion coefficient (*float*).
VisualSystemHD.diopters	Diopters value of the current eye buffer.
VisualSystemHD.setDiopters(diopters[, eye])	Set the diopters for a given eye.
VisualSystemHD.eyeOffset	Eye offset for the current buffer in centimeters used for stereoscopic rendering.
VisualSystemHD.setEyeOffset(dist[, eye])	Set the eye offset in centimeters.
VisualSystemHD.setBuffer(buffer[, clear])	Set the eye buffer to draw to.
VisualSystemHD.setPerspectiveView([...])	Set the projection and view matrix to render with perspective.

Details

class psychopy.visual.nnlvs.**VisualSystemHD**(*monoscopic=False*, *diopters=(-1, -1)*, *lensCorrection=True*, *distCoef=None*, *directDraw=False*, *model='vshd'*, **args*, ***kwargs*)

Class provides support for NordicNeuralLab's VisualSystemHD(tm) fMRI display hardware. This is a lazy-imported class, therefore import using full path *from psychopy.visual.nnlvs import VisualSystemHD* when inheriting from it.

Use this class in-place of the *Window* class for use with the VSHD hardware. Ensure that the VSHD headset display output is configured in extended desktop mode (eg. nVidia Surround). Extended desktops are only supported on Windows and Linux systems.

The VSHD is capable of both 2D and stereoscopic 3D rendering. You can select which eye to draw to by calling *setBuffer*, much like how stereoscopic rendering is implemented in the base *Window* class.

Notes

- This class handles drawing differently than the default window class, as a result, stimuli *autoDraw* is not supported.
- Edges of the warped image may appear jagged. To correct this, create a window using *multiSample=True* and *numSamples > 1* to smooth out these artifacts.

Examples

Here is a basic example of 2D rendering using the VisualSystemHD(tm). This is the binocular version of the dynamic 'plaid.py' demo:

```python
from psychopy import visual, core, event

# Create a visual window
win = visual.VisualSystemHD(fullscr=True, screen=1)

# Initialize some stimuli, note contrast, opacity, ori
grating1 = visual.GratingStim(win, mask="circle", color='white',
    contrast=0.5, size=(1.0, 1.0), sf=(4, 0), ori = 45, autoLog=False)
grating2 = visual.GratingStim(win, mask="circle", color='white',
```

(continues on next page)

(continued from previous page)

```
    opacity=0.5, size=(1.0, 1.0), sf=(4, 0), ori = -45, autoLog=False,
    pos=(0.1, 0.1))

trialClock = core.Clock()
t = 0
while not event.getKeys() and t < 20:
    t = trialClock.getTime()

    for eye in ('left', 'right'):
        win.setBuffer(eye)  # change the buffer
        grating1.phase = 1 * t  # drift at 1Hz
        grating1.draw()  # redraw it
        grating2.phase = 2 * t    # drift at 2Hz
        grating2.draw()  # redraw it

    win.flip()

win.close()
core.quit()
```

As you can see above, there are few changes needed to convert an existing 2D experiment to run on the VSHD. For 3D rendering with perspective, you need set *eyeOffset* and apply the projection by calling *setPerspectiveView*. (other projection modes are not implemented or supported right now):

```
from psychopy import visual, core, event

# Create a visual window
win = visual.VisualSystemHD(fullscr=True, screen=1,
    multiSample=True, nSamples=8)

# text to display
instr = visual.TextStim(win, text="Any key to quit", pos=(0, -.7))

# create scene light at the pivot point
win.lights = [
    visual.LightSource(win, pos=(0.4, 4.0, -2.0), lightType='point',
                diffuseColor=(0, 0, 0), specularColor=(1, 1, 1))
]
win.ambientLight = (0.2, 0.2, 0.2)

# Initialize some stimuli, note contrast, opacity, ori
ball = visual.SphereStim(win, radius=0.1, pos=(0, 0, -2), color='green',
    useShaders=False)

iod = 6.2  # interocular separation in CM
win.setEyeOffset(-iod / 2.0, 'left')
win.setEyeOffset(iod / 2.0, 'right')

trialClock = core.Clock()
t = 0
while not event.getKeys() and t < 20:
    t = trialClock.getTime()
```

(continues on next page)

(continued from previous page)

```
    for eye in ('left', 'right'):
        win.setBuffer(eye)  # change the buffer

        # setup drawing with perspective
        win.setPerspectiveView()

        win.useLights = True  # switch on lights
        ball.draw()  # draw the ball
        # shut the lights, needed to prevent light color from affecting
        # 2D stim
        win.useLights = False

        # reset transform to draw text correctly
        win.resetEyeTransform()

        instr.draw()

    win.flip()

win.close()
core.quit()
```

Parameters

- **monoscopic** (*bool*) – Use monoscopic rendering. If *True*, the same image will be drawn to both eye buffers. You will not need to call *setBuffer*. It is not possible to set monoscopic mode after the window is created. It is recommended that you use monoscopic mode if you intend to display only 2D stimuli about the center of the display as it uses a less memory intensive rendering pipeline.
- **diopters** (*tuple or list*) – Initial diopter values for the left and right eye. Default is *(-1, -1)*, values must be integers.
- **lensCorrection** (*bool*) – Apply lens correction (barrel distortion) to the output. The amount of distortion applied can be specified using *distCoef*. If *False*, no distortion will be applied to the output and the entire display will be used. Not applying correction will result in pincushion distortion which produces a non-rectilinear output.
- **distCoef** (*float*) – Distortion coefficient for barrel distortion. If *None*, the recommended value will be used for the model of display. You can adjust the value to fine-tune the barrel distortion.
- **directDraw** (*bool*) – Direct drawing mode. Stimuli are drawn directly to the back buffer instead of creating separate buffer. This saves video memory but does not permit barrel distortion or monoscopic rendering. If *False*, drawing is done with two FBOs containing each eye's image.
- **hwModel** (*str*) – Model of the VisualSystemHD in use. Used to set viewing parameters accordingly. Default is 'vshd'. Cannot be changed after starting the application.

USE_LEGACY_GL = True

_assignFlipTime(*obj*, *attrib*, *format=<class 'float'>*)

Helper function to assign the time of last flip to the obj.attrib

Parameters

- **obj** (*dict or object*) – A mutable object (usually a dict of class instance).
- **attrib** (*str*) – Key or attribute of `obj` to assign the flip time to.
- **format** (*str, class or None*) – Format in which to return time, see clock.Timestamp.resolve() for more info. Defaults to *float*.

_blitEyeBuffer(*eye*)

Warp and blit to the appropriate eye buffer.

Parameters

eye (*str*) – Eye buffer being used.

_checkMatchingSizes(*requested*, *actual*)

Checks whether the requested and actual screen sizes differ. If not then a warning is output and the window size is set to actual

_cleanEditables()

Make sure there are no dead refs in the editables list

_clearDepthBuffer()

Clear the depth buffer.

_endOfFlip(*clearBuffer*)

Override end of flip with custom color channel masking if required.

_getFrame(*rect=None*, *buffer='front'*)

Return the current Window as an image.

_getPixels(*rect=None*, *buffer='front'*, *includeAlpha=True*, *makeLum=False*)

Return an array of pixel values from the current window buffer or sub-region.

Parameters

- **rect** (*tuple[int], optional*) – The region of the window to capture in pixel coordinates (left, bottom, width, height). If *None*, the whole window is captured.
- **buffer** (*str, optional*) – Buffer to capture.
- **includeAlpha** (*bool, optional*) – Include the alpha channel in the returned array. Default is *True*.
- **makeLum** (*bool, optional*) – Convert the RGB values to luminance values. Values are rounded to the nearest integer. Default is *False*.

Returns

Pixel values as a 3D array of shape (height, width, channels). If *includeAlpha* is *False*, the array will have shape (height, width, 3). If *makeLum* is *True*, the array will have shape (height, width).

Return type

ndarray

Examples

Get the pixel values of the whole window:

```
pix = win._getPixels()
```

Get pixel values and convert to luminance and get average:

```
pix = win._getPixels(makeLum=True)
average = pix.mean()
```

`_getRegionOfFrame`(*rect=(-1, 1, 1, -1)*, *buffer='front'*, *power2=False*, *squarePower2=False*)

Deprecated function, here for historical reasons. You may now use *:py:attr:`~Window._getFrame()* and specify a rect to get a sub-region, just as used here.

power2 can be useful with older OpenGL versions to avoid interpolation in `PatchStim`. If power2 or squarePower2, it will expand rect dimensions up to next power of two. squarePower2 uses the max dimensions. You need to check what your hardware & OpenGL supports, and call `_getRegionOfFrame()` as appropriate.

`_getWarpExtents`(*eye*)

Get the horizontal and vertical extents of the barrel distortion in normalized device coordinates. This is used to determine the FOV along each axis after barrel distortion.

Parameters
eye (*str*) – Eye to compute the extents for.

Returns
2d array of coordinates [+X, -X, +Y, -Y] of the extents of the barrel distortion.

Return type
ndarray

`_renderFBO`()

Perform a warp operation.

(in this case a copy operation without any warping)

`_setCurrent`()

Make this window's OpenGL context current.

If called on a window whose context is current, the function will return immediately. This reduces the number of redundant calls if no context switch is required. If `useFBO=True`, the framebuffer is bound after the context switch.

`_setupEyeBuffers`()

Setup additional buffers for rendering content to each eye.

`_setupFrameBuffer`()

Setup the framebuffer object for this window.

Returns
True if the framebuffer was successfully setup, *False* otherwise. If *False*, the framebuffer was not complete. Make sure that your driver supports the necessary formats.

Return type
bool

`_setupGL`()

Setup OpenGL state for this window.

`_setupGamma`(*gammaVal*)

A private method to work out how to handle gamma for this Window given that the user might have specified an explicit value, or maybe gave a Monitor.

`_setupLensCorrection`()

Setup the VAOs needed for lens correction.

_startOfFlip()

Custom `_startOfFlip` for HMD rendering. This finalizes the HMD texture before diverting drawing operations back to the on-screen window. This allows `flip` to swap the on-screen and HMD buffers when called. This function always returns *True*.

Return type
True

_updateProjectionMatrix()

Update the default projection matrix based on the current window settings.

_updateViewMatrix()

Update the default orthographic view matrix based on the current window settings.

addEditable(*editable*)

Adds an editable element to the screen (something to which characters can be sent with meaning from the keyboard).

The current editable object receiving chars is Window.currentEditable

Parameters
editable

Returns

property ambientLight

Ambient light color for the scene [r, g, b, a]. Values range from 0.0 to 1.0. Only applicable if *useLights* is *True*.

Examples

Setting the ambient light color:

```
win.ambientLight = [0.5, 0.5, 0.5]

# don't do this!!!
win.ambientLight[0] = 0.5
win.ambientLight[1] = 0.5
win.ambientLight[2] = 0.5
```

applyEyeTransform(*clearDepth=True*)

Apply the current view and projection matrices.

Matrices specified by attributes `viewMatrix` and `projectionMatrix` are applied using 'immediate mode' OpenGL functions. Subsequent drawing operations will be affected until `flip()` is called.

All transformations in `GL_PROJECTION` and `GL_MODELVIEW` matrix stacks will be cleared (set to identity) prior to applying. After this is called, the current matrix mode will be set to `GL_MODELVIEW`.

Parameters
clearDepth (*bool*) – Clear the depth buffer. This may be required prior to rendering 3D objects.

Examples

Using a custom view and projection matrix:

```
# Must be called every frame since these values are reset after
# `flip()` is called!
win.viewMatrix = viewtools.lookAt( ... )
win.projectionMatrix = viewtools.perspectiveProjectionMatrix( ... )
win.applyEyeTransform()
# draw 3D objects here ...
```

property aspect

Aspect ratio of the current viewport (width / height).

backgroundFit

How should the background image of this window fit? Options are:

None, “None”, “none”
No scaling is applied, image is present at its pixel size unaltered.

“cover”
Image is scaled such that it covers the whole screen without changing its aspect ratio. In other words, both dimensions are evenly scaled such that its SHORTEST dimension matches the window’s LONGEST dimension.

“contain”
Image is scaled such that it is contained within the screen without changing its aspect ratio. In other words, both dimensions are evenly scaled such that its LONGEST dimension matches the window’s SHORTEST dimension.

“scaleDown”, “scale-down”, “scaledown”
If image is bigger than the window along any dimension, it will behave as if backgroundFit were “contain”. Otherwise, it will behave as if backgroundFit were None.

backgroundImage

Background image for the window, can be either a visual.ImageStim object or anything which could be passed to visual.ImageStim.image to create one. Will be drawn each time *win.flip()* is called, meaning it is always below all other contents of the window.

blendMode

Blend mode to use.

callOnFlip(*function*, **args*, ***kwargs*)

Call a function immediately after the next `flip()` command.

The first argument should be the function to call, the following args should be used exactly as you would for your normal call to the function (can use ordered arguments or keyword arguments as normal).

e.g. If you have a function that you would normally call like this:

```
pingMyDevice(portToPing, channel=2, level=0)
```

then you could call `callOnFlip()` to have the function call synchronized with the frame flip like this:

```
win.callOnFlip(pingMyDevice, portToPing, channel=2, level=0)
```

clearAutoDraw()

Remove all autoDraw components, meaning they get autoDraw set to False and are not added to any list (as in .stashAutoDraw)

clearBuffer(*color=True*, *depth=False*, *stencil=False*)

Clear the present buffer (to which you are currently drawing) without flipping the window.

Useful if you want to generate movie sequences from the back buffer without actually taking the time to flip the window.

Set *color* prior to clearing to set the color to clear the color buffer to. By default, the depth buffer is cleared to a value of 1.0.

Parameters

- **color** (*bool*) – Buffers to clear.
- **depth** (*bool*) – Buffers to clear.
- **stencil** (*bool*) – Buffers to clear.

Examples

Clear the color buffer to a specified color:

```
win.color = (1, 0, 0)
win.clearBuffer(color=True)
```

Clear only the depth buffer, *depthMask* must be *True* or else this will have no effect. Depth mask is usually *True* by default, but may change:

```
win.depthMask = True
win.clearBuffer(color=False, depth=True, stencil=False)
```

close()

Close the window (and reset the Bits++ if necess).

property color

Set the color of the window.

This command sets the color that the blank screen will have on the next clear operation. As a result it effectively takes TWO `flip()` operations to become visible (the first uses the color to create the new screen, the second presents that screen to the viewer). For this reason, if you want to changed background color of the window "on the fly", it might be a better idea to draw a `Rect` that fills the whole window with the desired `Rect.fillColor` attribute. That'll show up on first flip.

See other stimuli (e.g. `GratingStim.color`) for more info on the color attribute which essentially works the same on all PsychoPy stimuli.

See *Color spaces* for further information about the ways to specify colors and their various implications.

property colorSpace

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                       color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

property contentScaleFactor

Scaling factor (*float*) to use when drawing to the backbuffer to convert framebuffer to client coordinates.

> **See also**
>
> *getContentScaleFactor*

property convergeOffset

Convergence offset from monitor in centimeters.

This is value corresponds to the offset from screen plane to set the convergence plane (or point for *toe-in* projections). Positive offsets move the plane farther away from the viewer, while negative offsets nearer. This value is used by *setPerspectiveView* and should be set before calling it to take effect.

Notes

- This value is only applicable for *setToeIn* and *setOffAxisView*.

coordToRay(*screenXY*)

Convert a screen coordinate to a direction vector.

Takes a screen/window coordinate and computes a vector which projects a ray from the viewpoint through it (line-of-sight). Any 3D point touching the ray will appear at the screen coordinate.

Uses the current *viewport* and *projectionMatrix* to calculate the vector. The vector is in eye-space, where the origin of the scene is centered at the viewpoint and the forward direction aligned with the -Z axis. A ray of (0, 0, -1) results from a point at the very center of the screen assuming symmetric frustums.

Note that if you are using a flipped/mirrored view, you must invert your supplied screen coordinates (*screenXY*) prior to passing them to this function.

Parameters

screenXY (*array_like*) – X, Y screen coordinate. Must be in units of the window.

Returns

Normalized direction vector [x, y, z].

Return type

ndarray

Examples

Getting the direction vector between the mouse cursor and the eye:

```
mx, my = mouse.getPos()
dir = win.coordToRay((mx, my))
```

Set the position of a 3D stimulus object using the mouse, constrained to a plane. The object origin will always be at the screen coordinate of the mouse cursor:

```
# the eye position in the scene is defined by a rigid body pose
win.viewMatrix = camera.getViewMatrix()
win.applyEyeTransform()

# get the mouse location and calculate the intercept
mx, my = mouse.getPos()
ray = win.coordToRay([mx, my])
result = intersectRayPlane(   # from mathtools
    orig=camera.pos,
    dir=camera.transformNormal(ray),
    planeOrig=(0, 0, -10),
    planeNormal=(0, 1, 0))

# if result is `None`, there is no intercept
if result is not None:
    pos, dist = result
    objModel.thePose.pos = pos
else:
    objModel.thePose.pos = (0, 0, -10)  # plane origin
```

If you don't define the position of the viewer with a *RigidBodyPose*, you can obtain the appropriate eye position and rotate the ray by doing the following:

```
pos = numpy.linalg.inv(win.viewMatrix)[:3, 3]
ray = win.coordToRay([mx, my]).dot(win.viewMatrix[:3, :3])
# then ...
result = intersectRayPlane(
    orig=pos,
    dir=ray,
    planeOrig=(0, 0, -10),
    planeNormal=(0, 1, 0))
```

property cullFace

True if face culling is enabled.`

property cullFaceMode

Face culling mode, either *back*, *front* or *both*.

property currentEditable

The editable (Text?) object that currently has key focus

property depthFunc

Depth test comparison function for rendering.

property depthMask

True if depth masking is enabled. Writing to the depth buffer will be disabled.

property depthTest

True if depth testing is enabled.

property diopters

Diopters value of the current eye buffer.

`classmethod dispatchAllWindowEvents()`

Dispatches events for all pyglet windows. Used by iohub 2.0 psychopy kb event integration.

`property distCoef`

Distortion coefficient (*float*).

`property draw3d`

True if 3D drawing is enabled on this window.

`property eyeOffset`

Eye offset for the current buffer in centimeters used for stereoscopic rendering. This works differently than the main window class as it sets the offset for the current buffer. The offset is saved and automatically restored when the buffer is selected.

`property farClip`

Distance to the far clipping plane in meters.

`flip`(*clearBuffer=True*)

Flip the front and back buffers after drawing everything for your frame. (This replaces the `update()` method, better reflecting what is happening underneath).

Parameters

`clearBuffer` (*bool, optional*) – Clear the draw buffer after flipping. Default is *True*.

Returns

Wall-clock time in seconds the flip completed. Returns *None* if `waitBlanking` is *False*.

Return type

float or None

Notes

- The time returned when `waitBlanking` is *True* corresponds to when the graphics driver releases the draw buffer to accept draw commands again. This time is usually close to the vertical sync signal of the display.

Examples

Results in a clear screen after flipping:

```
win.flip(clearBuffer=True)
```

The screen is not cleared (so represent the previous screen):

```
win.flip(clearBuffer=False)
```

`fps()`

Report the frames per second since the last call to this function (or since the window was created if this is first call)

`property frameBufferSize`

Size of the framebuffer in pixels (w, h).

`property frontFace`

Face winding order to define front, either *ccw* or *cw*.

`property fullscr`

Return whether the window is in fullscreen mode.

`gamma`

Set the monitor gamma for linearization.

> ⚠ **Warning**
>
> Don't use this if using a Bits++ or Bits#, as it overrides monitor settings.

`gammaRamp`

Sets the hardware CLUT using a specified 3xN array of floats ranging between 0.0 and 1.0.

Array must have a number of rows equal to 2 ^ max(bpc).

`getActualFrameRate`(*nIdentical=10*, *nMaxFrames=100*, *nWarmUpFrames=10*, *threshold=1*, *infoMsg=None*)

Measures the actual frames-per-second (FPS) for the screen.

This is done by waiting (for a max of *nMaxFrames*) until *nIdentical* frames in a row have identical frame times (std dev below *threshold* ms).

Parameters

- **nIdentical** (*int, optional*) – The number of consecutive frames that will be evaluated. Higher –> greater precision. Lower –> faster.
- **nMaxFrames** (*int, optional*) – The maximum number of frames to wait for a matching set of nIdentical.
- **nWarmUpFrames** (*int, optional*) – The number of frames to display before starting the test (this is in place to allow the system to settle after opening the *Window* for the first time.
- **threshold** (*int or float, optional*) – The threshold for the std deviation (in ms) before the set are considered a match.

Returns

Frame rate (FPS) in seconds. If there is no such sequence of identical frames a warning is logged and *None* will be returned.

Return type

float or None

`getContentScaleFactor`()

Get the scaling factor required for scaling correctly on high-DPI displays.

If the returned value is 1.0, no scaling needs to be applied to objects drawn on the backbuffer. A value >1.0 indicates that the backbuffer is larger than the reported client area, requiring points to be scaled to maintain constant size across similarly sized displays. In other words, the scaling required to convert framebuffer to client coordinates.

Returns

Scaling factor to be applied along both horizontal and vertical dimensions.

Return type

float

Examples

Get the size of the client area:

```
clientSize = win.frameBufferSize / win.getContentScaleFactor()
```

Get the framebuffer size from the client size:

```
frameBufferSize = win.clientSize * win.getContentScaleFactor()
```

Convert client (window) to framebuffer pixel coordinates (eg., a mouse coordinate, vertices, etc.):

```
# `mousePosXY` is an array ...
frameBufferXY = mousePosXY * win.getContentScaleFactor()
# you can also use the attribute ...
frameBufferXY = mousePosXY * win.contentScaleFactor
```

Notes

- This value is only valid after the window has been fully realized.

getFutureFlipTime(*targetTime=0*, *clock=None*)

The expected time of the next screen refresh. This is currently calculated as win._lastFrameTime + refreshInterval

Parameters

- **targetTime** (*float*) – The delay *from now* for which you want the flip time. 0 will give the because that the earliest we can achieve. 0.15 will give the schedule flip time that gets as close to 150 ms as possible
- **clock** (*None, 'ptb', 'now' or any Clock object*) – If True then the time returned is compatible with ptb.GetSecs()
- **verbose** (*bool*) – Set to True to view the calculations along the way

getMovieFrame(*buffer='front'*)

Capture the current Window as an image.

Saves to stack for `saveMovieFrames()`. As of v1.81.00 this also returns the frame as a PIL image

This can be done at any time (usually after a `flip()` command).

Frames are stored in memory until a `saveMovieFrames()` command is issued. You can issue `getMovieFrame()` as often as you like and then save them all in one go when finished.

The back buffer will return the frame that hasn't yet been 'flipped' to be visible on screen but has the advantage that the mouse and any other overlapping windows won't get in the way.

The default front buffer is to be called immediately after a `flip()` and gives a complete copy of the screen at the window's coordinates.

Parameters

buffer (*str, optional*) – Buffer to capture.

Returns

Buffer pixel contents as a PIL/Pillow image object.

Return type

Image

getMsPerFrame(*nFrames=60*, *showVisual=False*, *msg=''*, *msDelay=0.0*)

Assesses the monitor refresh rate (average, median, SD) under current conditions, over at least 60 frames.

Records time for each refresh (frame) for n frames (at least 60), while displaying an optional visual. The visual is just eye-candy to show that something is happening when assessing many frames. You can also give it text to display instead of a visual, e.g., `msg='(testing refresh rate...)'`; setting msg implies `showVisual == False`.

To simulate refresh rate under cpu load, you can specify a time to wait within the loop prior to doing the `flip()`. If 0 < msDelay < 100, wait for that long in ms.

Returns timing stats (in ms) of:

- average time per frame, for all frames
- standard deviation of all frames
- median, as the average of 12 frame times around the median (~monitor refresh rate)

Author

- 2010 written by Jeremy Gray

hideMessage()

Remove any message that is currently being displayed.

hidePilotingIndicator()

Hide the visual indicator which shows we are in piloting mode.

property lensCorrection

True if using lens correction.

property lights

Scene lights.

This is specified as an array of *~psychopy.visual.LightSource* objects. If a single value is given, it will be converted to a *list* before setting. Set *useLights* to *True* before rendering to enable lighting/shading on subsequent objects. If *lights* is *None* or an empty *list*, no lights will be enabled if *useLights=True*, however, the scene ambient light set with *ambientLight* will be still be used.

Examples

Create a directional light source and add it to scene lights:

```
dirLight = gltools.LightSource((0., 1., 0.), lightType='directional')
win.lights = dirLight  # `win.lights` will be a list when accessed!
```

Multiple lights can be specified by passing values as a list:

```
myLights = [gltools.LightSource((0., 5., 0.)),
            gltools.LightSource((-2., -2., 0.))
win.lights = myLights
```

logOnFlip(*msg*, *level*, *obj=None*)

Send a log message that should be time-stamped at the next `flip()` command.

Parameters

- **msg** (*str*) – The message to be logged.

- **level** (*int*) – The level of importance for the message.
- **obj** (*object, optional*) – The python object that might be associated with this message if desired.

property monoscopic

True if using monoscopic mode.

property mouseVisible

Returns the visibility of the mouse cursor.

multiFlip(*flips=1*, *clearBuffer=True*)

Flip multiple times while maintaining the display constant. Use this method for precise timing.

WARNING: This function should not be used. See the *Notes* section for details.

Parameters

- **flips** (*int, optional*) – The number of monitor frames to flip. Floats will be rounded to integers, and a warning will be emitted. `Window.multiFlip(flips=1)` is equivalent to `Window.flip()`. Defaults to *1*.
- **clearBuffer** (*bool, optional*) – Whether to clear the screen after the last flip. Defaults to *True*.

Notes

- This function can behave unpredictably, and the PsychoPy authors recommend against using it. See https://github.com/psychopy/psychopy/issues/867 for more information.

Examples

Example of using `multiFlip`:

```
# Draws myStim1 to buffer
myStim1.draw()
# Show stimulus for 4 frames (90 ms at 60Hz)
myWin.multiFlip(clearBuffer=False, flips=6)
# Draw myStim2 "on top of" myStim1
# (because buffer was not cleared above)
myStim2.draw()
# Show this for 2 frames (30 ms at 60Hz)
myWin.multiFlip(flips=2)
# Show blank screen for 3 frames (buffer was cleared above)
myWin.multiFlip(flips=3)
```

property nearClip

Distance to the near clipping plane in meters.

nextEditable()

Moves focus of the cursor to the next editable window

onResize(*width*, *height*)

A default resize event handler.

This default handler updates the GL viewport to cover the entire window and sets the `GL_PROJECTION` matrix to be orthogonal in window space. The bottom-left corner is (0, 0) and the top-right corner is the width and height of the *Window* in pixels.

Override this event handler with your own to create another projection, for example in perspective.

property projectionMatrix

Projection matrix defined as a 4x4 numpy array.

recordFrameIntervals

Record time elapsed per frame.

Provides accurate measures of frame intervals to determine whether frames are being dropped. The intervals are the times between calls to `flip()`. Set to *True* only during the time-critical parts of the script. Set this to *False* while the screen is not being updated, i.e., during any slow, non-frame-time-critical sections of your code, including inter-trial-intervals, `event.waitkeys()`, `core.wait()`, or `image.setImage()`.

Examples

Enable frame interval recording, successive frame intervals will be stored:

```
win.recordFrameIntervals = True
```

Frame intervals can be saved by calling the `saveFrameIntervals` method:

```
win.saveFrameIntervals()
```

removeEditable(*editable*)

resetEyeTransform(*clearDepth=True*)

Restore the default projection and view settings to PsychoPy defaults. Call this prior to drawing 2D stimuli objects (i.e. GratingStim, ImageStim, Rect, etc.) if any eye transformations were applied for the stimuli to be drawn correctly.

Parameters

clearDepth (*bool*) – Clear the depth buffer upon reset. This ensures successive draw commands are not affected by previous data written to the depth buffer. Default is *True*.

Notes

- Calling `flip()` automatically resets the view and projection to defaults. So you don't need to call this unless you are mixing 3D and 2D stimuli.

Examples

Going between 3D and 2D stimuli:

```
# 2D stimuli can be drawn before setting a perspective projection
win.setPerspectiveView()
# draw 3D stimuli here ...
win.resetEyeTransform()
# 2D stimuli can be drawn here again ...
win.flip()
```

resetViewport()

Reset the viewport to cover the whole framebuffer.

Set the viewport to match the dimensions of the back buffer or framebuffer (if *useFBO=True*). The scissor rectangle is also set to match the dimensions of the viewport.

retrieveAutoDraw()

Add all stimuli which are on 'hold' back into the autoDraw list, and clear the hold list.

property rgb

saveFrameIntervals(*fileName=None*, *clear=True*)

Save recorded screen frame intervals to disk, as comma-separated values.

Parameters

- **fileName** (*None* or str) – *None* or the filename (including path if necessary) in which to store the data. If None then 'lastFrameIntervals.log' will be used.
- **clear** (*bool*) – Clear buffer frames intervals were stored after saving. Default is *True*.

saveMovieFrames(*fileName*, *codec='libx264'*, *fps=30*, *clearFrames=True*)

Writes any captured frames to disk.

Will write any format that is understood by PIL (tif, jpg, png, . . .)

Parameters

- **filename** (*str*) – Name of file, including path. The extension at the end of the file determines the type of file(s) created. If an image type (e.g. .png) is given, then multiple static frames are created. If it is .gif then an animated GIF image is created (although you will get higher quality GIF by saving PNG files and then combining them in dedicated image manipulation software, such as GIMP). On Windows and Linux *.mpeg* files can be created if *pymedia* is installed. On macOS *.mov* files can be created if the pyobjc-frameworks-QTKit is installed. Unfortunately the libs used for movie generation can be flaky and poor quality. As for animated GIFs, better results can be achieved by saving as individual .png frames and then combining them into a movie using software like ffmpeg.
- **codec** (*str, optional*) – The codec to be used **by moviepy** for mp4/mpg/mov files. If *None* then the default will depend on file extension. Can be one of `libx264`, `mpeg4` for mp4/mov files. Can be `rawvideo`, `png` for avi files (not recommended). Can be `libvorbis` for ogv files. Default is `libx264`.
- **fps** (*int, optional*) – The frame rate to be used throughout the movie. **Only for quicktime (.mov) movies.**. Default is *30*.
- **clearFrames** (*bool, optional*) – Set this to *False* if you want the frames to be kept for additional calls to `saveMovieFrames`. Default is *True*.

Examples

Writes a series of static frames as frame001.tif, frame002.tif etc.:

```
myWin.saveMovieFrames('frame.tif')
```

As of PsychoPy 1.84.1 the following are written with moviepy:

```
myWin.saveMovieFrames('stimuli.mp4') # codec = 'libx264' or 'mpeg4'
myWin.saveMovieFrames('stimuli.mov')
myWin.saveMovieFrames('stimuli.gif')
```

property scissor

Scissor rectangle (x, y, w, h) for the current draw buffer.

Values *x* and *y* define the origin, and *w* and *h* the size of the rectangle in pixels. The scissor operation is only active if *scissorTest=True*.

Usually, the scissor and viewport are set to the same rectangle to prevent drawing operations from *spilling* into other regions of the screen. For instance, calling *clearBuffer* will only clear within the scissor rectangle.

Setting the scissor rectangle but not the viewport will restrict drawing within the defined region (like a rectangular aperture), not changing the positions of stimuli.

property scissorTest

True if scissor testing is enabled.

property screenshot

setBlendMode(*blendMode*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setBuffer(*buffer*, *clear=True*)

Set the eye buffer to draw to. Subsequent draw calls will be diverted to the specified eye.

Parameters

- **buffer** (*str*) – Eye buffer to draw to. Values can either be 'left' or 'right'.
- **clear** (*bool*) – Clear the buffer prior to drawing.

setColor(*color*, *colorSpace=None*, *operation=''*, *log=None*)

Usually you can use `stim.attribute = value` syntax instead, but use this method if you want to set color and colorSpace simultaneously.

See `color` for documentation on colors.

setDiopters(*diopters*, *eye=None*)

Set the diopters for a given eye.

Parameters

- **diopters** (*int*) – Set diopters for a given eye, ranging between -7 and +5.
- **eye** (*str or None*) – Eye to set, either 'left' or 'right'. If *None*, the currently set buffer will be used.

setEyeOffset(*dist*, *eye=None*)

Set the eye offset in centimeters.

When set, successive rendering operations will use the new offset.

Parameters

- **dist** (*float or int*) – Lateral offset in centimeters from the nose, usually half the interocular separation. The distance is signed.
- **eye** (*str or None*) – Eye offset to set. Can either be 'left', 'right' or *None*. If *None*, the offset of the current buffer is used.

setGamma(*gamma*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setMouseType(*name='arrow'*)

Change the appearance of the cursor for this window. Cursor types provide contextual hints about how to interact with on-screen objects.

The graphics used 'standard cursors' provided by the operating system. They may vary in appearance and hot spot location across platforms. The following names are valid on most platforms:

- `arrow` : Default pointer.

- `ibeam` : Indicates text can be edited.
- `crosshair` : Crosshair with hot-spot at center.
- `hand` : A pointing hand.
- `hresize` : Double arrows pointing horizontally.
- `vresize` : Double arrows pointing vertically.

Parameters

name (*str*) – Type of standard cursor to use (see above). Default is `arrow`.

Notes

- On Windows the `crosshair` option is negated with the background color. It will not be visible when placed over 50% grey fields.

`setMouseVisible`(*visibility*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

`setOffAxisView`(*applyTransform=True*, *clearDepth=True*)

Set an off-axis projection.

Create an off-axis projection for subsequent rendering calls. Sets the *viewMatrix* and *projectionMatrix* accordingly so the scene origin is on the screen plane. If *eyeOffset* is correct and the view distance and screen size is defined in the monitor configuration, the resulting view will approximate *ortho-stereo* viewing.

The convergence plane can be adjusted by setting *convergeOffset*. By default, the convergence plane is set to the screen plane. Any points on the screen plane will have zero disparity.

Parameters

- **applyTransform** (*bool*) – Apply transformations after computing them in immediate mode. Same as calling `applyEyeTransform()` afterwards.
- **clearDepth** (*bool, optional*) – Clear the depth buffer.

`setOrthographicView`(*applyTransform=True*, *clearDepth=True*)

Set the projection and view matrix to render with orthographic view.

Orthographic projection is used to render 3D objects without perspective distortion. The scene origin is centered on the screen plane. The frustum is defined by the size of the window in pixels, with the origin at the center of the window. 2D stimuli are typically drawn using this projection.

Note that the values of `projectionMatrix` and `viewMatrix` will be replaced when calling this function.

Parameters

- **applyTransform** (*bool*) – Apply transformations after computing them in immediate mode. Same as calling `applyEyeTransform()` afterwards if *False*.
- **clearDepth** (*bool, optional*) – Clear the depth buffer.

`setPerspectiveView`(*applyTransform=True*, *clearDepth=True*)

Set the projection and view matrix to render with perspective.

Matrices are computed using values specified in the monitor configuration with the scene origin on the screen plane. Calculations assume units are in meters. If *eyeOffset != 0*, the view will be transformed laterally, however the frustum shape will remain the same.

Note that the values of `projectionMatrix` and `viewMatrix` will be replaced when calling this function.

Parameters

- **applyTransform** (*bool*) – Apply transformations after computing them in immediate mode. Same as calling `applyEyeTransform()` afterwards if *False*.
- **clearDepth** (*bool, optional*) – Clear the depth buffer.

`setRGB`(*newRGB*)

Deprecated: As of v1.61.00 please use *setColor()* instead

`setRecordFrameIntervals`(*value=True*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

`setScale`(*units*, *font='dummyFont'*, *prevScale=(1.0, 1.0)*)

DEPRECATED: this method used to be used to switch between units for stimulus drawing but this is now handled by the stimuli themselves and the window should always be left in units of 'pix'

`setToeInView`(*applyTransform=True*, *clearDepth=True*)

Set toe-in projection.

Create a toe-in projection for subsequent rendering calls. Sets the *viewMatrix* and *projectionMatrix* accordingly so the scene origin is on the screen plane. The value of *convergeOffset* will define the convergence point of the view, which is offset perpendicular to the center of the screen plane. Points falling on a vertical line at the convergence point will have zero disparity.

Parameters

- **applyTransform** (*bool*) – Apply transformations after computing them in immediate mode. Same as calling `applyEyeTransform()` afterwards.
- **clearDepth** (*bool, optional*) – Clear the depth buffer.

Notes

- This projection mode is only 'correct' if the viewer's eyes are converged at the convergence point. Due to perspective, this projection introduces vertical disparities which increase in magnitude with eccentricity. Use *setOffAxisView* if you want to display something the viewer can look around the screen comfortably.

`setUnits`(*value*, *log=True*)

`setViewOri`(*value*, *log=True*)

`setViewPos`(*value*, *log=True*)

`setViewScale`(*value*, *log=True*)

`showMessage`(*msg*)

Show a message in the window. This can be used to show information to the participant.

This creates a TextBox2 object that is displayed in the window. The text can be updated by calling this method again with a new message. The updated text will appear the next time *draw()* is called.

Parameters

msg (*str or None*) – Message text to display. If None, then any existing message is removed.

`showPilotingIndicator`()

Show the visual indicator which shows we are in piloting mode.

property size

Size of the drawable area in pixels (w, h).

stashAutoDraw()

Put autoDraw components on 'hold', meaning they get autoDraw set to False but are added to an internal list to be 'released' when .releaseAutoDraw is called.

property stencilTest

True if stencil testing is enabled.

timeOnFlip(*obj*, *attrib*, *format=<class 'float'>*)

Retrieves the time on the next flip and assigns it to the *attrib* for this *obj*.

Parameters

- **obj** (*dict or object*) – A mutable object (usually a dict of class instance).
- **attrib** (*str*) – Key or attribute of *obj* to assign the flip time to.
- **format** (*str, class or None*) – Format in which to return time, see clock.Timestamp.resolve() for more info. Defaults to *float*.

Examples

Assign time on flip to the `tStartRefresh` key of `myTimingDict`:

```
win.getTimeOnFlip(myTimingDict, 'tStartRefresh')
```

title

units

None, 'height' (of the window), 'norm', 'deg', 'cm', 'pix' Defines the default units of stimuli initialized in the window. I.e. if you change units, already initialized stimuli won't change their units.

Can be overridden by each stimulus, if units is specified on initialization.

See *Units for the window and stimuli* for explanation of options.

update()

Deprecated: use Window.flip() instead

updateLights(*index=None*)

Explicitly update scene lights if they were modified.

This is required if modifications to objects referenced in *lights* have been changed since assignment. If you removed or added items of *lights* you must refresh all of them.

Parameters

index (*int, optional*) – Index of light source in *lights* to update. If *None*, all lights will be refreshed.

Examples

Call *updateLights* if you modified lights directly like this:

```
win.lights[1].diffuseColor = [1., 0., 0.]
win.updateLights(1)
```

property useLights

Enable scene lighting.

Lights will be enabled if using legacy OpenGL lighting. Stimuli using shaders for lighting should check if *useLights* is *True* since this will have no effect on them, and disable or use a no lighting shader instead. Lights will be transformed to the current view matrix upon setting to *True*.

Lights are transformed by the present *GL_MODELVIEW* matrix. Setting *useLights* will result in their positions being transformed by it. If you want lights to appear at the specified positions in world space, make sure the current matrix defines the view/eye transformation when setting *useLights=True*.

This flag is reset to *False* at the beginning of each frame. Should be *False* if rendering 2D stimuli or else the colors will be incorrect.

property viewMatrix

View matrix defined as a 4x4 numpy array.

viewOri

Set the rotation of the view in degrees.

The rotation is applied around the origin of the window, which is defined by the viewPos attribute. The rotation is applied after scaling but before translation.

viewPos

The origin of the window onto which stimulus-objects are drawn.

The value should be given in the units defined for the window. NB: Never change a single component (x or y) of the origin, instead replace the viewPos-attribute in one shot, e.g.:

```
win.viewPos = [new_xval, new_yval]  # This is the way to do it
win.viewPos[0] = new_xval  # DO NOT DO THIS! Errors will result.
```

viewScale

Set the scale factors for the view.

The scaling is applied around the origin of the window, which is defined by the viewPos attribute. The scaling is applied before translation and rotation.

property viewport

Viewport rectangle (x, y, w, h) for the current draw buffer.

Values *x* and *y* define the origin, and *w* and *h* the size of the rectangle in pixels.

This is typically set to cover the whole buffer, however it can be changed for applications like multi-view rendering. Stimuli will draw according to the new shape of the viewport, for instance and stimulus with position (0, 0) will be drawn at the center of the viewport, not the window.

Examples

Constrain drawing to the left and right halves of the screen, where stimuli will be drawn centered on the new rectangle. Note that you need to set both the *viewport* and the *scissor* rectangle:

```
x, y, w, h = win.frameBufferSize  # size of the framebuffer
win.viewport = win.scissor = [x, y, w / 2.0, h]
# draw left stimuli ...

win.viewport = win.scissor = [x + (w / 2.0), y, w / 2.0, h]
# draw right stimuli ...
```

(continues on next page)

(continued from previous page)

```
# restore drawing to the whole screen
win.viewport = win.scissor = [x, y, w, h]
```

waitBlanking

After a call to `flip()` should we wait for the blank before the script continues.

property windowedSize

Size of the window to use when not fullscreen (w, h).

11.4.37 Window

A class representing a window for displaying one or more stimuli.

class psychopy.visual.Window(*size=(800, 600)*, *pos=None*, *color=(0, 0, 0)*, *colorSpace='rgb'*, *backgroundImage=None*, *backgroundFit='cover'*, *rgb=None*, *dkl=None*, *lms=None*, *fullscr=None*, *allowGUI=None*, *monitor=None*, *bitsMode=None*, *winType=None*, *units=None*, *gamma=None*, *blendMode='avg'*, *screen=0*, *viewScale=None*, *viewPos=None*, *viewOri=0.0*, *waitBlanking=True*, *allowStencil=False*, *multiSample=False*, *numSamples=2*, *stereo=False*, *name='window1'*, *title='PsychoPy'*, *checkTiming=True*, *useFBO=False*, *useRetina=True*, *autoLog=True*, *gammaErrorPolicy='raise'*, *bpc=(8, 8, 8)*, *depthBits=8*, *stencilBits=8*, *backendConf=None*, *infoMsg=None*)

Used to set up a context in which to draw objects, using either pyglet, pygame, or glfw.

The pyglet backend allows multiple windows to be created, allows the user to specify which screen to use (if more than one is available, duh!) and allows movies to be rendered.

The GLFW backend is a new addition which provides most of the same features as pyglet, but provides greater flexibility for complex display configurations.

Pygame may still work for you but it's officially deprecated in this project (we won't be fixing pygame-specific bugs).

These attributes can only be set at initialization. See further down for a list of attributes which can be changed after initialization of the Window, e.g. color, colorSpace, gamma etc.

Parameters

- **size** (*array-like of int*) – Size of the window in pixels [x, y].
- **pos** (*array-like of int*) – Location of the top-left corner of the window on the screen [x, y].
- **color** (*array-like of float*) – Color of background as [r, g, b] list or single value. Each gun can take values between -1.0 and 1.0.
- **fullscr** (*bool or None*) – Create a window in 'full-screen' mode. Better timing can be achieved in full-screen mode.
- **allowGUI** (*bool or None*) – If set to False, window will be drawn with no frame and no buttons to close etc., use *None* for value from preferences.
- **winType** (*str or None*) – Set the window type or back-end to use. If *None* then PsychoPy will revert to user/site preferences.
- **monitor** (*Monitor* or None) – The monitor to be used during the experiment. If *None* a default monitor profile will be used.

- **units** (*str or None*) – Defines the default units of stimuli drawn in the window (can be overridden by each stimulus). Values can be *None*, 'height' (of the window), 'norm' (normalised), 'deg', 'cm', 'pix'. See *Units for the window and stimuli* for explanation of options.
- **screen** (*int*) – Specifies the physical screen that stimuli will appear on ('pyglet' and 'glfw' *winType* only). Values can be >0 if more than one screen is present.
- **viewScale** (*array-like of float or None*) – Scaling factors [x, y] to apply custom scaling to the current units of the `Window` instance.
- **viewPos** (*array-like of float or None*) – If not *None*, redefines the origin within the window, in the units of the window. Values outside the borders will be clamped to lie on the border.
- **viewOri** (*float*) – A single value determining the orientation of the view in degrees.
- **waitBlanking** (*bool or None*) – After a call to `flip()` should we wait for the blank before the script continues.
- **bitsMode** (*bool*) – DEPRECATED in 1.80.02. Use BitsSharp class from pycrsltd instead.
- **checkTiming** (*bool*) – Whether to calculate frame duration on initialization. Estimated duration is saved in `monitorFramePeriod`. The message displayed on the screen can be set with the *infoMsg* argument.
- **allowStencil** (*bool*) – When set to *True*, this allows operations that use the OpenGL stencil buffer (notably, allowing the `Aperture` to be used).
- **multiSample** (*bool*) – If *True* and your graphics driver supports multisample buffers, multiple color samples will be taken per-pixel, providing an anti-aliased image through spatial filtering. This setting cannot be changed after opening a window. Only works with 'pyglet' and 'glfw' *winTypes*, and *useFBO* is *False*.
- **numSamples** (*int*) – A single value specifying the number of samples per pixel if multisample is enabled. The higher the number, the better the image quality, but can delay frame flipping. The largest number of samples is determined by `GL_MAX_SAMPLES`, usually 16 or 32 on newer hardware, will crash if number is invalid.
- **stereo** (*bool*) – If *True* and your graphics card supports quad buffers then this will be enabled. You can switch between left and right-eye scenes for drawing operations using `setBuffer()`.
- **title** (*str*) – Name of the Window according to your Operating System. This is the text which appears on the title sash.
- **useRetina** (*bool*) – In PsychoPy >1.85.3 this should always be *True* as pyglet (or Apple) no longer allows us to create a non-retina display. NB when you use Retina display the initial win size request will be in the larger pixels but subsequent use of `units='pix'` should refer to the tiny Retina pixels. Window.size will give the actual size of the screen in Retina pixels.
- **gammaErrorPolicy** (*str*) – If *raise*, an error is raised if the gamma table is unable to be retrieved or set. If *warn*, a warning is raised instead. If *ignore*, neither an error nor a warning are raised.
- **bpc** (*array_like or int*) – Bits per color (BPC) for the back buffer as a tuple to specify bit depths for each color channel separately (red, green, blue), or a single value to set all of them to the same value. Valid values depend on the output color depth of the display (screen) the window is set to use and the system graphics configuration. By default, it is assumed the display has 8-bits per color (8, 8, 8). Behaviour may be undefined for non-fullscreen windows, or if multiple screens are attached with varying color output depths.

- **depthBits** (*int*) – Back buffer depth bits. Default is 8, but can be set higher (eg. 24) if drawing 3D stimuli to minimize artifacts such a 'Z-fighting'.
- **stencilBits** (*int*) – Back buffer stencil bits. Default is 8.
- **backendConf** (*dict or None*) – Additional options to pass to the backend specified by *winType*. Each backend may provide unique functionality which may not be available across all of them. This allows you to pass special configuration options to a specific backend to configure the feature.
- **infoMsg** (*str or None*) – Message to display during frame rate measurement (i.e., when `checkTiming=True`). Default is None, which means that a default message is displayed. If you want to hide the message, pass an empty string.

Notes

- Some parameters (e.g. units) can now be given default values in the user/site preferences and these will be used if *None* is given here. If you do specify a value here it will take precedence over preferences.

size

Dimensions of the window's drawing area/buffer in pixels [w, h].

Type

array-like (float)

monitorFramePeriod

Refresh rate of the display if `checkTiming=True` on window instantiation.

Type

float

USE_LEGACY_GL = True

_assignFlipTime(*obj*, *attrib*, *format=<class 'float'>*)

Helper function to assign the time of last flip to the obj.attrib

Parameters

- **obj** (*dict or object*) – A mutable object (usually a dict of class instance).
- **attrib** (*str*) – Key or attribute of `obj` to assign the flip time to.
- **format** (*str, class or None*) – Format in which to return time, see clock.Timestamp.resolve() for more info. Defaults to *float*.

_checkMatchingSizes(*requested*, *actual*)

Checks whether the requested and actual screen sizes differ. If not then a warning is output and the window size is set to actual

_cleanEditables()

Make sure there are no dead refs in the editables list

_clearDepthBuffer()

Clear the depth buffer.

_endOfFlip(*clearBuffer*)

Override end of flip with custom color channel masking if required.

_getFrame(*rect=None*, *buffer='front'*)

Return the current Window as an image.

_getPixels(*rect=None*, *buffer='front'*, *includeAlpha=True*, *makeLum=False*)

Return an array of pixel values from the current window buffer or sub-region.

Parameters

- **rect** (*tuple[int], optional*) – The region of the window to capture in pixel coordinates (left, bottom, width, height). If *None*, the whole window is captured.
- **buffer** (*str, optional*) – Buffer to capture.
- **includeAlpha** (*bool, optional*) – Include the alpha channel in the returned array. Default is *True*.
- **makeLum** (*bool, optional*) – Convert the RGB values to luminance values. Values are rounded to the nearest integer. Default is *False*.

Returns

Pixel values as a 3D array of shape (height, width, channels). If *includeAlpha* is *False*, the array will have shape (height, width, 3). If *makeLum* is *True*, the array will have shape (height, width).

Return type

ndarray

Examples

Get the pixel values of the whole window:

```
pix = win._getPixels()
```

Get pixel values and convert to luminance and get average:

```
pix = win._getPixels(makeLum=True)
average = pix.mean()
```

_getRegionOfFrame(*rect=(-1, 1, 1, -1)*, *buffer='front'*, *power2=False*, *squarePower2=False*)

Deprecated function, here for historical reasons. You may now use *:py:attr:`~Window._getFrame()* and specify a rect to get a sub-region, just as used here.

power2 can be useful with older OpenGL versions to avoid interpolation in `PatchStim`. If power2 or squarePower2, it will expand rect dimensions up to next power of two. squarePower2 uses the max dimensions. You need to check what your hardware & OpenGL supports, and call `_getRegionOfFrame()` as appropriate.

_renderFBO()

Perform a warp operation.

(in this case a copy operation without any warping)

_setCurrent()

Make this window's OpenGL context current.

If called on a window whose context is current, the function will return immediately. This reduces the number of redundant calls if no context switch is required. If `useFBO=True`, the framebuffer is bound after the context switch.

_setupFrameBuffer()

Setup the framebuffer object for this window.

Returns

True if the framebuffer was successfully setup, *False* otherwise. If *False*, the framebuffer was not complete. Make sure that your driver supports the necessary formats.

Return type

bool

`_setupGL()`

Setup OpenGL state for this window.

`_setupGamma`(*gammaVal*)

A private method to work out how to handle gamma for this Window given that the user might have specified an explicit value, or maybe gave a Monitor.

`_startOfFlip()`

Custom hardware classes may want to prevent flipping from occurring and can override this method as needed.

Return *True* to indicate hardware flip.

`_updateProjectionMatrix()`

Update the default projection matrix based on the current window settings.

`_updateViewMatrix()`

Update the default orthographic view matrix based on the current window settings.

`addEditable`(*editable*)

Adds an editable element to the screen (something to which characters can be sent with meaning from the keyboard).

The current editable object receiving chars is Window.currentEditable

Parameters

`editable`

Returns

`property ambientLight`

Ambient light color for the scene [r, g, b, a]. Values range from 0.0 to 1.0. Only applicable if *useLights* is *True*.

Examples

Setting the ambient light color:

```
win.ambientLight = [0.5, 0.5, 0.5]

# don't do this!!!
win.ambientLight[0] = 0.5
win.ambientLight[1] = 0.5
win.ambientLight[2] = 0.5
```

`applyEyeTransform`(*clearDepth=True*)

Apply the current view and projection matrices.

Matrices specified by attributes *`viewMatrix`* and *`projectionMatrix`* are applied using 'immediate mode' OpenGL functions. Subsequent drawing operations will be affected until *`flip()`* is called.

All transformations in `GL_PROJECTION` and `GL_MODELVIEW` matrix stacks will be cleared (set to identity) prior to applying. After this is called, the current matrix mode will be set to `GL_MODELVIEW`.

Parameters

clearDepth (*bool*) – Clear the depth buffer. This may be required prior to rendering 3D objects.

Examples

Using a custom view and projection matrix:

```
# Must be called every frame since these values are reset after
# `flip()` is called!
win.viewMatrix = viewtools.lookAt( ... )
win.projectionMatrix = viewtools.perspectiveProjectionMatrix( ... )
win.applyEyeTransform()
# draw 3D objects here ...
```

property aspect

Aspect ratio of the current viewport (width / height).

backgroundFit

How should the background image of this window fit? Options are:

None, "None", "none"

No scaling is applied, image is present at its pixel size unaltered.

"cover"

Image is scaled such that it covers the whole screen without changing its aspect ratio. In other words, both dimensions are evenly scaled such that its SHORTEST dimension matches the window's LONGEST dimension.

"contain"

Image is scaled such that it is contained within the screen without changing its aspect ratio. In other words, both dimensions are evenly scaled such that its LONGEST dimension matches the window's SHORTEST dimension.

"scaleDown", "scale-down", "scaledown"

If image is bigger than the window along any dimension, it will behave as if backgroundFit were "contain". Otherwise, it will behave as if backgroundFit were None.

backgroundImage

Background image for the window, can be either a visual.ImageStim object or anything which could be passed to visual.ImageStim.image to create one. Will be drawn each time *win.flip()* is called, meaning it is always below all other contents of the window.

blendMode

Blend mode to use.

callOnFlip(*function*, **args*, ***kwargs*)

Call a function immediately after the next *flip()* command.

The first argument should be the function to call, the following args should be used exactly as you would for your normal call to the function (can use ordered arguments or keyword arguments as normal).

e.g. If you have a function that you would normally call like this:

```
pingMyDevice(portToPing, channel=2, level=0)
```

then you could call *callOnFlip()* to have the function call synchronized with the frame flip like this:

```
win.callOnFlip(pingMyDevice, portToPing, channel=2, level=0)
```

clearAutoDraw()

Remove all autoDraw components, meaning they get autoDraw set to False and are not added to any list (as in .stashAutoDraw)

clearBuffer(*color=True*, *depth=False*, *stencil=False*)

Clear the present buffer (to which you are currently drawing) without flipping the window.

Useful if you want to generate movie sequences from the back buffer without actually taking the time to flip the window.

Set *color* prior to clearing to set the color to clear the color buffer to. By default, the depth buffer is cleared to a value of 1.0.

Parameters

- **color** (*bool*) – Buffers to clear.
- **depth** (*bool*) – Buffers to clear.
- **stencil** (*bool*) – Buffers to clear.

Examples

Clear the color buffer to a specified color:

```
win.color = (1, 0, 0)
win.clearBuffer(color=True)
```

Clear only the depth buffer, *depthMask* must be *True* or else this will have no effect. Depth mask is usually *True* by default, but may change:

```
win.depthMask = True
win.clearBuffer(color=False, depth=True, stencil=False)
```

close()

Close the window (and reset the Bits++ if necess).

property color

Set the color of the window.

This command sets the color that the blank screen will have on the next clear operation. As a result it effectively takes TWO *flip()* operations to become visible (the first uses the color to create the new screen, the second presents that screen to the viewer). For this reason, if you want to changed background color of the window "on the fly", it might be a better idea to draw a `Rect` that fills the whole window with the desired `Rect.fillColor` attribute. That'll show up on first flip.

See other stimuli (e.g. *GratingStim.color*) for more info on the color attribute which essentially works the same on all PsychoPy stimuli.

See *Color spaces* for further information about the ways to specify colors and their various implications.

property colorSpace

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                       color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

property contentScaleFactor

Scaling factor (*float*) to use when drawing to the backbuffer to convert framebuffer to client coordinates.

See also

getContentScaleFactor

property convergeOffset

Convergence offset from monitor in centimeters.

This is value corresponds to the offset from screen plane to set the convergence plane (or point for *toe-in* projections). Positive offsets move the plane farther away from the viewer, while negative offsets nearer. This value is used by *setPerspectiveView* and should be set before calling it to take effect.

Notes

- This value is only applicable for *setToeIn* and *setOffAxisView*.

coordToRay(*screenXY*)

Convert a screen coordinate to a direction vector.

Takes a screen/window coordinate and computes a vector which projects a ray from the viewpoint through it (line-of-sight). Any 3D point touching the ray will appear at the screen coordinate.

Uses the current *viewport* and *projectionMatrix* to calculate the vector. The vector is in eye-space, where the origin of the scene is centered at the viewpoint and the forward direction aligned with the -Z axis. A ray of (0, 0, -1) results from a point at the very center of the screen assuming symmetric frustums.

Note that if you are using a flipped/mirrored view, you must invert your supplied screen coordinates (*screenXY*) prior to passing them to this function.

Parameters

screenXY (*array_like*) – X, Y screen coordinate. Must be in units of the window.

Returns

Normalized direction vector [x, y, z].

Return type

ndarray

Examples

Getting the direction vector between the mouse cursor and the eye:

```
mx, my = mouse.getPos()
dir = win.coordToRay((mx, my))
```

Set the position of a 3D stimulus object using the mouse, constrained to a plane. The object origin will always be at the screen coordinate of the mouse cursor:

```
# the eye position in the scene is defined by a rigid body pose
win.viewMatrix = camera.getViewMatrix()
win.applyEyeTransform()

# get the mouse location and calculate the intercept
mx, my = mouse.getPos()
ray = win.coordToRay([mx, my])
result = intersectRayPlane(   # from mathtools
    orig=camera.pos,
    dir=camera.transformNormal(ray),
    planeOrig=(0, 0, -10),
    planeNormal=(0, 1, 0))

# if result is `None`, there is no intercept
if result is not None:
    pos, dist = result
    objModel.thePose.pos = pos
else:
    objModel.thePose.pos = (0, 0, -10)  # plane origin
```

If you don't define the position of the viewer with a *RigidBodyPose*, you can obtain the appropriate eye position and rotate the ray by doing the following:

```
pos = numpy.linalg.inv(win.viewMatrix)[:3, 3]
ray = win.coordToRay([mx, my]).dot(win.viewMatrix[:3, :3])
# then ...
result = intersectRayPlane(
    orig=pos,
    dir=ray,
    planeOrig=(0, 0, -10),
    planeNormal=(0, 1, 0))
```

property cullFace

True if face culling is enabled.`

property cullFaceMode

Face culling mode, either *back*, *front* or *both*.

property currentEditable

The editable (Text?) object that currently has key focus

property depthFunc

Depth test comparison function for rendering.

property depthMask

True if depth masking is enabled. Writing to the depth buffer will be disabled.

property depthTest

True if depth testing is enabled.

property draw3d

True if 3D drawing is enabled on this window.

property eyeOffset

Eye offset in centimeters.

This value is used by *setPerspectiveView* to apply a lateral offset to the view, therefore it must be set prior to calling it. Use a positive offset for the right eye, and a negative one for the left. Offsets should be the distance to from the middle of the face to the center of the eye, or half the inter-ocular distance.

property farClip

Distance to the far clipping plane in meters.

flip(*clearBuffer=True*)

Flip the front and back buffers after drawing everything for your frame. (This replaces the `update()` method, better reflecting what is happening underneath).

Parameters
clearBuffer (*bool, optional*) – Clear the draw buffer after flipping. Default is *True*.

Returns
Wall-clock time in seconds the flip completed. Returns *None* if `waitBlanking` is *False*.

Return type
float or None

Notes

- The time returned when `waitBlanking` is *True* corresponds to when the graphics driver releases the draw buffer to accept draw commands again. This time is usually close to the vertical sync signal of the display.

Examples

Results in a clear screen after flipping:

```
win.flip(clearBuffer=True)
```

The screen is not cleared (so represent the previous screen):

```
win.flip(clearBuffer=False)
```

fps()

Report the frames per second since the last call to this function (or since the window was created if this is first call)

property frameBufferSize

Size of the framebuffer in pixels (w, h).

property frontFace

Face winding order to define front, either *ccw* or *cw*.

property fullscr

Return whether the window is in fullscreen mode.

`gamma`

Set the monitor gamma for linearization.

> ⚠ **Warning**
>
> Don't use this if using a Bits++ or Bits#, as it overrides monitor settings.

`gammaRamp`

Sets the hardware CLUT using a specified 3xN array of floats ranging between 0.0 and 1.0.

Array must have a number of rows equal to 2 ^ max(bpc).

`getActualFrameRate`(*nIdentical=10*, *nMaxFrames=100*, *nWarmUpFrames=10*, *threshold=1*, *infoMsg=None*)

Measures the actual frames-per-second (FPS) for the screen.

This is done by waiting (for a max of *nMaxFrames*) until *nIdentical* frames in a row have identical frame times (std dev below *threshold* ms).

Parameters

- **nIdentical** (*int, optional*) – The number of consecutive frames that will be evaluated. Higher –> greater precision. Lower –> faster.
- **nMaxFrames** (*int, optional*) – The maximum number of frames to wait for a matching set of nIdentical.
- **nWarmUpFrames** (*int, optional*) – The number of frames to display before starting the test (this is in place to allow the system to settle after opening the *Window* for the first time.
- **threshold** (*int or float, optional*) – The threshold for the std deviation (in ms) before the set are considered a match.

Returns

Frame rate (FPS) in seconds. If there is no such sequence of identical frames a warning is logged and *None* will be returned.

Return type

float or None

`getContentScaleFactor`()

Get the scaling factor required for scaling correctly on high-DPI displays.

If the returned value is 1.0, no scaling needs to be applied to objects drawn on the backbuffer. A value >1.0 indicates that the backbuffer is larger than the reported client area, requiring points to be scaled to maintain constant size across similarly sized displays. In other words, the scaling required to convert framebuffer to client coordinates.

Returns

Scaling factor to be applied along both horizontal and vertical dimensions.

Return type

float

Examples

Get the size of the client area:

```
clientSize = win.frameBufferSize / win.getContentScaleFactor()
```

Get the framebuffer size from the client size:

```
frameBufferSize = win.clientSize * win.getContentScaleFactor()
```

Convert client (window) to framebuffer pixel coordinates (eg., a mouse coordinate, vertices, etc.):

```
# `mousePosXY` is an array ...
frameBufferXY = mousePosXY * win.getContentScaleFactor()
# you can also use the attribute ...
frameBufferXY = mousePosXY * win.contentScaleFactor
```

Notes

- This value is only valid after the window has been fully realized.

getFutureFlipTime(*targetTime=0*, *clock=None*)

The expected time of the next screen refresh. This is currently calculated as win._lastFrameTime + refreshInterval

Parameters

- **targetTime** (*float*) – The delay *from now* for which you want the flip time. 0 will give the because that the earliest we can achieve. 0.15 will give the schedule flip time that gets as close to 150 ms as possible
- **clock** (*None, 'ptb', 'now' or any Clock object*) – If True then the time returned is compatible with ptb.GetSecs()
- **verbose** (*bool*) – Set to True to view the calculations along the way

getMovieFrame(*buffer='front'*)

Capture the current Window as an image.

Saves to stack for `saveMovieFrames()`. As of v1.81.00 this also returns the frame as a PIL image

This can be done at any time (usually after a `flip()` command).

Frames are stored in memory until a `saveMovieFrames()` command is issued. You can issue `getMovieFrame()` as often as you like and then save them all in one go when finished.

The back buffer will return the frame that hasn't yet been 'flipped' to be visible on screen but has the advantage that the mouse and any other overlapping windows won't get in the way.

The default front buffer is to be called immediately after a `flip()` and gives a complete copy of the screen at the window's coordinates.

Parameters

buffer (*str, optional*) – Buffer to capture.

Returns

Buffer pixel contents as a PIL/Pillow image object.

Return type

Image

getMsPerFrame(*nFrames=60*, *showVisual=False*, *msg=''*, *msDelay=0.0*)

Assesses the monitor refresh rate (average, median, SD) under current conditions, over at least 60 frames.

Records time for each refresh (frame) for n frames (at least 60), while displaying an optional visual. The visual is just eye-candy to show that something is happening when assessing many frames. You can also give it text to display instead of a visual, e.g., `msg='(testing refresh rate...)'`; setting msg implies `showVisual == False`.

To simulate refresh rate under cpu load, you can specify a time to wait within the loop prior to doing the *flip()*. If 0 < msDelay < 100, wait for that long in ms.

Returns timing stats (in ms) of:

- average time per frame, for all frames
- standard deviation of all frames
- median, as the average of 12 frame times around the median (~monitor refresh rate)

Author

- 2010 written by Jeremy Gray

hideMessage()

Remove any message that is currently being displayed.

hidePilotingIndicator()

Hide the visual indicator which shows we are in piloting mode.

property lights

Scene lights.

This is specified as an array of *~psychopy.visual.LightSource* objects. If a single value is given, it will be converted to a *list* before setting. Set *useLights* to *True* before rendering to enable lighting/shading on subsequent objects. If *lights* is *None* or an empty *list*, no lights will be enabled if *useLights=True*, however, the scene ambient light set with *ambientLight* will be still be used.

Examples

Create a directional light source and add it to scene lights:

```
dirLight = gltools.LightSource((0., 1., 0.), lightType='directional')
win.lights = dirLight  # `win.lights` will be a list when accessed!
```

Multiple lights can be specified by passing values as a list:

```
myLights = [gltools.LightSource((0., 5., 0.)),
            gltools.LightSource((-2., -2., 0.))
win.lights = myLights
```

logOnFlip(*msg*, *level*, *obj=None*)

Send a log message that should be time-stamped at the next *flip()* command.

Parameters

- **msg** (*str*) – The message to be logged.
- **level** (*int*) – The level of importance for the message.
- **obj** (*object, optional*) – The python object that might be associated with this message if desired.

property mouseVisible

Returns the visibility of the mouse cursor.

property nearClip

Distance to the near clipping plane in meters.

nextEditable()

Moves focus of the cursor to the next editable window

property projectionMatrix

Projection matrix defined as a 4x4 numpy array.

recordFrameIntervals

Record time elapsed per frame.

Provides accurate measures of frame intervals to determine whether frames are being dropped. The intervals are the times between calls to *flip()*. Set to *True* only during the time-critical parts of the script. Set this to *False* while the screen is not being updated, i.e., during any slow, non-frame-time-critical sections of your code, including inter-trial-intervals, `event.waitkeys()`, `core.wait()`, or `image.setImage()`.

Examples

Enable frame interval recording, successive frame intervals will be stored:

```
win.recordFrameIntervals = True
```

Frame intervals can be saved by calling the *saveFrameIntervals* method:

```
win.saveFrameIntervals()
```

removeEditable(*editable*)

resetEyeTransform(*clearDepth=True*)

Restore the default projection and view settings to PsychoPy defaults. Call this prior to drawing 2D stimuli objects (i.e. GratingStim, ImageStim, Rect, etc.) if any eye transformations were applied for the stimuli to be drawn correctly.

Parameters

clearDepth (*bool*) – Clear the depth buffer upon reset. This ensures successive draw commands are not affected by previous data written to the depth buffer. Default is *True*.

Notes

- Calling *flip()* automatically resets the view and projection to defaults. So you don't need to call this unless you are mixing 3D and 2D stimuli.

Examples

Going between 3D and 2D stimuli:

```
# 2D stimuli can be drawn before setting a perspective projection
win.setPerspectiveView()
# draw 3D stimuli here ...
win.resetEyeTransform()
# 2D stimuli can be drawn here again ...
win.flip()
```

`resetViewport()`

Reset the viewport to cover the whole framebuffer.

Set the viewport to match the dimensions of the back buffer or framebuffer (if *useFBO=True*). The scissor rectangle is also set to match the dimensions of the viewport.

`retrieveAutoDraw()`

Add all stimuli which are on 'hold' back into the autoDraw list, and clear the hold list.

`property rgb`

`saveFrameIntervals`(*fileName=None*, *clear=True*)

Save recorded screen frame intervals to disk, as comma-separated values.

Parameters

- **fileName** (*None* or str) – *None* or the filename (including path if necessary) in which to store the data. If None then 'lastFrameIntervals.log' will be used.
- **clear** (*bool*) – Clear buffer frames intervals were stored after saving. Default is *True*.

`saveMovieFrames`(*fileName*, *codec='libx264'*, *fps=30*, *clearFrames=True*)

Writes any captured frames to disk.

Will write any format that is understood by PIL (tif, jpg, png, . . .)

Parameters

- **filename** (*str*) – Name of file, including path. The extension at the end of the file determines the type of file(s) created. If an image type (e.g. .png) is given, then multiple static frames are created. If it is .gif then an animated GIF image is created (although you will get higher quality GIF by saving PNG files and then combining them in dedicated image manipulation software, such as GIMP). On Windows and Linux *.mpeg* files can be created if *pymedia* is installed. On macOS *.mov* files can be created if the pyobjc-frameworks-QTKit is installed. Unfortunately the libs used for movie generation can be flaky and poor quality. As for animated GIFs, better results can be achieved by saving as individual .png frames and then combining them into a movie using software like ffmpeg.
- **codec** (*str, optional*) – The codec to be used **by moviepy** for mp4/mpg/mov files. If *None* then the default will depend on file extension. Can be one of `libx264`, `mpeg4` for mp4/mov files. Can be `rawvideo`, `png` for avi files (not recommended). Can be `libvorbis` for ogv files. Default is `libx264`.
- **fps** (*int, optional*) – The frame rate to be used throughout the movie. **Only for quicktime (.mov) movies.**. Default is *30*.
- **clearFrames** (*bool, optional*) – Set this to *False* if you want the frames to be kept for additional calls to `saveMovieFrames`. Default is *True*.

Examples

Writes a series of static frames as frame001.tif, frame002.tif etc.:

```
myWin.saveMovieFrames('frame.tif')
```

As of PsychoPy 1.84.1 the following are written with moviepy:

```
myWin.saveMovieFrames('stimuli.mp4') # codec = 'libx264' or 'mpeg4'
myWin.saveMovieFrames('stimuli.mov')
myWin.saveMovieFrames('stimuli.gif')
```

property scissor

Scissor rectangle (x, y, w, h) for the current draw buffer.

Values *x* and *y* define the origin, and *w* and *h* the size of the rectangle in pixels. The scissor operation is only active if *scissorTest=True*.

Usually, the scissor and viewport are set to the same rectangle to prevent drawing operations from *spilling* into other regions of the screen. For instance, calling *clearBuffer* will only clear within the scissor rectangle.

Setting the scissor rectangle but not the viewport will restrict drawing within the defined region (like a rectangular aperture), not changing the positions of stimuli.

property scissorTest

True if scissor testing is enabled.

property screenshot

setBuffer(*buffer*, *clear=True*)

Choose which buffer to draw to ('left' or 'right').

Requires the Window to be initialised with stereo=True and requires a graphics card that supports quad buffering (e,g nVidia Quadro series)

PsychoPy always draws to the back buffers, so 'left' will use `GL_BACK_LEFT` This then needs to be flipped once both eye's buffers have been rendered.

Parameters

- **buffer** (*str*) – Buffer to draw to. Can either be 'left' or 'right'.
- **clear** (*bool, optional*) – Clear the buffer before drawing. Default is `True`.

Examples

Stereoscopic rendering example using quad-buffers:

```
win = visual.Window(...., stereo=True)
while True:
    # clear may not actually be needed
    win.setBuffer('left', clear=True)
    # do drawing for left eye
    win.setBuffer('right', clear=True)
    # do drawing for right eye
    win.flip()
```

setMouseType(*name='arrow'*)

Change the appearance of the cursor for this window. Cursor types provide contextual hints about how to interact with on-screen objects.

The graphics used 'standard cursors' provided by the operating system. They may vary in appearance and hot spot location across platforms. The following names are valid on most platforms:

- `arrow` : Default pointer.
- `ibeam` : Indicates text can be edited.
- `crosshair` : Crosshair with hot-spot at center.
- `hand` : A pointing hand.
- `hresize` : Double arrows pointing horizontally.

- `vresize` : Double arrows pointing vertically.

Parameters

name (*str*) – Type of standard cursor to use (see above). Default is `arrow`.

Notes

- On Windows the `crosshair` option is negated with the background color. It will not be visible when placed over 50% grey fields.

`setOffAxisView`(*applyTransform=True*, *clearDepth=True*)

Set an off-axis projection.

Create an off-axis projection for subsequent rendering calls. Sets the *viewMatrix* and *projectionMatrix* accordingly so the scene origin is on the screen plane. If *eyeOffset* is correct and the view distance and screen size is defined in the monitor configuration, the resulting view will approximate *ortho-stereo* viewing.

The convergence plane can be adjusted by setting *convergeOffset*. By default, the convergence plane is set to the screen plane. Any points on the screen plane will have zero disparity.

Parameters

- **applyTransform** (*bool*) – Apply transformations after computing them in immediate mode. Same as calling *applyEyeTransform()* afterwards.
- **clearDepth** (*bool, optional*) – Clear the depth buffer.

`setOrthographicView`(*applyTransform=True*, *clearDepth=True*)

Set the projection and view matrix to render with orthographic view.

Orthographic projection is used to render 3D objects without perspective distortion. The scene origin is centered on the screen plane. The frustum is defined by the size of the window in pixels, with the origin at the center of the window. 2D stimuli are typically drawn using this projection.

Note that the values of *projectionMatrix* and *viewMatrix* will be replaced when calling this function.

Parameters

- **applyTransform** (*bool*) – Apply transformations after computing them in immediate mode. Same as calling *applyEyeTransform()* afterwards if *False*.
- **clearDepth** (*bool, optional*) – Clear the depth buffer.

`setPerspectiveView`(*applyTransform=True*, *clearDepth=True*)

Set the projection and view matrix to render with perspective.

Matrices are computed using values specified in the monitor configuration with the scene origin on the screen plane. Calculations assume units are in meters. If *eyeOffset != 0*, the view will be transformed laterally, however the frustum shape will remain the same.

Note that the values of *projectionMatrix* and *viewMatrix* will be replaced when calling this function.

Parameters

- **applyTransform** (*bool*) – Apply transformations after computing them in immediate mode. Same as calling *applyEyeTransform()* afterwards if *False*.
- **clearDepth** (*bool, optional*) – Clear the depth buffer.

`setToeInView`(*applyTransform=True*, *clearDepth=True*)

Set toe-in projection.

Create a toe-in projection for subsequent rendering calls. Sets the *viewMatrix* and *projectionMatrix* accordingly so the scene origin is on the screen plane. The value of *convergeOffset* will define the convergence point of the view, which is offset perpendicular to the center of the screen plane. Points falling on a vertical line at the convergence point will have zero disparity.

Parameters

- **applyTransform** (*bool*) – Apply transformations after computing them in immediate mode. Same as calling *applyEyeTransform()* afterwards.
- **clearDepth** (*bool, optional*) – Clear the depth buffer.

Notes

- This projection mode is only 'correct' if the viewer's eyes are converged at the convergence point. Due to perspective, this projection introduces vertical disparities which increase in magnitude with eccentricity. Use *setOffAxisView* if you want to display something the viewer can look around the screen comfortably.

`setViewOri`(*value*, *log=True*)

`setViewScale`(*value*, *log=True*)

`showMessage`(*msg*)

Show a message in the window. This can be used to show information to the participant.

This creates a TextBox2 object that is displayed in the window. The text can be updated by calling this method again with a new message. The updated text will appear the next time *draw()* is called.

Parameters

msg (*str or None*) – Message text to display. If None, then any existing message is removed.

`showPilotingIndicator`()

Show the visual indicator which shows we are in piloting mode.

`property size`

Size of the drawable area in pixels (w, h).

`stashAutoDraw`()

Put autoDraw components on 'hold', meaning they get autoDraw set to False but are added to an internal list to be 'released' when .releaseAutoDraw is called.

`property stencilTest`

True if stencil testing is enabled.

`timeOnFlip`(*obj*, *attrib*, *format=<class 'float'>*)

Retrieves the time on the next flip and assigns it to the *attrib* for this *obj*.

Parameters

- **obj** (*dict or object*) – A mutable object (usually a dict of class instance).
- **attrib** (*str*) – Key or attribute of *obj* to assign the flip time to.
- **format** (*str, class or None*) – Format in which to return time, see clock.Timestamp.resolve() for more info. Defaults to *float*.

Examples

Assign time on flip to the `tStartRefresh` key of `myTimingDict`:

```
win.getTimeOnFlip(myTimingDict, 'tStartRefresh')
```

`title`

`units`

None, 'height' (of the window), 'norm', 'deg', 'cm', 'pix' Defines the default units of stimuli initialized in the window. I.e. if you change units, already initialized stimuli won't change their units.

Can be overridden by each stimulus, if units is specified on initialization.

See *Units for the window and stimuli* for explanation of options.

`updateLights`(*index=None*)

Explicitly update scene lights if they were modified.

This is required if modifications to objects referenced in *lights* have been changed since assignment. If you removed or added items of *lights* you must refresh all of them.

Parameters

index (*int, optional*) – Index of light source in *lights* to update. If *None*, all lights will be refreshed.

Examples

Call *updateLights* if you modified lights directly like this:

```
win.lights[1].diffuseColor = [1., 0., 0.]
win.updateLights(1)
```

`property useLights`

Enable scene lighting.

Lights will be enabled if using legacy OpenGL lighting. Stimuli using shaders for lighting should check if *useLights* is *True* since this will have no effect on them, and disable or use a no lighting shader instead. Lights will be transformed to the current view matrix upon setting to *True*.

Lights are transformed by the present *GL_MODELVIEW* matrix. Setting *useLights* will result in their positions being transformed by it. If you want lights to appear at the specified positions in world space, make sure the current matrix defines the view/eye transformation when setting *useLights=True*.

This flag is reset to *False* at the beginning of each frame. Should be *False* if rendering 2D stimuli or else the colors will be incorrect.

`property viewMatrix`

View matrix defined as a 4x4 numpy array.

`viewOri`

Set the rotation of the view in degrees.

The rotation is applied around the origin of the window, which is defined by the viewPos attribute. The rotation is applied after scaling but before translation.

`viewPos`

The origin of the window onto which stimulus-objects are drawn.

The value should be given in the units defined for the window. NB: Never change a single component (x or y) of the origin, instead replace the viewPos-attribute in one shot, e.g.:

```
win.viewPos = [new_xval, new_yval]  # This is the way to do it
win.viewPos[0] = new_xval  # DO NOT DO THIS! Errors will result.
```

viewScale

Set the scale factors for the view.

The scaling is applied around the origin of the window, which is defined by the viewPos attribute. The scaling is applied before translation and rotation.

property viewport

Viewport rectangle (x, y, w, h) for the current draw buffer.

Values *x* and *y* define the origin, and *w* and *h* the size of the rectangle in pixels.

This is typically set to cover the whole buffer, however it can be changed for applications like multi-view rendering. Stimuli will draw according to the new shape of the viewport, for instance and stimulus with position (0, 0) will be drawn at the center of the viewport, not the window.

Examples

Constrain drawing to the left and right halves of the screen, where stimuli will be drawn centered on the new rectangle. Note that you need to set both the *viewport* and the *scissor* rectangle:

```
x, y, w, h = win.frameBufferSize  # size of the framebuffer
win.viewport = win.scissor = [x, y, w / 2.0, h]
# draw left stimuli ...

win.viewport = win.scissor = [x + (w / 2.0), y, w / 2.0, h]
# draw right stimuli ...

# restore drawing to the whole screen
win.viewport = win.scissor = [x, y, w, h]
```

waitBlanking

After a call to `flip()` should we wait for the blank before the script continues.

property windowedSize

Size of the window to use when not fullscreen (w, h).

11.4.38 psychopy.visual.windowframepack - Pack multiple monochrome images into RGB frame

ProjectorFramePacker

class psychopy.visual.windowframepack.**ProjectorFramePacker**(*win*)

Class which packs 3 monochrome images per RGB frame.

Allowing 180Hz stimuli with DLP projectors such as TI LightCrafter 4500.

The class overrides methods of the visual.Window class to pack a monochrome image into each RGB channel. PsychoPy is running at 180Hz. The display device is running at 60Hz. The output projector is producing images at 180Hz.

Frame packing can work with any projector which can operate in 'structured light mode' where each RGB channel is presented sequentially as a monochrome image. Most home and office projectors cannot operate in this mode, but projectors designed for machine vision applications typically will offer this feature.

Example usage to use ProjectorFramePacker:

```
from psychopy.visual.windowframepack import ProjectorFramePacker
win = Window(monitor='testMonitor', screen=1,
             fullscr=True, useFBO = True)
framePacker = ProjectorFramePacker (win)
```

Parameters
win : Handle to the window.

endOfFlip(*clearBuffer*)

Mask RGB cyclically after each flip. We ignore clearBuffer and just auto-clear after each hardware flip.

startOfFlip()

Return True if all channels of the RGB frame have been filled with monochrome images, and the associated window should perform a hardware flip

11.4.39 psychopy.visual.windowwarp - warping to spherical, cylindrical, or other projections

Copyright (C) 2014 Allen Institute for Brain Science

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License Version 3 as published by the Free Software Foundation on 29 June 2007. This program is distributed WITHOUT WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE OR ANY OTHER WARRANTY, EXPRESSED OR IMPLIED. See the GNU General Public License Version 3 for more details. You should have received a copy of the GNU General Public License along with this program. If not, see http://www.gnu.org/licenses/

Warper

class psychopy.visual.windowwarp.**Warper**(*win*, *warp=None*, *warpfile=None*, *warpGridsize=300*, *eyepoint=(0.5, 0.5)*, *flipHorizontal=False*, *flipVertical=False*)

Class to perform warps.

Supports spherical, cylindrical, warpfile, or None (disabled) warps

Warping is a final operation which can be optionally performed on each frame just before transmission to the display. It is useful for perspective correction when the eye to monitor distance is small (say, under 50 cm), or when projecting to domes or other non-planar surfaces.

These attributes define the projection and can be altered dynamically using the changeProjection() method.

Parameters
win : Handle to the window.

warp
['spherical', 'cylindrical, 'warpfile' or *None*] This table gives the main properties of each projection

Warp	eyepoint modifies warp	verticals parallel	horizontals parallel	perspective correct
spherical	y	n	n	y
cylindrical	y	y	n	n
warpfile	n	?	?	?
None	n	y	y	n

warpfile
[*None* or filename containing Blender and Paul Bourke] compatible warp definition. (see http://paulbourke.net/dome/warpingfisheye/)

warpGridsize
[300] Defines the resolution of the warp in both X and Y when not using a warpfile. Typical values would be 64-300 trading off tolerance for jaggies for speed.

eyepoint
[[0.5, 0.5] center of the screen] Position of the eye in X and Y as a fraction of the normalized screen width and height. [0,0] is the bottom left of the screen. [1,1] is the top right of the screen.

flipHorizontal: True or *False*
Flip the entire output horizontally. Useful for back projection scenarious.

flipVertical: True or *False*
Flip the entire output vertically. useful if projector is flipped upside down.

Notes

1) **The eye distance from the screen is initialized from the**
monitor definition.

2) **The eye distance can be altered dynamically by changing**
'warper.dist_cm' and then calling changeProjection().

Example usage to create a spherical projection:

```
from psychopy.visual.windowwarp import Warper
win = Window(monitor='testMonitor', screen=1,
             fullscr=True, useFBO = True)
warper = Warper(win,
                warp='spherical',
                warpfile = "",
                warpGridsize = 128,
                eyepoint = [0.5, 0.5],
                flipHorizontal = False,
                flipVertical = False)
```

changeProjection(*warp*, *warpfile=None*, *eyepoint=(0.5, 0.5)*, *flipHorizontal=False*, *flipVertical=False*)
Allows changing the warp method on the fly. Uses the same parameter definitions as constructor.

Window to display all stimuli below.

Windows and and display devices:

- *Window* is the main class to display objects

- *Warper* for non-flat projection screens
- *ProjectorFramePacker* for handling displays with 'structured light mode' to achieve high framerates
- *Rift* for Oculus Rift support (Windows 64bit only)
- *VisualSystemHD* for NordicNeuralLab's VisualSystemHD in-scanner display.

Commonly used:

- *ImageStim* to show images
- *TextStim* to show text
- *TextBox2* rewrite of TextStim (faster, editable with more layout options and formatting)

Shapes (all special classes of `ShapeStim`):

- *ShapeStim* to draw shapes with arbitrary numbers of vertices
- *Rect* to show rectangles
- *Circle* to show circles
- *Polygon* to show polygons
- *Line* to show a line
- *Pie* to show wedges and semi-circles

Images and patterns:

- *SimpleImageStim* to show images without bells and whistles
- *GratingStim* to show gratings
- `RadialStim` to show annulus, a rotating wedge, a checkerboard etc
- *NoiseStim* to show filtered noise patterns of various forms
- *EnvelopeGrating* to generate second-order stimuli (gratings that can have a carrier and envelope)

Multiple stimuli:

- *ElementArrayStim* to show many stimuli of the same type
- *DotStim* to show and control movement of dots

3D shapes, materials, and lighting:

- *LightSource* to define a light source in a scene
- *SceneSkybox* to render a background skybox for VR and 3D scenes
- *BlinnPhongMaterial* to specify a material using the Blinn-Phong lighting model
- *SphereStim* to show a 3D sphere
- *BoxStim* to show 3D boxes and cubes
- *PlaneStim* to show 3D plane
- *ObjMeshStim* to show Wavefront OBJ meshes loaded from files

Other stimuli:

- *MovieStim* to show movies
- `VlcMovieStim` to show movies using VLC
- *Slider* a new improved class to collect ratings

- `RatingScale` to collect ratings
- *`CustomMouse`* to change the cursor in windows with GUI. OBS: will be deprecated soon

Meta stimuli (stimuli that operate on other stimuli):

- *`BufferImageStim`* to make a faster-to-show "screenshot" of other stimuli
- *`Aperture`* to restrict visibility area of other stimuli

Helper functions:

- *`filters`* for creating grating textures and Gaussian masks etc.
- `visualhelperfunctions` for tests about whether one stimulus contains another
- *`unittools`* to convert deg<->radians
- *`monitorunittools`* to convert cm<->pix<->deg etc.
- *`colorspacetools`* to convert between supported color spaces
- *`viewtools`* to work with view projections
- *`mathtools`* to work with vectors, quaternions, and matrices
- *`gltools`* to work with OpenGL directly (under development)

11.5 `psychopy.sound` - for playback and recording of sound

The *psychopy.sound* module provides an interface for audio playback and recording devices. It also provides tools for working with audio samples and performing speech-to-text transcription.

11.5.1 `Sound` - for audio playback

Audio playback is handled by the `Sound` class. currently supports a choice of sound engines: *PTB*, *pyo*, *sounddevice* or *pygame*. You can select which will be used via the *audioLib* preference. *sound.Sound()* will then refer to one of the following backends:

- *`SoundPTB`*
- SoundDevice
- SoundPyo
- SoundPygame

This preference can be set on a per-experiment basis by importing preferences, and *setting the audioLib option* to use. Audio playback backends vary in performance due to all sorts of factors. Based on testing done by the team and reports from users, their performance can be summarized as follows:

- The *PTB* library has by far the lowest latencies and is strongly recommended (requires 64 bit Python 3.6+)
- The *pyo* library is, in theory, the highest performer, but in practice it has often had issues (at least on MacOS) with crashes and freezing of experiments, or causing them not to finish properly. If those issues aren't affecting your studies then this could be the one for you.
- The *sounddevice* library has performance that appears to be good (although this might be less so in cases where you have complex rendering being done as well because it operates from the same computer core as the main experiment code). It's newer than *pyo* and so more prone to bugs and we haven't yet added microphone support to record your participants.
- The *pygame* backend is the oldest and should work without errors, but has the least good performance. Use it if latencies for your audio don't matter.

Sounds are actually generated by a variety of classes, depending on which "backend" you use (like pyo or sounddevice) and these different backends can have slightly different attributes, as below. The user should typically do:

```
from psychopy.sound import Sound
```

The class that gets imported will then be an alias of one of the *Sound Classes* described below.

PTB audio latency

PTB brings a number of advantages in terms of latency.

The first is that is has been designed specifically with low-latency playback in mind (rather than, say, on-the-fly mixing and filtering capabilities). Mario Kleiner has worked very hard get the best out of the drivers available on each operating system and, as a result, with the most aggressive low-latency settings you can get a sound to play in "immediate" mode with typically in the region of 5ms lag and maybe 1ms precision. That's pretty good compared to the other options that have a lag of 20ms upwards and several ms variability.

BUT, on top of that, PTB allows you to *preschedule* your sound to occur at a particular point in time (e.g. when the trigger is due to be sent or when the screen is due to flip) and the PTB engine will then prepare all the buffers ready to go and will also account for the known latencies in the card. With this method the PTB engine is capable of sub-ms *precision* and even sub-ms *lag*!

Of course, *capable* doesn't mean it's happening in your case. It can depend on many things about the local operating system and hardware. You should test it yourself for your kit, but here is an example of a standard Win10 box using built-in audio (not a fancy audio card):

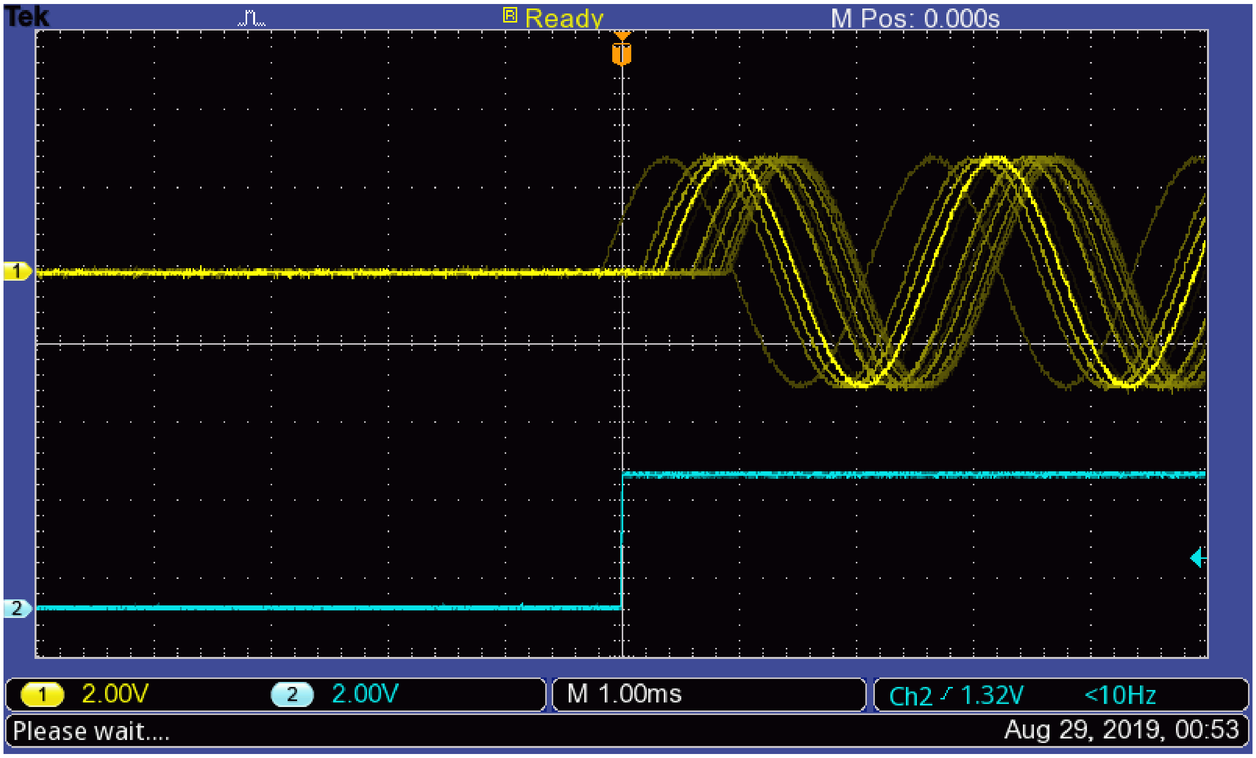


Fig. 11.1: Sub-ms audio timing with standard audio on Win10. Yellow trace is a 440 Hz tone played at 48 kHz with PTB engine. Cyan trace is the trigger (from a Labjack output). Gridlines are set to 1 ms.

Preschedule your sound

The most precise way to use the PTB audio backend is to preschedule the playing of a sound. By doing this PTB can actually take into account both the time taken to load the sound (it will preload ready) and also the time taken by the hardware to start playing it.

To do this you can call *play()* with an argument called *when*. The *when* argument needs to be in the PsychToolBox clock timebase which can be accessed by using *psychtoolbox.GetSecs()* if you want to play sound at an arbitrary time (not in sync with a window flip)

For instance:

```python
import psychtoolbox as ptb
from psychopy import sound

mySound = sound.Sound('A')
now = ptb.GetSecs()
mySound.play(when=now+0.5)  # play in EXACTLY 0.5s
```

or using *Window.getFutureFlipTime(clock='ptb')* if you want a synchronized time:

```python
import psychtoolbox as ptb
from psychopy import sound, visual

mySound = sound.Sound('A')

win = visual.Window()
win.flip()
nextFlip = win.getFutureFlipTime(clock='ptb')

mySound.play(when=nextFlip)  # sync with screen refresh
```

The precision of that timing is still dependent on the *PTB Audio Latency Modes* and can obviously not work if the delay before the requested time is not long enough for the requested mode (e.g. if you request that the sound starts on the next refresh but set the latency mode to be *0* (which has a lag of around 300 ms) then the timing will be very poor.

PTB Audio Latency Modes

When using the PTB backend you get the option to choose the Latency Mode, referred to in PsychToolBox as the *reqlatencyclass*, and can be set in `psychopy.hardware.speaker.SpeakerDevice`

uses Mode 3 in as a default, assuming that you want low latency and you don't care if other applications can't play sound at the same time (don't listen to iTunes while running your study!)

The modes are as follows:

0 : Latency not important
For when it really doesn't matter. Latency can easily be in the region of 300ms! The advantage of this move is that it will always work and always play a sound, whatever the format of the existing sounds that have been played (with 2, 3, 4 you can obtain low latency but the sampling rate must be the same throughout the experiment).

1 : Share low-latency access
Tries to use a low-latency setup in combination with other applications. Latency usually isn't very good and in MS Windows the sound you play must be the same sample rate as any other application that is using the sound system (which means you usually get restricted to exactly 48000 instead of 44100).

2 : Exclusive mode low-latency
Takes control of the audio device you're using and dominates it. That can cause some problems for other apps if they're trying to play sounds at the same time.

3 : Aggressive exclusive mode

As Mode 2 but with more aggressive settings to prioritise our use of the card over all others. **This is the recommended mode for most studies**

4 : Critical mode

As Mode 3 except that, if we fail to be totally dominant, then raise an error rather than just accepting our slightly less dominant status.

PTB Devices

To set the output audio device to use, you can set the *prefs.hardware['audioDevice']* setting. To determine the set of available devices, you can do for example:

```
from pprint import pprint
import psychtoolbox.audio
pprint(psychtoolbox.audio.get_devices())
```

Sound Classes

PTB Sound

`class psychopy.sound.backend_ptb.SoundPTB`(*value='C'*, *secs=0.5*, *octave=4*, *stereo=-1*, *volume=1.0*, *loops=0*, *sampleRate=None*, *blockSize=128*, *preBuffer=-1*, *hamming=True*, *startTime=0*, *stopTime=-1*, *name=''*, *autoLog=True*, *syncToWin=None*, *speaker=None*)

Play a variety of sounds using the new PsychPortAudio library

Parameters

- **value** – note name ("C","Bfl"), filename or frequency (Hz)
- **secs** – duration (for synthesised tones)
- **octave** – which octave to use for note names (4 is middle)
- **stereo** – -1 (auto), True or False to force sounds to stereo or mono
- **volume** – float 0-1
- **loops** – number of loops to play (-1=forever, 0=single repeat)
- **sampleRate** – sample rate for synthesized tones
- **blockSize** – the size of the buffer on the sound card (small for low latency, large for stability)
- **preBuffer** – integer to control streaming/buffering - -1 means store all - 0 (no buffer) means stream from disk - potentially we could buffer a few secs(!?)
- **hamming** – boolean (default True) to indicate if the sound should be apodized (i.e., the onset and offset smoothly ramped up from down to zero). The function apodize uses a Hanning window, but arguments named 'hamming' are preserved so that existing code is not broken by the change from Hamming to Hanning internally. Not applied to sounds from files.
- **startTime** – for sound files this controls the start of snippet
- **stopTime** – for sound files this controls the end of snippet
- **name** – string for logging purposes
- **autoLog** – whether to automatically log every change
- **syncToWin** – if you want start/stop to sync with win flips add this

`_EOS`(*reset=True*, *log=True*)

Function called on End Of Stream

`_channelCheck`(*array*)

Checks whether stream has fewer channels than data. If True, ValueError

`_checkPlaybackFinished`()

Checks whether playback has finished by looking up the status.

`property isFinished`

True if the audio playback has completed.

`property isPlaying`

True if the audio playback is ongoing.

`pause`(*log=True*)

Stops the sound without reset, so that play will continue from here if needed

`play`(*loops=None*, *when=None*, *log=True*)

Start the sound playing.

Calling this after the sound has finished playing will restart the sound.

`setSound`(*value*, *secs=0.5*, *octave=4*, *hamming=None*, *log=True*)

Set the sound to be played.

Often this is not needed by the user - it is called implicitly during initialisation.

Parameters

value: can be a number, string or an array:

- If it's a number between 37 and 32767 then a tone will be generated at that frequency in Hz.
- It could be a string for a note ('A', 'Bfl', 'B', 'C', 'Csh'. …). Then you may want to specify which octave.
- Or a string could represent a filename in the current location, or mediaLocation, or a full path combo
- Or by giving an Nx2 numpy array of floats (-1:1) you can specify the sound yourself as a waveform

secs: duration (only relevant if the value is a note name or
a frequency value)

octave: is only relevant if the value is a note name.
Middle octave of a piano is 4. Most computers won't output sounds in the bottom octave (1) and the top octave (8) is generally painful

`stop`(*reset=True*, *log=True*)

Stop the sound and return to beginning

`property stream`

Read-only property returns the stream on which the sound will be played

`property track`

The track on the master stream to which we belong

11.5.2 `Microphone` - for recording sound

The *Microphone* class provides an interface to audio recording devices connected to the computer. As of now, Psychtoolbox is required to use this feature and must be installed.

Overview

Microphone([device, sampleRateHz, channels, ...])	

Details

class psychopy.sound.microphone.**Microphone**(*device=None*, *sampleRateHz=None*, *channels=None*, *streamBufferSecs=2.0*, *maxRecordingSize=24000*, *policyWhenFull='warn'*, *exclusive=False*, *audioRunMode=0*, *name='mic'*, *recordingFolder=WindowsPath('C:/Users/psychopyci')*, *recordingExt='wav'*, *audioLatencyMode=None*)

property audioLatencyMode

bank(*tag=None*, *transcribe=False*, ***kwargs*)

Store current buffer as a clip within the microphone object.

This method is used internally by the Microphone component in Builder, don't use it for other applications. Either *stop()* or *pause()* must be called before calling this method.

Parameters

- **tag** (*str or None*) – Label for the clip.
- **transcribe** (*bool or str*) – Set to the name of a transcription engine (e.g. "GOOGLE") to transcribe using that engine, or set as *False* to not transcribe.
- **kwargs** (*dict*) – Additional keyword arguments to pass to `transcribe()`.

clear()

Wipe all clips. Deletes previously banked audio clips.

close()

flush()

Get a copy of all banked clips, then clear the clips from storage.

getClipFilename(*tag*, *i=0*)

Get the filename for a particular clip.

Parameters

- **tag** (*str*) – Tag assigned to the clip when *bank* was called
- **i** (*int*) – Index of clip within this tag (default is -1, i.e. the last clip)

Returns

Constructed filename for this clip

Return type

str

`getRecording()`

`property isRecBufferFull`

`property isRecording`

`property isStarted`

`property latencyBias`

`property maxRecordingSize`

`pause`(*blockUntilStopped=True*, *stopTime=None*)

`property policyWhenFull`

Until a file is saved, the audio data from a Microphone needs to be stored in RAM. To avoid a memory leak, we limit the amount which can be stored by a single Microphone object. The *policyWhenFull* parameter tells the Microphone what to do when it's reached that limit.

Parameters

`value` (*str*) – One of: - "ignore": When full, just don't record any new samples - "warn": Same as ignore, but will log a warning - "error": When full, will raise an error - "rolling": When full, clears the start of the buffer to make room for new samples

`poll()`

`property recBufferSecs`

`record`(*when=None*, *waitForStart=0*, *stopTime=None*)

`property recording`

`reopen()`

`saveClips`(*clear=True*)

Save all stored clips to audio files.

Parameters

`clear` (*bool*) – If True, clips will be removed from this object once saved to files.

`setMaxRecordingSize`(*value*)

`setMaxSize`(*value*)

`setPolicyWhenFull`(*value*)

Until a file is saved, the audio data from a Microphone needs to be stored in RAM. To avoid a memory leak, we limit the amount which can be stored by a single Microphone object. The *policyWhenFull* parameter tells the Microphone what to do when it's reached that limit.

Parameters

`value` (*str*) – One of: - "ignore": When full, just don't record any new samples - "warn": Same as ignore, but will log a warning - "error": When full, will raise an error - "rolling": When full, clears the start of the buffer to make room for new samples

`start`(*when=None*, *waitForStart=0*, *stopTime=None*)

`stop`(*blockUntilStopped=True*, *stopTime=None*)

`property streamBufferSecs`

`property streamStatus`

11.5.3 `AudioClip` - for working with audio data

Overview

AudioClip(samples[, sampleRateHz, userData])	Class for storing audio clip data.

Details

class psychopy.sound.**AudioClip**(*samples*, *sampleRateHz=48000*, *userData=None*)

Class for storing audio clip data.

This class is used to store and handle raw audio data, such as those obtained from microphone recordings or loaded from files. PsychoPy stores audio samples in contiguous arrays of 32-bit floating-point values ranging between -1 and 1.

The *AudioClip* class provides basic audio editing capabilities too. You can use operators on *AudioClip* instances to combine audio clips together. For instance, the + operator will return a new *AudioClip* instance whose samples are the concatenation of the two operands:

```
sndCombined = sndClip1 + sndClip2
```

Note that audio clips must have the same sample rates in order to be joined using the addition operator. For online compatibility, use the *append()* method instead.

There are also numerous static methods available to generate various tones (e.g., sine-, saw-, and square-waves). Audio samples can also be loaded and saved to files in various formats (e.g., WAV, FLAC, OGG, etc.)

You can play *AudioClip* by directly passing instances of this object to the `Sound` class:

```
import psychopy.core as core
import psychopy.sound as sound

myTone = AudioClip.sine(duration=5.0)  # generate a tone

mySound = sound.Sound(myTone)
mySound.play()
core.wait(5.0)  # wait for sound to finish playing
core.quit()
```

Parameters

- **samples** (*ArrayLike*) – Nx1 or Nx2 array of audio samples for mono and stereo, respectively. Values in the array representing the amplitude of the sound waveform should vary between -1 and 1. If not, they will be clipped.
- **sampleRateHz** (*int*) – Sampling rate used to obtain *samples* in Hertz (Hz). The sample rate or frequency is related to the quality of the audio, where higher sample rates usually result in better sounding audio (albeit a larger memory footprint and file size). The value specified should match the frequency the clip was recorded at. If not, the audio may sound distorted when played back. Usually, a sample rate of 48kHz is acceptable for most applications (DVD audio quality). For convenience, module level constants with form `SAMPLE_RATE_*` are provided to specify many common samples rates.
- **userData** (*dict or None*) – Optional user data to associated with the audio clip.

static _checkCodecSupported(*codec*, *raiseError=False*)

Check if the audio format string corresponds to a supported codec. Used internally to check if the user specified a valid codec identifier.

Parameters

- **codec** (*str*) – Codec identifier (e.g., 'wav', 'mp3', etc.)
- **raiseError** (*bool*) – Raise an error (``) instead of returning a value. Default is *False*.

Returns

True if the format is supported.

Return type

bool

append(*clip*)

Append samples from another sound clip to the end of this one.

The *AudioClip* object must have the same sample rate and channels as this object.

Parameters

clip (AudioClip) – Audio clip to append.

Returns

This object with samples from *clip* appended.

Return type

AudioClip

Examples

Join two sound clips together:

```
snd1.append(snd2)
```

asMono(*copy=True*)

Convert the audio clip to mono (single channel audio).

Parameters

copy (*bool*) – If *True* an *AudioClip* containing a copy of the samples will be returned. If *False*, channels will be mixed inplace resulting in the same object being returned. User data is not copied.

Returns

Mono version of this object.

Return type

AudioClip

asStereo(*copy=True*)

Convert the audio clip to stereo (two channel audio).

Parameters

copy (*bool*) – If *True* an *AudioClip* containing a copy of the samples will be returned. If *False*, channels will be mixed inplace resulting in the same object being returned. User data is not copied.

Returns

Stereo version of this object.

Return type
AudioClip

property channels

Number of audio channels in the clip (*int*).

If *channels* > 1, the audio clip is in stereo.

convertToWAV()

Get a copy of stored audio samples in WAV PCM format.

Returns
Array with the same shapes as *.samples* but in 16-bit WAV PCM format.

Return type
ndarray

copy()

Create an independent copy of this *AudioClip*.

Return type
AudioClip

property duration

The duration of the audio in seconds (*float*).

This value is computed using the specified sampling frequency and number of samples.

gain(*factor*, *channel=None*)

Apply gain the audio samples.

This will modify the internal store of samples inplace. Clipping is automatically applied to samples after applying gain.

Parameters

- **factor** (*float or int*) – Gain factor to multiply audio samples.
- **channel** (*int or None*) – Channel to apply gain to. If *None*, gain will be applied to all channels.

property isMono

True if there is only one channel of audio data.

property isStereo

True if there are two channels of audio samples.

Usually one for each ear. The first channel is usually the left ear, and the second the right.

static load(*filename*, *codec=None*)

Load audio samples from a file. Note that this is a static method!

Parameters

- **filename** (*str*) – File name to load.
- **codec** (*str or None*) – Codec to use. If *None*, the format will be implied from the file name.

Returns
Audio clip containing samples loaded from the file.

Return type
AudioClip

`resample`(*targetSampleRateHz*, *resampleType='default'*, *equalEnergy=False*, *copy=False*)

Resample audio to another sample rate.

This method will resample the audio clip to a new sample rate. The method used for resampling can be specified using the *method* parameter.

Parameters

- **targetSampleRateHz** (*int*) – New sample rate.
- **resampleType** (*str*) – Fitler (or method) to use for resampling. The methods available depend on the packages installed. The 'default' method uses *scipy.signal.resample* to resample the audio. Other methods require the user to install *librosa* or *resampy*. Default is 'default'.
- **equalEnergy** (*bool*) – Make the output have similar energy to the input. Option not available for the 'default' method. Default is *False*.
- **copy** (*bool*) – Return a copy of the resampled audio clip at the new sample rate. If *False*, the audio clip will be resampled inplace. Default is *False*.

Returns

Resampled audio clip.

Return type

AudioClip

Notes

- Resampling audio clip may result in distortion which is exacerbated by successive resampling.
- When using *librosa* for resampling, the *fix* parameter is set to *False*.
- The resampling types 'linear', 'zero_order_hold', 'sinc_best', 'sinc_medium' and 'sinc_fastest' require the *samplerate* package to be installed in addition to *librosa*.
- Specifying either the 'fft' or 'scipy' method will use the same resampling method as the 'default' method, howwever it will allow for the *equalEnergy* option to be used.

Examples

Resample an audio clip to 44.1kHz:

```
snd.resample(44100)
```

Use the 'soxr_vhq' method for resampling:

```
snd.resample(44100, resampleType='soxr_vhq')
```

Create a copy of the audio clip resampled to 44.1kHz:

```
sndResampled = snd.resample(44100, copy=True)
```

Resample the audio clip to be playable on a certain device:

```
import psychopy.sound as sound
from psychopy.sound.audioclip import AudioClip

audioClip = sound.AudioClip.load('/path/to/audio.wav')

```

(continues on next page)

(continued from previous page)

```
deviceSampleRateHz = sound.Sound().sampleRate
audioClip.resample(deviceSampleRateHz)
```

rms(*channel=None*)

Compute the root mean square (RMS) of the samples to determine the average signal level.

Parameters

channel (*int or None*) – Channel to compute RMS (zero-indexed). If *None*, the RMS of all channels will be computed.

Returns

An array of RMS values for each channel if `channel=None` (even if there is one channel an array is returned). If *channel was* specified, a *float* will be returned indicating the RMS of that single channel.

Return type

ndarray or float

property sampleRateHz

Sample rate of the audio clip in Hz (*int*). Should be the same value as the rate *samples* was captured at.

property samples

Nx1 or Nx2 array of audio samples (*~numpy.ndarray*).

Values must range from -1 to 1. Values outside that range will be clipped, possibly resulting in distortion.

save(*filename*, *codec=None*)

Save an audio clip to file.

Parameters

- **filename** (*str*) – File name to write audio clip to.
- **codec** (*str or None*) – Format to save audio clip data as. If *None*, the format will be implied from the extension at the end of *filename*.

static sawtooth(*duration=1.0*, *freqHz=440*, *peak=1.0*, *gain=0.8*, *sampleRateHz=48000*, *channels=2*)

Generate audio samples for a tone with a sawtooth waveform.

Parameters

- **duration** (*float or int*) – Length of the sound in seconds.
- **freqHz** (*float or int*) – Frequency of the tone in Hertz (Hz). Note that this differs from the *sampleRateHz*.
- **peak** (*float*) – Location of the peak between 0.0 and 1.0. If the peak is at 0.5, the resulting wave will be triangular. A value of 1.0 will cause the peak to be located at the very end of a cycle.
- **gain** (*float*) – Gain factor ranging between 0.0 and 1.0. Default is 0.8.
- **sampleRateHz** (*int*) – Samples rate of the audio for playback.
- **channels** (*int*) – Number of channels for the output.

Return type

AudioClip

static silence(*duration=1.0*, *sampleRateHz=48000*, *channels=2*)

Generate audio samples for a silent period.

This is used to create silent periods of a very specific duration between other audio clips.

Parameters

- **duration** (*float or int*) – Length of the sound in seconds.
- **sampleRateHz** (*int*) – Samples rate of the audio for playback.
- **channels** (*int*) – Number of channels for the output.

Return type

AudioClip

Examples

Generate 10 seconds of silence to enjoy:

```
import psychopy.sound as sound
silence = sound.AudioClip.silence(10.)
```

Use the silence as a break between two audio clips when concatenating them:

```
fullClip = clip1 + sound.AudioClip.silence(10.) + clip2
```

static sine(*duration=1.0*, *freqHz=440*, *gain=0.8*, *sampleRateHz=48000*, *channels=2*)

Generate audio samples for a tone with a sine waveform.

Parameters

- **duration** (*float or int*) – Length of the sound in seconds.
- **freqHz** (*float or int*) – Frequency of the tone in Hertz (Hz). Note that this differs from the *sampleRateHz*.
- **gain** (*float*) – Gain factor ranging between 0.0 and 1.0. Default is 0.8.
- **sampleRateHz** (*int*) – Samples rate of the audio for playback.
- **channels** (*int*) – Number of channels for the output.

Return type

AudioClip

Examples

Generate an audio clip of a tone 10 seconds long with a frequency of 400Hz:

```
import psychopy.sound as sound
tone400Hz = sound.AudioClip.sine(10., 400.)
```

Create a marker/cue tone and append it to pre-recorded instructions:

```
import psychopy.sound as sound
voiceInstr = sound.AudioClip.load('/path/to/instructions.wav')
markerTone = sound.AudioClip.sine(
    1.0, 440.,  # duration and freq
    sampleRateHz=voiceInstr.sampleRateHz)  # must be the same!
```

(continues on next page)

(continued from previous page)

```
fullInstr = voiceInstr + markerTone  # create instructions with cue
fullInstr.save('/path/to/instructions_with_tone.wav')  # save it
```

static square(*duration=1.0*, *freqHz=440*, *dutyCycle=0.5*, *gain=0.8*, *sampleRateHz=48000*, *channels=2*)

Generate audio samples for a tone with a square waveform.

Parameters

- **duration** (*float or int*) – Length of the sound in seconds.
- **freqHz** (*float or int*) – Frequency of the tone in Hertz (Hz). Note that this differs from the *sampleRateHz*.
- **dutyCycle** (*float*) – Duty cycle between 0.0 and 1.0.
- **gain** (*float*) – Gain factor ranging between 0.0 and 1.0. Default is 0.8.
- **sampleRateHz** (*int*) – Samples rate of the audio for playback.
- **channels** (*int*) – Number of channels for the output.

Return type

AudioClip

static synthesizeSpeech(*text*, *engine='gtts'*, *synthConfig=None*, *outFile=None*)

Synthesize speech from text using a text-to-speech (TTS) engine.

This method is used to generate audio samples from text using a text-to-speech (TTS) engine. The synthesized speech can be used for various purposes, such as generating audio cues for experiments or creating audio instructions for participants.

This method returns an *AudioClip* object containing the synthesized speech. The quality and format of the retured audio may vary depending on the TTS engine used.

Please note that online TTS engines may require an active internet connection to work. This also may send the text to a remote server for processing, so be mindful of privacy concerns.

Parameters

- **text** (*str*) – Text to synthesize into speech.
- **engine** (*str*) – TTS engine to use for speech synthesis. Default is 'gtts'.
- **synthConfig** (*dict or None*) – Additional configuration options for the specified engine. These are specified using a dictionary (ex. *synthConfig={'slow': False}*). These paramters vary depending on the engine in use. Default is *None* which uses the default configuration for the engine.
- **outFile** (*str or None*) – File name to save the synthesized speech to. This can be used to save the audio to a file for later use. If *None*, the audio clip will be returned in memory. If you plan on using the same audio clip multiple times, it is recommended to save it to a file and load it later.

Returns

Audio clip containing the synthesized speech.

Return type

AudioClip

Examples

Synthesize speech using the default gTTS engine:

```python
import psychopy.sound as sound
voiceClip = sound.AudioClip.synthesizeSpeech(
    'How are you doing today?')
```

Save the synthesized speech to a file for later use:

```python
voiceClip = sound.AudioClip.synthesizeSpeech(
    'How are you doing today?', outFile='/path/to/speech.mp3')
```

Synthesize speech using the gTTS engine with a specific language, timeout, and top-level domain:

```python
voiceClip = sound.AudioClip.synthesizeSpeech(
    'How are you doing today?',
    engine='gtts',
    synthConfig={'lang': 'en', 'timeout': 10, 'tld': 'us'})
```

`transcribe`(*engine='whisper'*, *language='en-US'*, *expectedWords=None*, *config=None*)

Convert speech in audio to text.

This function accepts an audio clip and returns a transcription of the speech in the clip. The efficacy of the transcription depends on the engine selected, audio quality, and language support.

Speech-to-text conversion blocks the main application thread when used on Python. Don't transcribe audio during time-sensitive parts of your experiment! Instead, initialize the transcriber before the experiment begins by calling this function with *audioClip=None*.

Parameters

- **engine** (*str*) – Speech-to-text engine to use.
- **language** (*str*) – BCP-47 language code (eg., 'en-US'). Note that supported languages vary between transcription engines.
- **expectedWords** (*list or tuple*) – List of strings representing expected words or phrases. This will constrain the possible output words to the ones specified which constrains the model for better accuracy. Note not all engines support this feature (only Sphinx and Google Cloud do at this time). A warning will be logged if the engine selected does not support this feature. CMU PocketSphinx has an additional feature where the sensitivity can be specified for each expected word. You can indicate the sensitivity level to use by putting a `:` after each word in the list (see the Example below). Sensitivity levels range between 0 and 100. A higher number results in the engine being more conservative, resulting in a higher likelihood of false rejections. The default sensitivity is 80% for words/phrases without one specified.
- **config** (*dict or None*) – Additional configuration options for the specified engine. These are specified using a dictionary (ex. *config={'pfilter': 1}* will enable the profanity filter when using the *'google'* engine).

Returns

Transcription result.

Return type

TranscriptionResult

Notes

- The recommended transcriber is OpenAI Whisper which can be used locally without an internet connection once a model is downloaded to cache. It can be selected by passing *engine='whisper'* to this function.
- Online transcription services (eg., Google) provide robust and accurate speech recognition capabilities with broader language support than offline solutions. However, these services may require a paid subscription to use, reliable broadband internet connections, and may not respect the privacy of your participants as their responses are being sent to a third-party. Also consider that a track of audio data being sent over the network can be large, users on metered connections may incur additional costs to run your experiment. Offline transcription services (eg., CMU PocketSphinx and OpenAI Whisper) do not require an internet connection after the model has been downloaded and installed.
- If the audio clip has multiple channels, they will be combined prior to being passed to the transcription service if needed.

property userData

User data associated with this clip (*dict*). Can be used for storing additional data related to the clip. Note that *userData* is not saved with audio files!

Example

Adding fields to *userData*. For instance, we want to associated the start time the clip was recorded at with it:

```
myClip.userData['date_recorded'] = t_start
```

We can access that field later by:

```
thisRecordingStartTime = myClip.userData['date_recorded']
```

static whiteNoise(*duration=1.0*, *sampleRateHz=48000*, *channels=2*)

Generate gaussian white noise.

New feature, use with caution.

Parameters

- **duration** (*float or int*) – Length of the sound in seconds.
- **sampleRateHz** (*int*) – Samples rate of the audio for playback.
- **channels** (*int*) – Number of channels for the output.

Return type

AudioClip

11.5.4 AudioDeviceInfo and AudioDeviceStatus - descriptors for audio devices

These classes are used to store information about audio devices and their status. Only a subset of PsychoPy's sound API currently use these classes, such as the *psychopy.sound.microphone.Microphone* class.

Overview

AudioDeviceInfo([deviceIndex, deviceName, ...])	Descriptor for an audio device (playback or recording) on this system.
AudioDeviceStatus([active, state, ...])	Descriptor for audio device status information.

Details

class psychopy.sound.**AudioDeviceInfo**(*deviceIndex=-1*, *deviceName='Null Device'*, *hostAPIName='Null Audio Driver'*, *outputChannels=0*, *outputLatency=(0.0, 0.0)*, *inputChannels=0*, *inputLatency=(0.0, 0.0)*, *defaultSampleRate=48000*, *audioLib=''*)

Descriptor for an audio device (playback or recording) on this system.

Properties associated with this class provide information about a specific audio playback or recording device. An object can be then passed to `Microphone` to open a stream using the device described by the object.

This class is usually instanced only by calling `getDevices()`. Users should avoid creating instances of this class themselves unless they have good reason to.

Parameters

- **deviceIndex** (*int*) – Enumerated index of the audio device. This number is specific to the engine used for audio.
- **deviceName** (*str*) – Human-readable name of the device.
- **hostAPIName** (*str*) – Human-readable name of the host API used for audio.
- **outputChannels** (*int*) – Number of output channels.
- **outputLatency** (*tuple*) – Low (*float*) and high (*float*) output latency in milliseconds.
- **inputChannels** (*int*) – Number of input channels.
- **inputLatency** (*tuple*) – Low (*float*) and high (*float*) input latency in milliseconds.
- **defaultSampleRate** (*int*) – Default sample rate for the device in Hertz (Hz).
- **audioLib** (*str*) – Audio library that queried device information used to populate the properties of this descriptor (e.g., `'ptb'` for Psychtoolbox).

Examples

Get a list of available devices:

```
import psychopy.sound as sound
recordingDevicesList = sound.Microphone.getDevices()
```

Get the low and high input latency of the first recording device:

```
recordingDevice = recordingDevicesList[0]  # assume not empty
inputLatencyLow, inputLatencyHigh = recordingDevice.inputLatency
```

Get the device name as it may appear in the system control panel or sound settings:

```
deviceName = recordingDevice.deviceName
```

Specifying the device to use for capturing audio from a microphone:

```
# get the first suitable capture device found by the sound engine
recordingDevicesList = sound.Microphone.getDevices()
recordingDevice = recordingDevicesList[0]

# pass the descriptor to microphone to configure it
mic = sound.Microphone(device=recordingDevice)
mic.start()  # start recording sound
```

`property audioLib`

Audio library used to query device information (*str*).

`static createFromPTBDesc(`*desc*`)`

Create an *AudioDevice* instance with values populated using a descriptor (*dict*) returned from the PTB *audio.get_devices* API call.

Parameters
`desc` (`dict`) – Audio device descriptor returned from Psychtoolbox's *get_devices* function.

Returns
Audio device descriptor with properties set using *desc*.

Return type
AudioDeviceInfo

`property defaultSampleRate`

Default sample rate in Hertz (Hz) for this device (*int*).

`property deviceIndex`

Enumerated index (*int*) of the audio device.

`property deviceName`

Human-readable name (*str*) for the audio device reported by the driver.

`property hostAPIName`

Human-readable name (*str*) for the host API.

`property inputChannels`

Number of input channels (*int*). If >0, this is likely a audio capture device.

`property inputLatency`

Low and high input latency in milliseconds *(low, high)*.

`property isCapture`

True if this device is suitable for capture (*bool*).

`property isDuplex`

True if this device is suitable for capture and playback (*bool*).

`property isPlayback`

True if this device is suitable for playback (*bool*).

`property outputChannels`

Number of output channels (*int*). If >0, this is likely a audio playback device.

`property outputLatency`

Low and high output latency in milliseconds *(low, high)*.

`class psychopy.sound.AudioDeviceStatus(`*active=0*, *state=0*, *requestedStartTime=0.0*, *startTime=0.0*, *captureStartTime=0.0*, *requestedStopTime=0.0*, *estimatedStopTime=0.0*, *currentStreamTime=0.0*, *elapsedOutSamples=0*, *positionSecs=0.0*, *recordedSecs=0.0*, *readSecs=0.0*, *schedulePosition=0.0*, *xRuns=0*, *totalCalls=0*, *timeFailed=0*, *bufferSize=0*, *cpuLoad=0.0*, *predictedLatency=0.0*, *latencyBias=0.0*, *sampleRate=44100*, *outDeviceIndex=0*, *inDeviceIndex=0*, *audioLib='Null Audio Library'*`)`

Descriptor for audio device status information.

Properties of this class are standardized on the status information returned by Psychtoolbox. Other audio backends should try to populate these fields as best they can with their equivalent status values.

Users should never instance this class themselves unless they have good reason to.

Parameters

- **active** (*bool*) – *True* if playback or recording has started, else *False*.
- **state** (*int*) – State of the device, either *1* for playback, *2* for recording or *3* for duplex (recording and playback).
- **requestedStartTime** (*float*) – Requested start time of the audio stream after the start of playback or recording.
- **startTime** (*float*) – The actual (real) start time of audio playback or recording.
- **captureStartTime** (*float*) – Estimate of the start time of audio capture. Only valid if audio capture is active. Usually, this time corresponds to the time when the first sound was captured.
- **requestedStopTime** (*float*) – Stop time requested when starting the stream.
- **estimatedStopTime** (*float*) – Estimated stop time given *requestedStopTime*.
- **currentStreamTime** (*float*) – Estimate of the time it will take for the most recently submitted sample to reach the speaker. Value is in absolute system time and reported for playback only.
- **elapsedOutSamples** (*int*) – Total number of samples submitted since the start of playback.
- **positionSecs** (*float*) – Current stream playback position in seconds this loop. Does not account for hardware of driver latency.
- **recordedSecs** (*float*) – Total amount of recorded sound data (in seconds) since start of capture.
- **readSecs** (*float*) – Total amount of sound data in seconds that has been fetched from the internal buffer.
- **schedulePosition** (*float*) – Current position in a running schedule in seconds.
- **xRuns** (*int*) – Number of dropouts due to buffer over- and under-runs. Such conditions can result is glitches during playback/recording. Even if the number remains zero, that does not mean that glitches did not occur.
- **totalCalls** (*int*) – **Debug** - Used for debugging the audio engine.
- **timeFailed** (*float*) – **Debug** - Used for debugging the audio engine.
- **bufferSize** (*int*) – **Debug** - Size of the buffer allocated to contain stream samples. Used for debugging the audio engine.
- **cpuLoad** (*float*) – Amount of load on the CPU imparted by the sound engine. Ranges between 0.0 and 1.0 where 1.0 indicates maximum load on the core running the sound engine process.
- **predictedLatency** (*float*) – Latency for the given hardware and driver. This indicates how far ahead you need to start the device to ensure is starts at a scheduled time.
- **latencyBias** (*float*) – Additional latency bias added by the user.
- **sampleRate** (*int*) – Sample rate in Hertz (Hz) the playback recording is using.

- **outDeviceIndex** (*int*) – Enumerated index of the output device.
- **inDeviceIndex** (*int*) – Enumerated index of the input device.
- **audioLib** (*str*) – Identifier for the audio library which created this status.

`property active`

True if playback or recording has started (*bool*).

`property audioLib`

Identifier for the audio library which created this status (*str*).

`property bufferSize`

Debug - Size of the buffer allocated to contain stream samples. Used for debugging the audio engine.

`property captureStartTime`

Estimate of the start time of audio capture (*float*). Only valid if audio capture is active. Usually, this time corresponds to the time when the first sound was captured.

`property cpuLoad`

Amount of load on the CPU imparted by the sound engine (*float*). Ranges between 0.0 and 1.0 where 1.0 indicates maximum load on the core running the sound engine process.

`static createFromPTBDesc(desc)`

Create an *AudioDeviceStatus* instance using a status descriptor returned by Psychtoolbox.

Parameters
desc (*dict*) – Audio device status descriptor.

Returns
Audio device descriptor with properties set using *desc*.

Return type
AudioDeviceStatus

`property currentStreamTime`

Estimate of the time it will take for the most recently submitted sample to reach the speaker (*float*). Value is in absolute system time and reported for playback mode only.

`property elapsedOutSamples`

Total number of samples submitted since the start of playback (*int*).

`property estimatedStopTime`

Estimated stop time given *requestedStopTime* (*float*).

`property inDeviceIndex`

Enumerated index of the input device (*int*).

`property isCapture`

True if this device is operating in capture mode (*bool*).

`property isDuplex`

True if this device is operating capture and recording mode (*bool*).

`property isPlayback`

True if this device is operating in playback mode (*bool*).

`property latencyBias`

Additional latency bias added by the user (*float*).

property outDeviceIndex

Enumerated index of the output device (*int*).

property positionSecs

Current stream playback position in seconds this loop (*float*). Does not account for hardware of driver latency.

property predictedLatency

Latency for the given hardware and driver (*float*). This indicates how far ahead you need to start the device to ensure is starts at a scheduled time.

property readSecs

Total amount of sound data in seconds that has been fetched from the internal buffer (*float*).

property recordedSecs

Total amount of recorded sound data (in seconds) since start of capture (*float*).

property requestedStartTime

Requested start time of the audio stream after the start of playback or recording (*float*).

property requestedStopTime

Stop time requested when starting the stream (*float*).

property sampleRate

Sample rate in Hertz (Hz) the playback recording is using (*int*).

property schedulePosition

Current position in a running schedule in seconds (*float*).

property startTime

The actual (real) start time of audio playback or recording (*float*).

property state

State of the device (*int*). Either *1* for playback, *2* for recording or *3* for duplex (recording and playback).

property timeFailed

Debug - Used for debugging the audio engine (*float*).

property totalCalls

Debug - Used for debugging the audio engine (*int*).

property xRuns

Number of dropouts due to buffer over- and under-runs (*int*). Such conditions can result is glitches during playback/recording. Even if the number remains zero, that does not mean that glitches did not occur.

11.5.5 TranscriptionResult - results of an audio transcription

Overview

TranscriptionResult(words, unknownValue, ...)	Descriptor for returned transcription data.

Details

`class psychopy.sound.transcribe.TranscriptionResult`(*words*, *unknownValue*, *requestFailed*, *engine*, *language*)

Descriptor for returned transcription data.

Fields within this class can be used to access transcribed words and other information related to the transcription request.

This is returned by functions and methods which perform speech-to-text transcription from audio data within PsychoPy. The user usually does not create instances of this class themselves.

Parameters

- `words` (*list of str*) – Words extracted from the audio clip.
- `unknownValue` (*bool*) – *True* if the transcription API failed make sense of the audio and did not complete the transcription.
- `requestFailed` (*bool*) – *True* if there was an error with the transcriber itself. For instance, network error or improper formatting of the audio data.
- `engine` (*str*) – Name of engine used to perform this transcription.
- `language` (*str*) – Identifier for the language used to perform the transcription.

`property engine`

Name of engine used to perform this transcription (*str*).

`property error`

True if there was an error during transcription (*bool*). Value is always the compliment of *.success*.

`getSpeechInterval()`

Get the start and stop times for the interval of speech in the audio clip.

This feature is only supported by the Whisper transcriber. The start and end times of the speech interval are returned in seconds.

Returns

Start and end times of the speech interval in seconds. If the engine does not support this feature, or if the data is missing, *(None, None)* is returned. In cases where either the start or end time is missing, the value will be *None* for that field.

Return type

tuple

`property language`

Identifier for the language used to perform the transcription (*str*).

`property requestFailed`

True if there was an error with the transcriber itself (*bool*). For instance, network error or improper formatting of the audio data, invalid key, or if there was network connection error.

`property response`

Raw API response from the transcription engine (*str*).

`property responseData`

Values from self.response, parsed into a *dict*.

`property success`

True if the transcriber returned a result successfully (*bool*).

property text

Text transcribed for the audio data (*str*).

property unknownValue

True if the transcription API failed make sense of the audio and did not complete the transcription (*bool*).

property wordCount

Number of words found (*int*).

property wordData

Additional data about each word (*list*).

Not all engines provide this data in the same format or at all.

property words

Words extracted from the audio clip (*list* of *str*).

11.6 `psychopy.hardware` - hardware interfaces

can access a wide range of external hardware. For some devices the interface has already been created in the following sub-packages of . For others you may need to write the code to access the serial port etc. manually.

Contents:

11.6.1 Keyboard

To handle input from keyboard (supersedes event.getKeys)

The Keyboard class was new in PsychoPy 3.1 and replaces the older *event.getKeys()* calls.

Psychtoolbox versus event.getKeys

On 64 bits Python3 installations it provides access to the Psychtoolbox kbQueue series of functions using the same compiled C code (available in python-psychtoolbox lib).

On 32 bit installations and Python2 it reverts to the older `psychopy.event.getKeys()` calls.

The new calls have several advantages:

- the polling is performed and timestamped asynchronously with the main thread so that times relate to when the key was pressed, not when the call was made
- the polling is direct to the USB HID library in C, which is faster than waiting for the operating system to poll and interpret those same packets
- we also detect the KeyUp events and therefore provide the option of returning keypress duration
- on Linux and Mac you can also distinguish between different keyboard devices (see `getKeyboards()`)

This library makes use, where possible of the same low-level asynchronous hardware polling as in Psychtoolbox

Example usage

```
from psychopy.hardware import keyboard
from psychopy import core

kb = keyboard.Keyboard()

```

(continues on next page)

(continued from previous page)

```
# during your trial
kb.clock.reset()  # when you want to start the timer from
keys = kb.getKeys(['right', 'left', 'quit'], waitRelease=True)
if 'quit' in keys:
    core.quit()
for key in keys:
    print(key.name, key.rt, key.duration)
```

Classes and functions

`class psychopy.hardware.keyboard.`**`Keyboard`**(*deviceName=None*, *device=-1*, *bufferSize=10000*, *waitForStart=False*, *clock=None*, *backend=None*)

`getState`(*keys*)

Get the current state of a key or set of keys

Parameters

keys (*`str` or `list[str]`*) – Either the code for a single key, or a list of key codes.

Returns

keys – True if pressed, False if not. Will be a single value if given a single key, or a list of bools if given a list of keys.

Return type

bool or list[bool]

`class psychopy.hardware.keyboard.`**`KeyPress`**(*code*, *tDown*, *name=None*)

Class to store key presses, as returned by *Keyboard.getKeys()*

Unlike keypresses from the old event.getKeys() which returned a list of strings (the names of the keys) we now return several attributes for each key:

> .name: the name as a string (matching the previous pyglet name) .rt: the reaction time (relative to last clock reset) .tDown: the time the key went down in absolute time .duration: the duration of the keypress (or None if not released)

Although the keypresses are a class they will test ==, *!=* and *in* based on their name. So you can still do:

```
kb = KeyBoard()
# wait for keypresses here
keys = kb.getKeys()
for thisKey in keys:
    if thisKey=='q':  # it is equivalent to the string 'q'
        core.quit()
    else:
        print(thisKey.name, thisKey.tDown, thisKey.rt)
```

`psychopy.hardware.keyboard.`**`getKeyboards`**()

Get info about the available keyboards.

Only really useful on Mac/Linux because on these the info can be used to select a particular physical device when calling *`Keyboard`*. On Win this function does return information correctly but the :class:Keyboard can't make use of it.

Returns

USB Info including with name, manufacturer, id, etc for each device

Return type

A list of dicts

11.6.2 BrainProducts

Interfaces for Brain Products GMBH hardware.

Here we have implemented support for the Remote Control Server application, which allows you to control recordings, send annotations etc. all from Python.

`class psychopy.hardware.brainproducts.`**`RemoteControlServer`**`(`**args*, ***kwargs*`)`

psychopy.hardware.brainproducts is now located within the psychopy-brainproducts plugin.

When initialised, rather than creating an object, will log an error.

11.6.3 Camera

Classes and functions for reading and writing camera streams.

A camera may be used to document participant responses on video or used by the experimenter to create movie stimuli or instructions.

11.6.4 Overview

11.6.5 Details

`class psychopy.hardware.camera.`**`Camera`**`(`*device=0*, *mic=None*, *cameraLib='ffpyplayer'*, *frameRate=None*, *frameSize=None*, *bufferSecs=4*, *win=None*, *name='cam'*`)`

Class for displaying and recording video from a USB/PCI connected camera.

This class is capable of opening, recording, and saving camera video streams to disk. Camera stream reading/writing is done in a separate thread, allowing capture to occur in the background while the main thread is free to perform other tasks. This allows for capture to occur at higher frame rates than the display refresh rate. Audio recording is also supported if a microphone interface is provided, where recording will be synchronized with the video stream (as best as possible). Video and audio can be saved to disk either as a single file or as separate files.

GNU/Linux is supported only by the OpenCV backend (*cameraLib='opencv'*).

Parameters

- **`device`** (*`str or int`*) – Camera to open a stream with. If the ID is not valid, an error will be raised when *open()* is called. Value can be a string or number. String values are platform-dependent: a DirectShow URI or camera name on Windows, or a camera name/index on MacOS. Specifying a number (>=0) is a platform-independent means of selecting a camera. PsychoPy enumerates possible camera devices and makes them selectable without explicitly having the name of the cameras attached to the system. Use caution when specifying an integer, as the same index may not reference the same camera every time.
- **`mic`** (*`Microphone`* or None) – Microphone to record audio samples from during recording. The microphone input device must not be in use when *record()* is called. The audio track will be merged with the video upon calling *save()*. Make sure that *Microphone.maxRecordingSize* is specified to a reasonable value to prevent the audio track from being truncated. Specifying a microphone adds some latency to starting and stopping camera recording due to the added overhead involved with synchronizing the audio and video streams.
- **`frameRate`** (*`int or None`*) – Frame rate to record the camera stream at. If *None*, the camera's default frame rate will be used.

- **frameSize** (*tuple or None*) – Size (width, height) of the camera stream frames to record. If *None*, the camera's default frame size will be used.
- **cameraLib** (*str*) – Interface library (backend) to use for accessing the camera. May either be *ffpyplayer* or *opencv*. If *None*, the default library for the recommended by the PsychoPy developers will be used. Switching camera libraries could help resolve issues with camera compatibility. More camera libraries may be installed via extension packages.
- **bufferSecs** (*float*) – Size of the real-time camera stream buffer specified in seconds (only valid on Windows and MacOS). This is not the same as the recording buffer size. This option might not be available for all camera libraries.
- **win** (*Window* or None) – Optional window associated with this camera. Some functionality may require an OpenGL context for presenting frames to the screen. If you are not planning to display the camera stream, this parameter can be safely ignored.
- **name** (*str*) – Label for the camera for logging purposes.

Examples

Opening a camera stream and closing it:

```
camera = Camera(device=0)
camera.open()  # exception here on invalid camera
camera.close()
```

Recording 5 seconds of video and saving it to disk:

```
cam = Camera(0)
cam.open()
cam.record()  # starts recording

while cam.recordingTime < 5.0:  # record for 5 seconds
    if event.getKeys('q'):
        break
    cam.update()

cam.stop()  # stops recording
cam.save('myVideo.mp4')
cam.close()
```

Providing a microphone as follows enables audio recording:

```
mic = Microphone(0)
cam = Camera(0, mic=mic)
```

Overriding the default frame rate and size (if *cameraLib* supports it):

```
cam = Camera(0, frameRate=30, frameSize=(640, 480), cameraLib=u'opencv')
```

_assertCameraReady()

Assert that the camera is ready. Raises a *CameraNotReadyError* if the camera is not ready.

_assertMediaPlayer()

Assert that we have a media player instance open.

This will raise a *RuntimeError* if there is no player open. Use this function to ensure that a player is present before running subsequent code.

`_download()`

Download video file to an online repository. Not implemented locally, needed for auto translate to JS.

`_enqueueFrame()`

Pull waiting frames from the capture thread.

This function will pull frames from the capture thread and add them to the buffer. The last frame in the buffer will be set as the most recent frame (*lastFrame*).

Returns

True if a frame has been enqueued. Returns *False* if the camera is not ready or if the stream was closed.

Return type

bool

`_upload()`

Upload video file to an online repository. Not implemented locally, needed for auto translate to JS.

`authorize()`

Get permission to access the camera. Not implemented locally yet.

`close()`

Close the camera.

This will close the camera stream and free up any resources used by the device. If the camera is currently recording, this will stop the recording, but will not discard any frames. You may still call *save()* to save the frames to disk.

`property device`

Camera to use (*str* or *None*).

String specifying the name of the camera to open a stream with. This must be set prior to calling *start()*. If the name is not valid, an error will be raised when *start()* is called.

`property frameCount`

Number of frames captured in the present recording (*int*).

`property frameRate`

Frame rate of the video stream (*float* or *None*).

Only valid after an *open()* and successive *_enqueueFrame()* call as metadata needs to be obtained from the stream. Returns *None* if not valid.

`property frameSize`

Size of the video frame obtained from recent metadata (*float* or *None*).

Only valid after an *open()* and successive *_enqueueFrame()* call as metadata needs to be obtained from the stream. Returns *None* if not valid.

`static getCameraDescriptions`(*collapse=False*)

Get a mapping or list of camera descriptions.

Camera descriptions are a compact way of representing camera settings and formats. Description strings can be used to specify which camera device and format to use with it to the *Camera* class.

Descriptions have the following format (example):

```
'[Live! Cam Sync 1080p] 160x120@30fps, mjpeg'
```

This shows a specific camera format for the 'Live! Cam Sync 1080p' webcam which supports 160x120 frame size at 30 frames per second. The last value is the codec or pixel format used to decode the stream. Different pixel formats and codecs vary in performance.

Parameters
`collapse` (*bool*) – Return camera information as string descriptions instead of *CameraInfo* objects. This provides a more compact way of representing camera formats in a (reasonably) human-readable format.

Returns
Mapping (*dict*) of camera descriptions, where keys are camera names (*str*) and values are a *list* of format description strings associated with the camera. If *collapse=True*, all descriptions will be returned in a single flat list. This might be more useful for specifying camera formats from a single GUI list control.

Return type
dict or list

`static getCameras(`*cameraLib=None*`)`

Get information about installed cameras on this system.

Returns
Mapping of camera information objects.

Return type
dict

`getLastClip()`

File path to the last saved recording.

This value is only valid if a previous recording has been saved to disk (*save()* was called).

Returns
Path to the file the most recent call to *save()* created. Returns *None* if no file is ready.

Return type
str or None

`getMetadata()`

Get stream metadata.

Returns
Metadata about the video stream, retrieved during the last frame update (*_enqueueFrame* call).

Return type
MovieMetadata

`getVideoFrame()`

Pull the most recent frame from the stream (if available).

Returns
Most recent video frame. Returns *NULL_MOVIE_FRAME_INFO* if no frame was available, or we timed out.

Return type
MovieFrame

`property isNotStarted`

True if the stream may not have started yet (*bool*). This status is given before *open()* or after *close()* has been called on this object.

property isReady

Is the camera ready (*bool*)?

The camera is ready when the following conditions are met. First, we've created a player interface and opened it. Second, we have received metadata about the stream. At this point we can assume that the camera is 'hot' and the stream is being read.

This is a legacy property used to support older versions of PsychoPy. The *isOpened* property should be used instead.

property isRecording

True if the video is presently recording (*bool*).

property isStarted

True if the stream has started (*bool*). This status is given after *open()* has been called on this object.

property isStopped

True if the recording has stopped (*bool*). This does not mean that the stream has stopped, *getVideoFrame()* will still yield frames until *close()* is called.

property lastClip

File path to the last recording (*str* or *None*).

This value is only valid if a previous recording has been saved successfully (*save()* was called), otherwise it will be set to *None*.

property lastFrame

Most recent frame pulled from the camera (*VideoFrame*) since the last call of *getVideoFrame*.

property metadata

Video metadata retrieved during the last frame update (*MovieMetadata*).

property mic

Microphone to record audio samples from during recording (`Microphone` or *None*).

If *None*, no audio will be recorded. Cannot be set after opening a camera stream.

open()

Open the camera stream and begin decoding frames (if available).

This function returns when the camera is ready to start getting frames.

Call *record()* to start recording frames to memory. Captured frames came be saved to disk using *save()*.

record(*clearLastRecording=True*)

Start recording frames.

This function will start recording frames and audio (if available). The value of *lastFrame* will be updated as new frames arrive and the *frameCount* will increase. You can access image data for the most recent frame to be captured using *lastFrame*.

If this is called before *open()* the camera stream will be opened automatically. This is not recommended as it may incur a longer than expected delay in the recording start time.

> ⚠ **Warning**
>
> If a recording has been previously made without calling *save()* it will be discarded if *record()* is called again unless *clearLastRecording=False*.

Parameters

clearLastRecording (*bool*) – Clear the frame buffer before starting the recording. If *True*, the frame buffer will be cleared before starting the recording. If *False*, the frame buffer will be kept and new frames will be added to the buffer. Default is *True*.

property recordingBytes

Current size of the recording in bytes (*int*).

property recordingTime

Current recording timestamp (*float*).

This returns the timestamp of the last frame captured in the recording.

This value increases monotonically from the last *record()* call. It will reset once *stop()* is called. This value is invalid outside *record()* and *stop()* calls.

save(*filename*, *useThreads=True*, *mergeAudio=True*, *encoderLib=None*, *encoderOpts=None*)

Save the last recording to file.

This will write frames to *filename* acquired since the last call of *record()* and subsequent *stop()*. If *record()* is called again before *save()*, the previous recording will be deleted and lost.

This is a slow operation and will block for some time depending on the length of the video. This can be sped up by setting *useThreads=True*.

Parameters

- **filename** (*str*) – File to save the resulting video to, should include the extension.
- **useThreads** (*bool*) – Use threading where possible to speed up the saving process. If *True*, the video will be saved and composited in a separate thread and this function will return quickly. If *False*, the video will be saved and composited in the main thread and this function will block until the video is saved. Default is *True*.
- **mergeAudio** (*bool*) – Merge the audio track from the microphone with the video. If *True*, the audio track will be merged with the video. If *False*, the audio track will be saved to a separate file. Default is *True*.
- **encoderLib** (*str or None*) – Encoder library to use for saving the video. This can be either *'ffpyplayer'* or *'opencv'*. If *None*, the same library that was used to open the camera stream. Default is *None*.
- **encoderOpts** (*dict*) – Options to pass to the encoder. This is a dictionary of options specific to the encoder library being used. See the documentation for *~psychopy.tools.movietools.MovieFileWriter* for more details.

stop()

Stop recording frames and audio (if available).

property streamTime

Current stream time in seconds (*float*). This time increases monotonically from startup.

This is *-1.0* if there is no active stream running or if the backend does not support this feature.

update()

Acquire the newest data from the camera stream. If the *Camera* object is not being monitored by a *ImageStim*, this must be explicitly called.

property win

Window which frames are being presented (*psychopy.visual.Window* or *None*).

class psychopy.hardware.camera.**CameraInfo**(*index=-1*, *name='Null'*, *frameSize=(-1, -1)*, *frameRate=-1.0*, *pixelFormat='Unknown'*, *codecFormat='Unknown'*, *cameraLib='Null'*, *cameraAPI='Null'*)

Information about a specific operating mode for a camera attached to the system.

Parameters

- **index** (*int*) – Index of the camera. This is the enumeration for the camera which is used to identify and select it by the *cameraLib*. This value may differ between operating systems and the *cameraLib* being used.
- **name** (*str*) – Camera name retrieved by the OS. This may be a human-readable name (i.e. DirectShow on Windows), an index on MacOS or a path (e.g., */dev/video0* on Linux). If the *cameraLib* does not support this feature, then this value will be generated.
- **frameSize** (*ArrayLike*) – Resolution of the frame *(w, h)* in pixels.
- **frameRate** (*ArrayLike*) – Allowable framerate for this camera mode.
- **pixelFormat** (*str*) – Pixel format for the stream. If *u'Null'*, then *codecFormat* is being used to configure the camera.
- **codecFormat** (*str*) – Codec format for the stream. If *u'Null'*, then *pixelFormat* is being used to configure the camera. Usually this value is used for high-def stream formats.
- **cameraLib** (*str*) – Library used to access the camera. This can be either, 'ffpyplayer', 'opencv'.
- **cameraAPI** (*str*) – API used to access the camera. This relates to the external interface being used by *cameraLib* to access the camera. This value can be: 'AVFoundation', 'DirectShow' or 'Video4Linux2'.

property **cameraAPI**

Camera API in use to obtain this information (*str*).

property **cameraLib**

Camera library these settings are targeted towards (*str*).

property **codecFormat**

Codec format, may be used instead of *pixelFormat* for some configurations. Default is *''*.

description()

Get a description as a string.

For all backends, this value is guaranteed to be valid after the camera has been opened. Some backends may be able to provide this information before the camera is opened.

Returns
Description of the camera format as a human readable string.

Return type
str

property **frameRate**

Frame rate (*float*) or range (*ArrayLike*).

Depends on the backend being used. If a range is provided, then the first value is the maximum and the second value is the minimum frame rate.

property **frameSize**

Resolution (w, h) in pixels (*ArrayLike* or *None*).

frameSizeAsFormattedString()

Get image size as as formatted string.

Returns

Size formatted as *'WxH'* (e.g. *'480x320'*).

Return type

str

property index

Camera index (*int*). This is the enumerated index of this camera.

property name

Camera name (*str*). This is the camera name retrieved by the OS.

property pixelFormat

Video pixel format (*str*). An empty string indicates this field is not initialized.

11.6.6 Cedrus (response boxes)

Interfaces for Cedrus Corporation devices such as button boxes.

These are optional components that can be obtained by installing the *psychopy-cedrus* extension into the current environment.

class psychopy.hardware.cedrus.**RB730**(**args*, ***kwargs*)

psychopy.hardware.cedrus is now located within the psychopy-cedrus plugin.

When initialised, rather than creating an object, will log an error.

11.6.7 egi (pynetstation)

Support for egi is now provided via the egi-pynetstation third-party library. This is included within Standalone PsychoPy from 2022.2.0

See the **`egi-pynetstation documentation<https://egi-pynetstation.readthedocs.io/en/latest/>`_** for further details.

11.6.8 Launch an fMRI experiment: Test or Scan

Software fMRI machine emulator.

Idea: Run or debug an experiment script using exactly the same code, i.e., for both testing and online data acquisition. To debug timing, you can emulate sync pulses and user responses.

Limitations: pyglet only; keyboard events only.

These are optional components that can be obtained by installing the *psychopy-mri-emulator* extension into the current environment.

class psychopy.hardware.emulator.**ResponseEmulator**(**args*, ***kwargs*)

psychopy.hardware.emulator is now located within the psychopy-mri-emulator plugin.

When initialised, rather than creating an object, will log an error.

class psychopy.hardware.emulator.**SyncGenerator**(**args*, ***kwargs*)

psychopy.hardware.emulator is now located within the psychopy-mri-emulator plugin.

When initialised, rather than creating an object, will log an error.

`class psychopy.hardware.emulator.launchScan(*args, **kwargs)`

psychopy.hardware.emulator is now located within the psychopy-mri-emulator plugin.

When initialised, rather than creating an object, will log an error.

`class psychopy.hardware.emulator.ResponseEmulator(*args, **kwargs)`

psychopy.hardware.emulator is now located within the psychopy-mri-emulator plugin.

When initialised, rather than creating an object, will log an error.

`class psychopy.hardware.emulator.SyncGenerator(*args, **kwargs)`

psychopy.hardware.emulator is now located within the psychopy-mri-emulator plugin.

When initialised, rather than creating an object, will log an error.

11.6.9 fORP response box

Interfaces for Current Designs Inc. devices such as button boxes.

This class is only useful when the fORP is connected via the serial port. If you're connecting via USB, just treat it like a standard keyboard. E.g., use a Keyboard component, and typically listen for Allowed keys `'1', '2', '3', '4', '5'`. Or use `event.getKeys()`.

These are optional components that can be obtained by installing the *psychopy-curdes* extension into the current environment.

`class psychopy.hardware.forp.ButtonBox(*args, **kwargs)`

psychopy.hardware.forp is now located within the psychopy-curdes plugin.

When initialised, rather than creating an object, will log an error.

11.6.10 iolab

The ioLab button box has not been made for many years now (and the company no longer exists) so we have withdrawn support for this device in . We recommend you get a more modern device, such as the LabHackers Millikey

11.6.11 joystick (pyglet and pygame)

AT THE MOMENT JOYSTICK DOES NOT APPEAR TO WORK UNDER PYGLET. We need someone motivated and capable to go and get this right (problem is with event polling under pyglet) Control joysticks and gamepads from within PsychoPy.

For most backends, you do need a window using the same backend (and you need to be flipping it) for the joystick to be updated.

`exception psychopy.hardware.joystick.JoysticButtonNotAvailableError`

Exception raised when a button is not available on the joystick.

`class psychopy.hardware.joystick.Joystick(device=0, **kwargs)`

Class for interfacing with a multi-axis joystick or gamepad.

Upon creating a *Joystick* object, the joystick device is opened and the states of the device's axes and buttons can be read.

Values for the axes are returned as floating point numbers, typically between -1.0 and +1.0 unless scaling is applied. The values for the buttons are returned as booleans, where True indicates the button is pressed down at the time the device was last polled.

Scaling factors can be set for each axis to adjust the range of the axis values. The scaling factor is a floating point value that is multiplied by the axis value. If the scaling factor is negative, the axis value is inverted. Deadzones can also be applied for each axis to prevent small fluctuations in the joystick's resting position from being interpreted as valid input. The deadzone is a floating point value between 0.0 and 1.0. If the absolute value of the axis value is less than the deadzone, the axis value is set to zero.

Device inputs can be named to provide a more human-readable interface. The names can be set for axes, buttons, and hats where they can be used to get the input values instead of using the integer indices. Furthermore, like inputs can be grouped together under a single name. For example, both X and Y of a thumbstick can be grouped together under the name 'thumbstick'. When getting the value of the thumbstick, a tuple of the X and Y values is returned instead of having to get each axis individually.

Parameters

device (*int or str*) – The index or name of the joystick to control.

Examples

Typical usage:

```
from psychopy.hardware import joystick
from psychopy import visual

joystick.backend='pyglet'  # must match the Window
win = visual.Window([400,400], winType='pyglet')

nJoys = joystick.getNumJoysticks()  # to check if we have any
id = 0
joy = joystick.Joystick(id)  # id must be <= nJoys - 1

nAxes = joy.getNumAxes()  # for interest
while True:  # while presenting stimuli
    joyX = joy.getX()
    # ...
    win.flip()  # flipping implicitly updates the joystick info
```

Set the deadzone for axis 0 to 0.1:

```
joy.setAxisDeadzone(0, 0.1)
```

Set the scaling factor for 1 axis to 2.0:

```
joy.setAxisScale(1, 2.0)
```

Setting the names of the inputs can be useful for debugging and for providing a more human-readable interface:

```
joy.setInputName('axis', 0, 'x')
joy.setInputName('axis', 1, 'y')
```

You can get the imput value by name by passing it to the get method for the input type:

```
joy.getAxis('axis', 'x')  # instead of joy.getAxis(0)
```

Automatically set the input names to the default Xbox controller mapping scheme:

```
joy.setInputScheme('xbox')
# ...
xVal, yVal = joy.getAxis('left_thumbstick')
leftTrigger, rightTrigger = joy.getAxis('triggers')
```

Notes

- You do need to be flipping frames (or dispatching events manually) in order for the values of the joystick to be updated.
- Currently under pyglet backends the axis values initialise to zero rather than reading the current true value. This gets fixed on the first change to each axis.
- Currently pygame (1.9.1) spits out lots of debug messages about the joystick and these can't be turned off :-/
- The GLFW backend can be used without first opening a window and can be used with other window backends.

`_getIndexFromName`(*inputType*, *name*)

Get the index of an input from its name.

Parameters

- **inputType** (*str*) – The type of input to get the index for. Must be one of 'axis', 'button', or 'hat'.
- **name** (*str*) – The name of the input to get the index for.

Returns

The index of the input. If the input name is not found, *None* is returned.

Return type

int or None

Raises

`InvalidInputNameError` – If the input name is not valid or has not been set.

`close()`

Close the joystick device.

`property deviceIndex`

The index of the joystick (*int*).

`getAllAxes()`

Get a list of all current axis values (*int*).

`getAllButtons()`

Get the state of all buttons on the devics.

Returns

A list of button states. Each state is a boolean.

Return type

list

`getAllHats()`

Get the current values of all available hats.

Returns

Each value is a tuple (x, y) where x and y axis states are trinary (-1, 0, +1)

Return type

list

`static getAvailableDevices()`

Return a list of available joystick devices.

This method is used by *DeviceManager* to get a list of available devices.

Returns
A list of available joystick devices.

Return type
list

`getAxis(axisId)`

Get the value of an axis by an integer id.

Parameters
`axisId` (*int, str or list*) – The axis ID to get the value for. If a string is supplied, the name of the axis is used to get the value. If a list of axes indices or names is supplied, a list of values is returned.

Returns
The value of the axis. If a list of axes is supplied, a list of values is returned.

Return type
float or list

`getAxisDeadzone(axisId)`

Get the deadzone for a given axis.

Parameters
`axisId` (*int*) – The axis ID to get the deadzone for.

Returns
The deadzone for the given axis.

Return type
float

`getAxisScale(axisId)`

Get the scale factor for a given axis.

Parameters
`axisId` (*int*) – The axis ID to get the scale factor for.

Returns
The scale factor for the given axis.

Return type
float

`getButton(buttonId)`

Get the state of a given button on the device (*bool*).

Parameters
`buttonId` (*int, str or list*) – The button ID to get the state for. If a string is supplied, the name of the button is used to get the state. If a list of button indices or names is supplied, a list of states is returned.

Returns
The state of the button. If a list of buttons was passed as *buttonId*, a list of states is returned where each state is a boolean.

Return type
bool or list

`getHat`(*hatId=0*)

Get the position of a particular hat.

Parameters
hatId (*int or str*) – The hat ID to get the position for. If a string is supplied, the name of the hat is used to get the position.

Returns
The position returned is an (x, y) tuple where x and y can be -1, 0 or +1.

Return type
tuple

`getName`()

Return the manufacturer-defined name describing the device (*str*).

`getNumAxes`()

Get the number of available joystick axes.

The first axis usually corresponds to the X axis, the second to the Y axis for most joysticks. Additional axes may be present for other controls such as addtional thumbsticks or throttle lever.

Returns
The number of axes found on the joystick.

Return type
int

`getNumButtons`()

Get the number of buttons on the device (*int*).

Returns
The number of buttons on the joystick.

Return type
int

`getNumHats`()

Get the number of hats on this joystick.

The GLFW backend makes no distinction between hats and buttons. Calling ‘getNumHats()’ will return 0.

`getRX`()

Return the RX axis value (equivalent to joystick.getAxis(3)).

`getRY`()

Return the RY axis value (equivalent to joystick.getAxis(4)).

`getRZ`()

Return the RZ axis value (equivalent to joystick.getAxis(5)).

`getX`()

Return the X axis value (equivalent to joystick.getAxis(0)).

`getXY`()

Return the X and Y axis values as a tuple.

Returns
The X and Y axis values as a tuple.

Return type
tuple

`getY()`

Return the Y axis value (equivalent to joystick.getAxis(1)).

`getZ()`

Return the Z axis value (equivalent to joystick.getAxis(2)).

`property hasTracking`

Check if the joystick has tracking capabilities.

Returns
True if the joystick has tracking capabilities, False otherwise.

Return type
bool

`property inputLib`

Input interface library used (*str*).

`property isOpen`

Check if the joystick device is open.

Returns
True if the joystick device is open, False otherwise.

Return type
bool

`isSameDevice(`*otherDevice*`)`

Check if the device is the same as another device.

Parameters
otherDevice (`Joystick`) – The other device to compare against.

Returns
True if the devices are the same, False otherwise.

Return type
bool

`lastUpdateTime()`

Return the time of the last update to the joystick state.

Returns
The time of the last update to the joystick state.

Return type
float

`property name`

Name of the joystick reported by the system (*str*).

`open()`

Open the joystick device.

`poll()`

Poll the joystick device for the current state.

This method should be called at the beginning of each frame to update the state of the joystick device. The time of the last update is stored and can be accessed using the *lastUpdateTime* property.

property rx

The RX axis value (*float*).

property ry

The RY axis value (*float*).

property rz

The RZ axis value (*float*).

setAxisDeadzone(*axisId=None*, *deadzone=0.1*)

Set the deadzone for a given axis.

Parameters

- **axisId** (*int, str, list or None*) – The axis ID to set the deadzone for. If None, set the deadzone for all axes to the given value. A string can be supplied to set the deadzone for an axis by name. A list of axes can also be supplied to set the deadzone for multiple axes at once.
- **deadzone** (*float*) – The deadzone to set, must be between 0.0 and 1.0.

setAxisScale(*axisId*, *scale*)

Set the scale factor for a given axis.

Parameters

- **axisId** (*int or None*) – The axis ID to set the scale factor for. If None, set the scale factor for all axes to the given value.
- **scale** (*float*) – The scale factor to set. This factor will be multiplied by the axis value. If negative, the axis value will be inverted.

setInputName(*inputType*, *inputIndex*, *name*)

Set the name of an input.

Parameters

- **inputType** (*str*) – The type of input to set the name for. Must be one of 'axis', 'button', or 'hat'.
- **inputIndex** (*int or list of int*) – The index of the input to set the name for. If a list of indices is supplied, multiple axes will be grouped together.
- **name** (*str or None*) – The name to set for the axis. If None, the name for the axis is removed.

Raises

ValueError – If the inputType is not 'axis', 'button', or 'hat'.

Examples

Set the name of axis *0* to 'x' and get its value by name:

```
joy.setInputName('axis', 0, 'x')
xVal = joy.getAxis('x')  # instead of joy.getAxis(0)
```

Joystick inputs often have multiple axes ganged together on a single control, such as a thumbstick. You can group axes together by passing a list of indices:

```
joy.setInputName('axis', [0, 1], 'left_thumbstick')
xVal, yVal = joy.getAxis('left_thumbstick')  # returns 2 values
```

`setInputScheme`(*mapping*)

Set the input mapping scheme for the joystick.

The input mapping scheme determines the names of the inputs for the joystick. The mapping scheme can be set to 'default', 'xbox', or 'custom'. The default mapping scheme provides names for the axes and buttons that are common to most joysticks.

Note that setting the mapping scheme will overwrite any custom input names that have been set prior to calling this method.

Parameters
`mapping` (*str*) – The mapping scheme to set. Must be one of 'default', 'xbox', or 'custom'.

`property trackerData`

Tracker data for the controller.

Returns
The tracker data.

Return type
TrackerData or *None*

`property x`

The X axis value (*float*).

`property y`

The Y axis value (*float*).

`property z`

The Z axis value (*float*).

`exception psychopy.hardware.joystick.JoystickAxisNotAvailableError`

Exception raised when an axis is not available on the joystick.

`exception psychopy.hardware.joystick.JoystickError`

Exception raised for errors in the joystick module.

`psychopy.hardware.joystick.getJoystickInterfaces()`

Get available joystick input interfaces.

Returns
A mapping of joystick interfaces available where the key is the input library identifier and the value is the joystick interface class. Setting the backend to one of these keys will use the corresponding joystick interface.

Return type
dict

`class psychopy.hardware.joystick.Joystick`(*device=0*, ***kwargs*)

Class for interfacing with a multi-axis joystick or gamepad.

Upon creating a *Joystick* object, the joystick device is opened and the states of the device's axes and buttons can be read.

Values for the axes are returned as floating point numbers, typically between -1.0 and +1.0 unless scaling is applied. The values for the buttons are returned as booleans, where True indicates the button is pressed down at the time the device was last polled.

Scaling factors can be set for each axis to adjust the range of the axis values. The scaling factor is a floating point value that is multiplied by the axis value. If the scaling factor is negative, the axis value is inverted. Deadzones can also be applied for each axis to prevent small fluctuations in the joystick's resting position from being interpreted as valid input. The deadzone is a floating point value between 0.0 and 1.0. If the absolute value of the axis value is less than the deadzone, the axis value is set to zero.

Device inputs can be named to provide a more human-readable interface. The names can be set for axes, buttons, and hats where they can be used to get the input values instead of using the integer indices. Furthermore, like inputs can be grouped together under a single name. For example, both X and Y of a thumbstick can be grouped together under the name 'thumbstick'. When getting the value of the thumbstick, a tuple of the X and Y values is returned instead of having to get each axis individually.

Parameters

device (*int or str*) – The index or name of the joystick to control.

Examples

Typical usage:

```
from psychopy.hardware import joystick
from psychopy import visual

joystick.backend='pyglet'  # must match the Window
win = visual.Window([400,400], winType='pyglet')

nJoys = joystick.getNumJoysticks()  # to check if we have any
id = 0
joy = joystick.Joystick(id)  # id must be <= nJoys - 1

nAxes = joy.getNumAxes()  # for interest
while True:  # while presenting stimuli
    joyX = joy.getX()
    # ...
    win.flip()  # flipping implicitly updates the joystick info
```

Set the deadzone for axis 0 to 0.1:

```
joy.setAxisDeadzone(0, 0.1)
```

Set the scaling factor for 1 axis to 2.0:

```
joy.setAxisScale(1, 2.0)
```

Setting the names of the inputs can be useful for debugging and for providing a more human-readable interface:

```
joy.setInputName('axis', 0, 'x')
joy.setInputName('axis', 1, 'y')
```

You can get the imput value by name by passing it to the get method for the input type:

```
joy.getAxis('axis', 'x')  # instead of joy.getAxis(0)
```

Automatically set the input names to the default Xbox controller mapping scheme:

```
joy.setInputScheme('xbox')
# ...
xVal, yVal = joy.getAxis('left_thumbstick')
leftTrigger, rightTrigger = joy.getAxis('triggers')
```

Notes

- You do need to be flipping frames (or dispatching events manually) in order for the values of the joystick to be updated.
- Currently under pyglet backends the axis values initialise to zero rather than reading the current true value. This gets fixed on the first change to each axis.
- Currently pygame (1.9.1) spits out lots of debug messages about the joystick and these can't be turned off :-/
- The GLFW backend can be used without first opening a window and can be used with other window backends.

`_getIndexFromName`(*inputType*, *name*)

Get the index of an input from its name.

Parameters

- **inputType** (*str*) – The type of input to get the index for. Must be one of 'axis', 'button', or 'hat'.
- **name** (*str*) – The name of the input to get the index for.

Returns

The index of the input. If the input name is not found, *None* is returned.

Return type

int or None

Raises

`InvalidInputNameError` – If the input name is not valid or has not been set.

`close()`

Close the joystick device.

`property deviceIndex`

The index of the joystick (*int*).

`getAllAxes()`

Get a list of all current axis values (*int*).

`getAllButtons()`

Get the state of all buttons on the devics.

Returns

A list of button states. Each state is a boolean.

Return type

list

`getAllHats()`

Get the current values of all available hats.

Returns

Each value is a tuple (x, y) where x and y axis states are trinary (-1, 0, +1)

Return type

list

`static getAvailableDevices()`

Return a list of available joystick devices.

This method is used by *DeviceManager* to get a list of available devices.

Returns
A list of available joystick devices.

Return type
list

`getAxis(axisId)`

Get the value of an axis by an integer id.

Parameters
axisId (*int, str or list*) – The axis ID to get the value for. If a string is supplied, the name of the axis is used to get the value. If a list of axes indices or names is supplied, a list of values is returned.

Returns
The value of the axis. If a list of axes is supplied, a list of values is returned.

Return type
float or list

`getAxisDeadzone(axisId)`

Get the deadzone for a given axis.

Parameters
axisId (*int*) – The axis ID to get the deadzone for.

Returns
The deadzone for the given axis.

Return type
float

`getAxisScale(axisId)`

Get the scale factor for a given axis.

Parameters
axisId (*int*) – The axis ID to get the scale factor for.

Returns
The scale factor for the given axis.

Return type
float

`getButton(buttonId)`

Get the state of a given button on the device (*bool*).

Parameters
buttonId (*int, str or list*) – The button ID to get the state for. If a string is supplied, the name of the button is used to get the state. If a list of button indices or names is supplied, a list of states is returned.

Returns
The state of the button. If a list of buttons was passed as *buttonId*, a list of states is returned where each state is a boolean.

Return type
bool or list

`getHat`(*hatId=0*)

Get the position of a particular hat.

Parameters
hatId (*int or str*) – The hat ID to get the position for. If a string is supplied, the name of the hat is used to get the position.

Returns
The position returned is an (x, y) tuple where x and y can be -1, 0 or +1.

Return type
tuple

`getName`()

Return the manufacturer-defined name describing the device (*str*).

`getNumAxes`()

Get the number of available joystick axes.

The first axis usually corresponds to the X axis, the second to the Y axis for most joysticks. Additional axes may be present for other controls such as addtional thumbsticks or throttle lever.

Returns
The number of axes found on the joystick.

Return type
int

`getNumButtons`()

Get the number of buttons on the device (*int*).

Returns
The number of buttons on the joystick.

Return type
int

`getNumHats`()

Get the number of hats on this joystick.

The GLFW backend makes no distinction between hats and buttons. Calling 'getNumHats()' will return 0.

`getRX`()

Return the RX axis value (equivalent to joystick.getAxis(3)).

`getRY`()

Return the RY axis value (equivalent to joystick.getAxis(4)).

`getRZ`()

Return the RZ axis value (equivalent to joystick.getAxis(5)).

`getX`()

Return the X axis value (equivalent to joystick.getAxis(0)).

`getXY`()

Return the X and Y axis values as a tuple.

Returns
The X and Y axis values as a tuple.

Return type
tuple

`getY()`

Return the Y axis value (equivalent to joystick.getAxis(1)).

`getZ()`

Return the Z axis value (equivalent to joystick.getAxis(2)).

`property hasTracking`

Check if the joystick has tracking capabilities.

Returns
True if the joystick has tracking capabilities, False otherwise.

Return type
bool

`property inputLib`

Input interface library used (*str*).

`property isOpen`

Check if the joystick device is open.

Returns
True if the joystick device is open, False otherwise.

Return type
bool

`isSameDevice(`*otherDevice*`)`

Check if the device is the same as another device.

Parameters
otherDevice (`Joystick`) – The other device to compare against.

Returns
True if the devices are the same, False otherwise.

Return type
bool

`lastUpdateTime()`

Return the time of the last update to the joystick state.

Returns
The time of the last update to the joystick state.

Return type
float

`property name`

Name of the joystick reported by the system (*str*).

`open()`

Open the joystick device.

`poll()`

Poll the joystick device for the current state.

This method should be called at the beginning of each frame to update the state of the joystick device. The time of the last update is stored and can be accessed using the *lastUpdateTime* property.

property rx

The RX axis value (*float*).

property ry

The RY axis value (*float*).

property rz

The RZ axis value (*float*).

setAxisDeadzone(*axisId=None*, *deadzone=0.1*)

Set the deadzone for a given axis.

Parameters

- **axisId** (*int, str, list or None*) – The axis ID to set the deadzone for. If None, set the deadzone for all axes to the given value. A string can be supplied to set the deadzone for an axis by name. A list of axes can also be supplied to set the deadzone for multiple axes at once.
- **deadzone** (*float*) – The deadzone to set, must be between 0.0 and 1.0.

setAxisScale(*axisId*, *scale*)

Set the scale factor for a given axis.

Parameters

- **axisId** (*int or None*) – The axis ID to set the scale factor for. If None, set the scale factor for all axes to the given value.
- **scale** (*float*) – The scale factor to set. This factor will be multiplied by the axis value. If negative, the axis value will be inverted.

setInputName(*inputType*, *inputIndex*, *name*)

Set the name of an input.

Parameters

- **inputType** (*str*) – The type of input to set the name for. Must be one of ‘axis’, ‘button’, or ‘hat’.
- **inputIndex** (*int or list of int*) – The index of the input to set the name for. If a list of indices is supplied, multiple axes will be grouped together.
- **name** (*str or None*) – The name to set for the axis. If None, the name for the axis is removed.

Raises

ValueError – If the inputType is not ‘axis’, ‘button’, or ‘hat’.

Examples

Set the name of axis *0* to ‘x’ and get its value by name:

```
joy.setInputName('axis', 0, 'x')
xVal = joy.getAxis('x')  # instead of joy.getAxis(0)
```

Joystick inputs often have multiple axes ganged together on a single control, such as a thumbstick. You can group axes together by passing a list of indices:

```
joy.setInputName('axis', [0, 1], 'left_thumbstick')
xVal, yVal = joy.getAxis('left_thumbstick')  # returns 2 values
```

setInputScheme(*mapping*)

Set the input mapping scheme for the joystick.

The input mapping scheme determines the names of the inputs for the joystick. The mapping scheme can be set to 'default', 'xbox', or 'custom'. The default mapping scheme provides names for the axes and buttons that are common to most joysticks.

Note that setting the mapping scheme will overwrite any custom input names that have been set prior to calling this method.

Parameters

mapping (*str*) – The mapping scheme to set. Must be one of 'default', 'xbox', or 'custom'.

property trackerData

Tracker data for the controller.

Returns

The tracker data.

Return type

TrackerData or *None*

property x

The X axis value (*float*).

property y

The Y axis value (*float*).

property z

The Z axis value (*float*).

11.6.12 labjacks (USB I/O devices)

provides an interface to the labjack U3 class with a couple of minor additions.

This is accessible by:

```
from psychopy.hardware.labjacks import U3
```

Except for the additional *setdata* function the U3 class operates exactly as that in the U3 library that labjack provides, documented here:

http://labjack.com/support/labjackpython

Note

To use labjack devices you do need also to install the driver software described on the page above

class psychopy.hardware.labjacks.U3(**args*, ***kwargs*)

psychopy.hardware.labjacks is now located within the psychopy-labjack plugin.

When initialised, rather than creating an object, will log an error.

11.6.13 Minolta

Interfaces for Minolta light-measuring devices.

These are optional components that can be obtained by installing the *psychopy-minolta* extension into the current environment.

class psychopy.hardware.minolta.**CS100A**(**args*, ***kwargs*)

psychopy.hardware.minolta is now located within the psychopy-minolta plugin.

When initialised, rather than creating an object, will log an error.

class psychopy.hardware.minolta.**LS100**(**args*, ***kwargs*)

psychopy.hardware.minolta is now located within the psychopy-minolta plugin.

When initialised, rather than creating an object, will log an error.

11.6.14 PhotoResearch

Supported devices:

- *PR650*
- *PR655/PR670*

Interfaces for Photo Research Inc. spectroradiometers.

These are optional components that can be obtained by installing the *psychopy-photoresearch* extension into the current environment.

class psychopy.hardware.pr.**PR650**(**args*, ***kwargs*)

psychopy.hardware.pr is now located within the psychopy-photoresearch plugin.

When initialised, rather than creating an object, will log an error.

class psychopy.hardware.pr.**PR655**(**args*, ***kwargs*)

psychopy.hardware.pr is now located within the psychopy-photoresearch plugin.

When initialised, rather than creating an object, will log an error.

11.6.15 pylink (SR Research)

For now the SR Research `pylink` module is packaged with the Standalone flavours of and can be imported with:

```
import pylink
```

You do need to install the Display Software (which they also call Eyelink Developers Kit) for your particular platform. This can be found by following the threads from the SR Research support forum (creating an account is required):

https://www.sr-support.com/thread-13.html

For documentation of `pylink`, see:

https://www.sr-support.com/thread-48.html

11.6.16 `pump` - A simple interface to the Cetoni neMESYS syringe pump system

Please specify the name of the pump configuration to use in the preferences under `Hardware / Qmix pump configuration`. See the readme file of the `pyqmix` project for details on how to set up your computer and create the configuration file. Interfaces for Cetoni neMESYS syringe pump systems.

These are optional components that can be obtained by installing the *psychopy-qmix* extension into the current environment.

class psychopy.hardware.qmix.**Pump**(**args*, ***kwargs*)

psychopy.hardware.qmix is now located within the psychopy-labjack plugin.

When initialised, rather than creating an object, will log an error.

11.6.17 Speaker

class psychopy.hardware.speaker.**SpeakerDevice**(*index=None*, *name=None*, *latencyClass=1*, *resample=True*)

Class for managing a physical speaker device for audio playback.

Parameters

- **index** (*int, optional*) – Numeric index for the physical speaker device, according to psychtoolbox. Leave as None to find the speaker by name.
- **name** (*str, optional*) – String name for the physical speaker device, according to your operating system. Leave as None to find the speaker by numeric index.
- **latencyClass** (*int*) – One of:
  - 0: Don't take exclusive control over the speaker, so other apps can still use it. Send

  sounds via the system mixer so that sample rates are all handled, even though this introduces latency.

  - 1: Don't take exclusive control over the speaker, so other apps can still use it. Send

  sounds directly to reduce latency, so sounds will need to match the sample rate of the speaker. **Recommended in most cases; if `resample` is True then sample rates are already handled on load!**

  - 2: Take exclusive control over the speaker, so other apps can't use it. Send sounds

  directly to reduce latency, so sounds will need to be the same sample rate as one another, but this can be any sample rate supported by the speaker.

  - 3: Take exclusive control over the speaker, so other apps can't use it. Send sounds

  directly to reduce latency, so sounds will need to be the same sample rate as one another, but this can be any sample rate supported by the speaker. Force the system to prioritise resources towards playing sounds on this speaker for absolute minimum latency, but fallback to mode 2 if the system rejects this.

  - 4: Take exclusive control over the speaker, so other apps can't use it. Send sounds

  directly to reduce latency, so sounds will need to be the same sample rate as one another, but this can be any sample rate supported by the speaker. Force the system to prioritise resources towards playing sounds on this speaker for absolute minimum latency, and raise an error if the system rejects this.

- **resample** (*bool, optional*) – If the sample rate of an audio clip doesn't match the sample rate of the speaker, should PsychoPy resample the sound on load?

close()

Close the audio stream for this speaker.

createStream()

Create the psychtoolbox audio stream

Calling this method will set the following attributes

profile

The profile from psychtoolbox, a dict with the following keys: Active, State, RequestedStartTime, StartTime, CaptureStartTime, RequestedStopTime, EstimatedStopTime, CurrentStreamTime, ElapsedOutSamples, PositionSecs, RecordedSecs, ReadSecs, SchedulePosition, XRuns, TotalCalls, TimeFailed, BufferSize, CPULoad, PredictedLatency, LatencyBias, SampleRate, OutDeviceIndex, InDeviceIndex

Type
dict

`index`

A numeric index referring to the device. This may differ from the value of *index* this object was initialised with, as this will be the numeric index of the actual physical speaker best matching what was requested.

Type
int

`name`

A string name referring to the device. This may differ from the value of *name* this object was initialised with, as this will be the system-reported name of the actual physical speaker best matching what was requested.

Type
str

`property exclusive`

returns: Does PsychoPy have exclusive control of this speaker? If True then other apps will not be
able to play sounds on the same speaker.

Return type
bool

`static getAvailableDevices()`

Get all available devices of this type.

Returns
List of dictionaries containing the parameters needed to initialise each device.

Return type
list[dict]

`getDeviceProfile()`

Generate a dictionary describing this device by finding the profile from getAvailableDevices which represents the same physical device as this object.

Returns
Dictionary representing this device

Return type
dict

`getJSON`(*asString=True*)

Convert the output of getDeviceProfile to a JSON string.

Parameters
`asString` (*bool*) – If True, then the output will be converted to a string, otherwise will simply be a JSON-friendly dict.

Returns
JSON string (or JSON friendly dict) of getDeviceProfile.

Return type
str or dict

`property isOpen`

Is this speaker "open", i.e. is it active and ready for a Sound to play tracks on it

isSameDevice(*other*)

Determine whether this object represents the same physical speaker as a given other object.

Parameters

other (SpeakerDevice, *dict*) – Other SpeakerDevice to compare against, or a dict of params (which must include *index* as a key)

Returns

True if the two objects represent the same physical device

Return type

bool

open()

Open the audio stream for this speaker so that sound can be played to it.

testDevice()

Play a simple sound to check whether this device is working.

Classes and functions for using trigger boxes.

Trigger boxes are used to send electrical signals to external devices. They are typically used to synchronize the presentation of stimuli with the recording of physiological data. This module provides a common interface for accessing trigger boxes from within PsychoPy.

This module serves as the entry point for plugin classes implementing third-party trigger box interfaces. All installed interfaces are discoverable by calling the *getAllTriggerBoxes()* function. To have your trigger box interface discovered by PsychoPy, you need to create a Python module that defines a class inheriting from *BaseTriggerBox* and set its entry point to `psychopy.hardware.triggerbox` in your plugin setup script or configuration file.

11.6.18 Overview

BaseTriggerBox(*args, **kwargs)	Base class for trigger box interfaces.
BaseTriggerBox.deviceName	Get the name of the trigger box.
BaseTriggerBox.deviceVendor	Get the name of the manufacturer.
BaseTriggerBox.getCapabilities(**kwargs)	Get the capabilities of the trigger box.
BaseTriggerBox.open(**kwargs)	Open a connection to the trigger box.
BaseTriggerBox.close(**kwargs)	Close the trigger box.
BaseTriggerBox.isOpen	Check if the trigger box connection is open.
BaseTriggerBox.setData(data, **kwargs)	Set the data to be sent.
BaseTriggerBox.getData(**kwargs)	Get the data to be sent.
BaseTriggerBox.setPin(pin, value, **kwargs)	Set the value of a pin.
BaseTriggerBox.getPin(pin, **kwargs)	Read the value of a pin.
getAllTriggerBoxes()	Get all trigger box interface classes.

11.6.19 Details

class psychopy.hardware.triggerbox.**BaseTriggerBox**(**args*, ***kwargs*)

Base class for trigger box interfaces.

This class defines the minimal interface for trigger box implementations. All trigger box implementations should inherit from this class and override its methods.

Initialize the trigger box interface.

`close(**kwargs)`

Close the trigger box.

`property deviceName`

Get the name of the trigger box.

Returns
Name of the trigger box.

Return type
str

`property deviceVendor`

Get the name of the manufacturer.

Returns
Name of the manufacturer.

Return type
str

`static getAvailableDevices()`

Get all available devices of this type.

Returns
List of dictionaries containing the parameters needed to initialise each device.

Return type
list[dict]

`getCapabilities(**kwargs)`

Get the capabilities of the trigger box.

The returned dictionary contains information about the capabilities of the trigger box. The strutcture of the dictionary may vary between trigger box implementations, so it is recommended to check if a key exists before accessing it.

Returns
Capabilities of the trigger box. The names of the capabilities are the keys of the dictionary. The values are information realted to the specified capability.

Return type
dict

Examples

Check what the required baudrate of the device is:

useBaudrate = getCapabilities()['baudrate']

`getData(**kwargs)`

Get the data to be sent.

Returns
Data to be sent.

Return type
int

`getDeviceProfile()`

Generate a dictionary describing this device by finding the profile from getAvailableDevices which represents the same physical device as this object.

Returns
Dictionary representing this device

Return type
dict

`getJSON`(*asString=True*)

Convert the output of getDeviceProfile to a JSON string.

Parameters
`asString` (*`bool`*) – If True, then the output will be converted to a string, otherwise will simply be a JSON-friendly dict.

Returns
JSON string (or JSON friendly dict) of getDeviceProfile.

Return type
str or dict

`getPin`(*pin*, ***kwargs*)

Read the value of a pin.

Parameters
`pin` (*`int`*) – Pin number.

Returns
Value of the pin.

Return type
int

`property isOpen`

Check if the trigger box connection is open.

Returns
True if the trigger box is open, False otherwise.

Return type
bool

`isSameDevice`(*other*)

Determine whether this object represents the same physical device as a given other object.

Parameters
`other` (*`BaseDevice, dict`*) – Other device object to compare against, or a dict of params.

Returns
True if the two objects represent the same physical device

Return type
bool

`open`(***kwargs*)

Open a connection to the trigger box.

`setData`(*data*, ***kwargs*)

Set the data to be sent.

Parameters
`data` (*`int`*) – Data to be sent.

setPin(*pin*, *value*, ***kwargs*)

Set the value of a pin.

Parameters

- **pin** (*int*) – Pin number.
- **value** (*int*) – Value to be set.

psychopy.hardware.triggerbox.**getAllTriggerBoxes**()

Get all trigger box interface classes.

Returns

Mapping of trigger box classes.

Return type

dict

psychopy.hardware.**findPhotometer**(*ports=None*, *device=None*)

Try to find a connected photometer/photospectrometer!

PsychoPy will sweep a series of serial ports trying to open them. If a port successfully opens then it will try to issue a command to the device. If it responds with one of the expected values then it is assumed to be the appropriate device.

Parameters

- **ports** (*list*) – A list of ports to search. Each port can be a string (e.g. 'COM1', '/dev/tty.Keyspan1.1') or a number (for win32 comports only). If *None* is provided then PsychoPy will sweep COM0-10 on Win32 and search known likely port names on MacOS and Linux.
- **device** (*str*) – String giving expected device (e.g. 'PR650', 'PR655', 'CS100A', 'LS100', 'LS110', 'S470'). If this is not given then an attempt will be made to find a device of any type, but this often fails.

Returns

An object representing the first photometer found, *None* if the ports didn't yield a valid response. *None* if there were not even any valid ports (suggesting a driver not being installed.)

Return type

object or None

Examples

Sweeps ports 0 to 10 searching for a PR655:

```
photom = findPhotometer(device='PR655')
print(photom.getLum())
if hasattr(photom, 'getSpectrum'):
    # can retrieve spectrum (e.g. a PR650)
    print(photom.getSpectrum())
```

11.7 psychopy.iohub - ioHub event monitoring framework

ioHub monitors for device events in parallel with the experiment execution by running in a separate process than the main script. This means, for instance, that keyboard and mouse event timing is not quantized by the rate at which the window.flip() method is called.

ioHub reports device events to the experiment runtime as they occur. Optionally, events can be saved to a HDF5 file.

All iohub events are timestamped using the global time base (psychopy.core.getTime()). Events can be accessed as a device independent event stream, or from a specific device of interest.

A comprehensive set of examples that each use at least one of the iohub devices is available in the psychopy/demos/coder/iohub folder.

11.7.1 Starting the psychopy.iohub Process

To use ioHub within your Coder experiment script, ioHub needs to be started at the beginning of the experiment script.

The easiest way to do this is by calling the launchHubServer function.

launchHubServer Function

`psychopy.iohub.client.`**`launchHubServer`**`(`***kwargs*`)`

Starts the ioHub Server subprocess, and return a *`psychopy.iohub.client.ioHubConnection`* object that is used to access enabled iohub device's events, get events, and control the ioHub process during the experiment.

By default (no kwargs specified), the ioHub server does not create an ioHub HDF5 file, events are available to the experiment program at runtime. The following Devices are enabled by default:

- Keyboard: named 'keyboard', with runtime event reporting enabled.
- Mouse: named 'mouse', with runtime event reporting enabled.
- Monitor: named 'monitor'.
- Experiment: named 'experiment'.

To customize how the ioHub Server is initialized when started, use one or more of the following keyword arguments when calling the function:

Parameters

- **experiment_code** (*str, <= 256 char*) – If experiment_code is provided, an ioHub HDF5 file will be created for the session.
- **session_code** (*str, <= 256 char*) – When specified, used as the name of the ioHub HDF5 file created for the session.
- **experiment_info** (*dict*) – Can be used to save the following experiment metadata fields: code (<=256 chars), title (<=256 chars), description (<=4096 chars), version (<=32 chars)
- **session_info** (*dict*) – Can be used to save the following session metadata fields: code (<=256 chars), name (<=256 chars), comments (<=4096 chars), user_variables (dict)
- **datastore_name** (*str*) – Used to provide an ioHub HDF5 file name different than the session_code.
- **window** (*psychopy.visual.Window*) – The psychoPy experiment window being used. Information like display size, viewing distance, coord / color type is used to update the ioHub Display device.
- **iohub_config_name** (*str*) – Specifies the name of the iohub_config.yaml file that contains the ioHub Device list to be used by the ioHub Server. i.e. the 'device_list' section of the yaml file.
- **iohub.device.path** (*str*) – Add an ioHub Device by using the device class path as the key, and the device's configuration in a dict value.
- **psychopy_monitor** (*(deprecated)*) – The path to a Monitor Center config file
- **Examples** –

A. Wait for the 'q' key to be pressed:

```
from psychopy.iohub.client import launchHubServer

# Start the ioHub process. 'io' can now be used during the
# experiment to access iohub devices and read iohub device events.
io=launchHubServer()

print("Press any Key to Exit Example.....")

# Wait until a keyboard event occurs
keys = io.devices.keyboard.waitForKeys(keys=['q',])

print("Key press detected: {}".format(keys))
print("Exiting experiment....")

# Stop the ioHub Server
io.quit()
```

- **examples** (*Please see the psychopy/demos/coder/iohub/launchHub.py demo for*)
- **function.** (*of different ways to use the launchHubServer*)

ioHubConnection Class

The psychopy.iohub.ioHubConnection object returned from the launchHubServer function provides methods for controlling the iohub process and accessing iohub devices and events.

class psychopy.iohub.client.**ioHubConnection**(*ioHubConfig=None*, *ioHubConfigAbsPath=None*)

ioHubConnection is responsible for creating, sending requests to, and reading replies from the ioHub Process. This class is also used to shut down and disconnect the ioHub Server process.

The ioHubConnection class is also used as the interface to any ioHub Device instances that have been created so that events from the device can be monitored. These device objects can be accessed via the ioHubConnection .devices attribute, providing 'dot name' access to enabled devices. Alternatively, the .getDevice(name) method can be used and will return None if the device name specified does not exist.

Using the .devices attribute is handy if you know the name of the device to be accessed and you are sure it is actually enabled on the ioHub Process.

An example of accessing a device using the .devices attribute:

```
# get the Mouse device, named mouse
mouse=hub.devices.mouse
mouse_position = mouse.getPosition()

print('mouse position: ', mouse_position)

# Returns something like:
# >> mouse position:  [-211.0, 371.0]
```

getDevice(*deviceName*)

Returns the ioHubDeviceView that has a matching name (based on the device : name property specified in the ioHub_config.yaml for the experiment). If no device with the given name is found, None is returned. Example, accessing a Keyboard device that was named 'kb'

```
keyboard = self.getDevice('kb')
kb_events= keyboard.getEvent()
```

This is the same as using the ‘natural naming’ approach supported by the .devices attribute, i.e:

```
keyboard = self.devices.kb
kb_events= keyboard.getEvent()
```

However the advantage of using getDevice(device_name) is that an exception is not created if you provide an invalid device name, or if the device is not enabled on the ioHub server; None is returned instead.

Parameters

deviceName (*str*) – Name given to the ioHub Device to be returned

Returns

The ioHubDeviceView instance for deviceName.

getEvents(*device_label=None*, *as_type='namedtuple'*)

Retrieve any events that have been collected by the ioHub Process from monitored devices since the last call to getEvents() or clearEvents().

By default all events for all monitored devices are returned, with each event being represented as a namedtuple of all event attributes.

When events are retrieved from an event buffer, they are removed from that buffer as well.

If events are only needed from one device instead of all devices, providing a valid device name as the device_label argument will result in only events from that device being returned.

Events can be received in one of several object types by providing the optional as_type property to the method. Valid values for as_type are the following str values:

- ‘list’: Each event is a list of ordered attributes.
- ‘namedtuple’: Each event is converted to a namedtuple object.
- ‘dict’: Each event converted to a dict object.
- **‘object’: Each event is converted to a DeviceEvent subclass**
  based on the event’s type.

Parameters

- **device_label** (*str*) – Name of device to retrieve events for. If None (the default) returns device events from all devices.
- **as_type** (*str*) – Returned event object type. Default: ‘namedtuple’.

Returns

List of event objects; object type controlled by ‘as_type’.

Return type

tuple

clearEvents(*device_label='all'*)

Clears unread events from the ioHub Server’s Event Buffer(s) so that unneeded events are not discarded.

If device_label is ‘all’, (the default), then events from both the ioHub *Global Event Buffer* and all *Device Event Buffer’s* are cleared.

If device_label is None then all events in the ioHub *Global Event Buffer* are cleared, but the *Device Event Buffers* are unaffected.

If device_label is a str giving a valid device name, then that *Device Event Buffer* is cleared, but the *Global Event Buffer* is not affected.

Parameters

device_label (*str*) – device name, 'all', or None

Returns

None

sendMessageEvent(*text*, *category=''*, *offset=0.0*, *sec_time=None*)

Create and send an Experiment MessageEvent to the ioHub Server for storage in the ioDataStore hdf5 file.

Parameters

- **text** (*str*) – The text message for the message event. 128 char max.
- **category** (*str*) – A str grouping code for the message. Optional. 32 char max.
- **offset** (*float*) – Optional sec.msec offset applied to the message event time stamp. Default 0.
- **sec_time** (*float*) – Absolute sec.msec time stamp for the message in. If not provided, or None, then the MessageEvent is time stamped when this method is called using the global timer (core.getTime()).

cacheMessageEvent(*text*, *category=''*, *offset=0.0*, *sec_time=None*)

Create an Experiment MessageEvent and store in local cache. Message must be sent before it is saved to hdf5 file.

Parameters

- **text** (*str*) – The text message for the message event. 128 char max.
- **category** (*str*) – A str grouping code for the message. Optional. 32 char max.
- **offset** (*float*) – Optional sec.msec offset applied to the message event time stamp. Default 0.
- **sec_time** (*float*) – Absolute sec.msec time stamp for the message in. If not provided, or None, then the MessageEvent is time stamped when this method is called using the global timer (core.getTime()).

createTrialHandlerRecordTable(*trials*, *cv_order=None*)

Create a condition variable table in the ioHub data file based on the a psychopy TrialHandler. By doing so, the iohub data file can contain the DV and IV values used for each trial of an experiment session, along with all the iohub device events recorded by iohub during the session.

Example psychopy code usage:

```
# Load a trial handler and
# create an associated table in the iohub data file
#
from psychopy.data import TrialHandler, importConditions

exp_conditions=importConditions('trial_conditions.xlsx')
trials = TrialHandler(exp_conditions, 1)

# Inform the ioHub server about the TrialHandler
#
io.createTrialHandlerRecordTable(trials)

```

(continues on next page)

(continued from previous page)

```
# Read a row of the trial handler for
# each trial of your experiment
#
for trial in trials:
    # do whatever...


# During the trial, trial variable values can be updated
#
trial['TRIAL_START']=flip_time

# At the end of each trial, before getting
# the next trial handler row, send the trial
# variable states to iohub so they can be stored for future
# reference.
#
io.addTrialHandlerRecord(trial)
```

addTrialHandlerRecord(*cv_row*)

Adds the values from a TriaHandler row / record to the iohub data file for future data analysis use.

Parameters
cv_row

Returns
None

getTime()

Deprecated Method: Use Computer.getTime instead. Remains here for testing time bases between processes only.

syncClock(*clock*)

Synchronise ioHub's internal clock with a given instance of MonotonicClock.

setPriority(*level='normal'*, *disable_gc=False*)

See Computer.setPriority documentation, where current process will be the iohub process.

getPriority()

See Computer.getPriority documentation, where current process will be the iohub process.

getProcessAffinity()

Returns the current **ioHub Process** affinity setting, as a list of 'processor' id's (from 0 to getSystemProcessorCount()-1). A Process's Affinity determines which CPU's or CPU cores a process can run on. By default the ioHub Process can run on any CPU or CPU core.

This method is not supported on OS X at this time.

Parameters
None

Returns

A list of integer values between 0 and
Computer.getSystemProcessorCount()-1, where values in the list indicate processing unit indexes that the ioHub process is able to run on.

Return type
list

`setProcessAffinity`(*processor_list*)

Sets the **ioHub Process** Affinity based on the value of processor_list.

A Process's Affinity determines which CPU's or CPU cores a process can run on. By default the ioHub Process can run on any CPU or CPU core.

The processor_list argument must be a list of 'processor' id's; integers in the range of 0 to Computer.processing_unit_count-1, representing the processing unit indexes that the ioHub Server should be allowed to run on.

If processor_list is given as an empty list, the ioHub Process will be able to run on any processing unit on the computer.

This method is not supported on OS X at this time.

Parameters
processor_list (*list*) – A list of integer values between 0 and Computer.processing_unit_count-1, where values in the list indicate processing unit indexes that the ioHub process is able to run on.

Returns
None

`flushDataStoreFile`()

Manually tell the iohub datastore to flush any events it has buffered in memory to disk. Any cached message events are sent to the iohub server before flushing the iohub datastore.

Parameters
None

Returns
None

`startCustomTasklet`(*task_name*, *task_class_path*, ***class_kwargs*)

Instruct the iohub server to start running a custom tasklet given by task_class_path. It is important that the custom task does not block for any significant amount of time, or the processing of events by the iohub server will be negatively effected.

See the customtask.py demo for an example of how to make a long running task not block the rest of the iohub server.

`stopCustomTasklet`(*task_name*)

Instruct the iohub server to stop the custom task that was previously started by calling self.startCustomTasklet(....). task_name identifies which custom task should be stopped and must match the task_name of a previously started custom task.

`shutdown`()

Tells the ioHub Server to close all ioHub Devices, the ioDataStore, and the connection monitor between the PsychoPy and ioHub Processes. Then end the server process itself.

Parameters
None

Returns
None

`quit`()

Same as the shutdown() method, but has same name as PsychoPy core.quit() so maybe easier to remember.

_startServer(*ioHubConfig=None*, *ioHubConfigAbsPath=None*)

Starts the ioHub Process, storing it's process id, and creating the experiment side device representation for IPC access to public device methods.

_createDeviceList(*monitor_devices_config*)

Create client side iohub device views.

_addDeviceView(*dev_cls_name*, *dev_config*)

Add an iohub device view to self.devices

_sendToHubServer(*tx_data*)

General purpose local <-> iohub server process UDP based request - reply code. The method blocks until the request is fulfilled and and a response is received from the ioHub server.

Parameters

tx_data (*tuple*) – data to send to iohub server

Return (object): response from the ioHub Server process.

_sendExperimentInfo(*experimentInfoDict*)

Sends the experiment info from the experiment config file to the ioHub Server, which passes it to the ioDataStore, determines if the experiment already exists in the hdf5 file based on 'experiment_code', and returns a new or existing experiment ID based on that criteria.

_sendSessionInfo(*sess_info*)

Sends the experiment session info from the experiment config file and the values entered into the session dialog to the ioHub Server, which passes it to the ioDataStore.

The dataStore determines if the session already exists in the experiment file based on 'session_code', and returns a new session ID if session_code is not in use by the experiment.

static **_isErrorReply**(*data*)

Check if an iohub server reply contains an error that should be raised by the local process.

11.7.2 Supported Devices

psychopy.iohub supports several different device types.

Details for each device can be found in the following sections.

Keyboard Device

The iohub Keyboard device provides methods to:

- Check for any new keyboard events that have occurred since the last time keyboard events were checked or cleared.
- Wait until a keyboard event occurs.
- Clear the device of any unread events.
- Get a list of all currently pressed keys.

class psychopy.iohub.client.keyboard.**Keyboard**(*ioclient*, *dev_cls_name*, *dev_config*)

The Keyboard device provides access to KeyboardPress and KeyboardRelease events as well as the current keyboard state.

Examples

A. Print all keyboard events received for 5 seconds:

```
from psychopy.iohub import launchHubServer
from psychopy.core import getTime

# Start the ioHub process. 'io' can now be used during the
# experiment to access iohub devices and read iohub device events.
io = launchHubServer()

keyboard = io.devices.keyboard

# Check for and print any Keyboard events received for 5 seconds.
stime = getTime()
while getTime()-stime < 5.0:
    for e in keyboard.getEvents():
        print(e)

# Stop the ioHub Server
io.quit()
```

B. Wait for a keyboard press event (max of 5 seconds):

```
from psychopy.iohub import launchHubServer
from psychopy.core import getTime

# Start the ioHub process. 'io' can now be used during the
# experiment to access iohub devices and read iohub device events.
io = launchHubServer()

keyboard = io.devices.keyboard

# Wait for a key keypress event ( max wait of 5 seconds )
presses = keyboard.waitForPresses(maxWait=5.0)

print(presses)

# Stop the ioHub Server
io.quit()
```

`_syncDeviceState()`

An optimized iohub server request that receives all device state and event information in one response.

Returns

None

`getKeys`(*keys=None*, *chars=None*, *ignoreKeys=None*, *mods=None*, *duration=None*, *etype=None*, *clear=True*)

Return a list of any KeyboardPress or KeyboardRelease events that have occurred since the last time either:

- this method was called with the kwarg clear=True (default)
- the keyboard.clear() method was called.

Other than the ‘clear’ kwarg, any kwargs that are not None or an empty list are used to filter the possible events that can be returned. If multiple filter criteria are provided, only events that match **all** specified criteria are returned.

If no KeyboardEvent's are found that match the filtering criteria, an empty tuple is returned.

Returned events are sorted by time.

Parameters

- **keys** – Include events where .key in keys.
- **chars** – Include events where .char in chars.
- **ignoreKeys** – Ignore events where .key in ignoreKeys.
- **mods** – Include events where .modifiers include >=1 mods element.
- **duration** – Include KeyboardRelease events where .duration > duration or .duration < -(duration).
- **etype** – Include events that match etype of Keyboard.KEY_PRESS or Keyboard.KEY_RELEASE.
- **clear** – True (default) = clear returned events from event buffer, False = leave the keyboard event buffer unchanged.

Returns

tuple of KeyboardEvent instances, or ()

getPresses(*keys=None*, *chars=None*, *ignoreKeys=None*, *mods=None*, *clear=True*)

See the getKeys() method documentation.

This method is identical, but only returns KeyboardPress events.

getReleases(*keys=None*, *chars=None*, *ignoreKeys=None*, *mods=None*, *duration=None*, *clear=True*)

See the getKeys() method documentation.

This method is identical, but only returns KeyboardRelease events.

property reporting

Specifies if the keyboard device is reporting / recording events.

- True: keyboard events are being reported.
- False: keyboard events are not being reported.

By default, the Keyboard starts reporting events automatically when the ioHub process is started and continues to do so until the process is stopped.

This property can be used to read or set the device reporting state:

```
# Read the reporting state of the keyboard.
is_reporting_keyboard_event = keyboard.reporting

# Stop the keyboard from reporting any new events.
keyboard.reporting = False
```

property state

time values. The key is taken from the originating press event .key field. The time value is time of the key press event.

Note that any pressed, or active, modifier keys are included in the return value.

Returns
dict

Type
Returns all currently pressed keys as a dictionary of key

`waitForKeys`(*maxWait=None*, *keys=None*, *chars=None*, *mods=None*, *duration=None*, *etype=None*, *clear=True*, *checkInterval=0.002*)

Blocks experiment execution until at least one matching KeyboardEvent occurs, or until maxWait seconds has passed since the method was called.

Keyboard events are filtered the same way as in the getKeys() method.

As soon as at least one matching KeyboardEvent occurs prior to maxWait, the matching events are returned as a tuple.

Returned events are sorted by time.

Parameters

- **maxWait** – Maximum seconds method waits for >=1 matching event. If <=0.0, method functions the same as getKeys(). If None, the methods blocks indefinitely.
- **keys** – Include events where .key in keys.
- **chars** – Include events where .char in chars.
- **mods** – Include events where .modifiers include >=1 mods element.
- **duration** – Include KeyboardRelease events where .duration > duration or .duration < -(duration).
- **etype** – Include events that match etype of Keyboard.KEY_PRESS or Keyboard.KEY_RELEASE.
- **clear** – True (default) = clear returned events from event buffer, False = leave the keyboard event buffer unchanged.
- **checkInterval** – The time between geyKeys() calls while waiting. The method sleeps between geyKeys() calls, up until checkInterval*2.0 sec prior to the maxWait. After that time, keyboard events are constantly checked until the method times out.

Returns
tuple of KeyboardEvent instances, or ()

`waitForPresses`(*maxWait=None*, *keys=None*, *chars=None*, *mods=None*, *duration=None*, *clear=True*, *checkInterval=0.002*)

See the waitForKeys() method documentation.

This method is identical, but only returns KeyboardPress events.

`waitForReleases`(*maxWait=None*, *keys=None*, *chars=None*, *mods=None*, *duration=None*, *clear=True*, *checkInterval=0.002*)

See the waitForKeys() method documentation.

This method is identical, but only returns KeyboardRelease events.

Keyboard Events

The Keyboard device can return two types of events, which represent key press and key release actions on the keyboard.

KeyboardPress Event

class `psychopy.iohub.client.keyboard.`**`KeyboardPress`**(*ioe_array*)

An iohub Keyboard device key press event.

property char

The unicode value of the keyboard event, if available. This field is only populated when the keyboard event results in a character that could be printable.

Returns
unicode, ‘’ if no char value is available for the event.

property device

The ioHubDeviceView that is associated with the event, i.e. the iohub device view for the device that generated the event.

Returns
ioHubDeviceView

property modifiers

A list of any modifier keys that were pressed when this keyboard event occurred. Each element of the list contains a keyboard modifier string constant. Possible values are:

- ‘lctrl’, ‘rctrl’
- ‘lshift’, ‘rshift’
- ‘lalt’, ‘ralt’ (labelled as ‘option’ keys on Apple Keyboards)
- ‘lcmd’, ‘rcmd’ (map to the ‘windows’ key(s) on Windows keyboards)
- ‘menu’
- ‘capslock’
- ‘numlock’
- ‘function’ (OS X only)
- ‘modhelp’ (OS X only)

If no modifiers were active when the event occurred, an empty list is returned.

Returns
tuple

property time

The time stamp of the event. Uses the same time base that is used by psychopy.core.getTime()

Returns
float

property type

The event type string constant.

Returns
str

KeyboardRelease Event

class psychopy.iohub.client.keyboard.**KeyboardRelease**(*ioe_array*)

An iohub Keyboard device key release event.

property duration

The duration (in seconds) of the key press. This is calculated by subtracting the current event.time from the associated keypress.time.

If no matching keypress event was reported prior to this event, then 0.0 is returned. This can happen, for example, when the key was pressed prior to psychopy starting to monitor the device. This condition can also happen when keyboard.reset() method is called between the press and release event times.

Returns
float

property pressEventID

The event.id of the associated press event.

The key press id is 0 if no associated KeyboardPress event was found. See the duration property documentation for details on when this can occur.

Returns
unsigned int

property char

The unicode value of the keyboard event, if available. This field is only populated when the keyboard event results in a character that could be printable.

Returns
unicode, ‘’ if no char value is available for the event.

property device

The ioHubDeviceView that is associated with the event, i.e. the iohub device view for the device that generated the event.

Returns
ioHubDeviceView

property modifiers

A list of any modifier keys that were pressed when this keyboard event occurred. Each element of the list contains a keyboard modifier string constant. Possible values are:

- ‘lctrl’, ‘rctrl’
- ‘lshift’, ‘rshift’
- ‘lalt’, ‘ralt’ (labelled as ‘option’ keys on Apple Keyboards)
- ‘lcmd’, ‘rcmd’ (map to the ‘windows’ key(s) on Windows keyboards)
- ‘menu’
- ‘capslock’
- ‘numlock’
- ‘function’ (OS X only)
- ‘modhelp’ (OS X only)

If no modifiers were active when the event occurred, an empty list is returned.

Returns
tuple

property time

The time stamp of the event. Uses the same time base that is used by psychopy.core.getTime()

Returns
float

property type

The event type string constant.

Returns
str

The ioHub Mouse Device

Platforms: Windows, macOS, Linux

class psychopy.iohub.devices.mouse.**MouseDevice**(**args*, ***kwargs*)

getPosition(*return_display_index=False*)

Returns the current position of the ioHub Mouse Device. Mouse Position is in display coordinate units, with 0,0 being the center of the screen.

Parameters

- **return_display_index** – If True, the display index that is
- **returned.** (*associated with the mouse position will also be*)

Returns
If return_display_index is false (default), return (x, y) position of mouse. If return_display_index is True return ((x,y), display_index).

Return type
tuple

setPosition(*pos*, *display_index=None*)

Sets the current position of the ioHub Mouse Device. Mouse position (pos) should be specified in Display coordinate units, with 0,0 being the center of the screen.

Parameters

- **pos** (*(x,y) list or tuple*) – The position, in Display
- **space** (*coordinate*)
- **too.** (*to set the mouse position*)
- **display_index** (*int*) – Optional argument giving the display index
- **None** (*to set the mouse pos within. If*)
- **Display** (*the active ioHub*)
- **used.** (*device index is*)

Returns
new (x,y) position of mouse in Display coordinate space.

Return type
tuple

`getPositionAndDelta`(*return_display_index=False*)

Returns a tuple of tuples, being the current position of the ioHub Mouse Device as an (x,y) tuple, and the amount the mouse position changed the last time it was updated (dx,dy). Mouse Position and Delta are in display coordinate units.

Parameters
`None`

Returns
((x,y), (dx,dy)) position of mouse, change in mouse position, both in Display coordinate space.

Return type
tuple

`getScroll`()

Returns the current vertical scroll value for the mouse. The vertical scroll value changes when the scroll wheel on a mouse is moved up or down. The vertical scroll value is in an arbitrary value space ranging for -32648 to +32648. Scroll position is initialize to 0 when the experiment starts.

Parameters
`None`

Returns
current vertical scroll value.

Return type
int

`setScroll`(*s*)

Sets the current vertical scroll value for the mouse. The vertical scroll value changes when the scroll wheel on a mouse is moved up or down. The vertical scroll value is in an arbitrary value space ranging for -32648 to +32648. Scroll position is initialize to 0 when the experiment starts. This method allows you to change the scroll value to anywhere in the valid value range.

Args (int):
The scroll position you want to set the vertical scroll to. Should be a number between -32648 to +32648.

Returns
current vertical scroll value.

Return type
int

`getEvents`(**args*, ***kwargs*)

Retrieve any DeviceEvents that have occurred since the last call to the device's getEvents() or clearEvents() methods.

Note that calling getEvents() at a device level does not change the Global Event Buffer's contents.

Parameters

- **event_type_id** (*int*) – If specified, provides the ioHub DeviceEvent ID for which events
- **ID** (*should be returned for. Events that have occurred but do not match the event*)
- **class;** (*specified are ignored. Event type ID's can be accessed via the EventConstants*)

- **EventConstants.** (*all available event types are class attributes of*)
- **clearEvents** (*int*) – Can be used to indicate if the events being returned should also be
- **True** (*removed from the device event buffer.*)
- **buffer.** (*being returned. False results in events being left in the device event*)
- **asType** (*str*) – Optional kwarg giving the object type to return events as. Valid values
- **'namedtuple'** (*are*)

Returns

New events that the ioHub has received since the last getEvents() or clearEvents() call to the device. Events are ordered by the ioHub time of each event, older event at index 0. The event object type is determined by the asType parameter passed to the method. By default a namedtuple object is returned for each event.

Return type

(list)

clearEvents(*event_type=None*, *filter_id=None*, *call_proc_events=True*)

Clears any DeviceEvents that have occurred since the last call to the device's getEvents(), or clearEvents() methods.

Note that calling clearEvents() at the device level only clears the given device's event buffer. The ioHub Process's Global Event Buffer is unchanged.

Parameters

None

Returns

None

Mouse Event Types

The Mouse device supports the following event types. Device events returned by getEvents() are automatically converted to either namedtuple or dictionary objects with the same attributes / keys as the associated event class attributes.

class psychopy.iohub.devices.mouse.**MouseMoveEvent**(**args*, ***kwargs*)

MouseMoveEvent's occur when the mouse position changes. Mouse position is mapped to the coordinate space defined in the ioHub configuration file for the Display.

Event Type ID: EventConstants.MOUSE_MOVE

Event Type String: 'MOUSE_MOVE'

time

time of event, in sec.msec format, using psychopy timebase.

x_position

x position of the Mouse when the event occurred; in display coordinate space.

y_position

y position of the Mouse when the event occurred; in display coordinate space.

scroll_x

Horizontal scroll wheel absolute position when the event occurred. macOS only. Always 0 on other OS's.

scroll_y

Vertical scroll wheel absolute position when the event occurred.

modifiers

List of the modifiers that were active when the mouse event occurred, provided as a list of the modifier constant labels.

display_id

The id of the display that the mouse was over when the event occurred. Only supported on Windows at this time. Always 0 on other OS's.

window_id

Window handle reference that the mouse was over when the event occurred (window does not need to have focus).

event_id

The id assigned to the device event. Every event generated by iohub during an experiment session is assigned a unique id, starting from 0.

type

The type id for the event. This is used to create DeviceEvent objects or dictionary representations of an event based on the data from an event list.

class psychopy.iohub.devices.mouse.**MouseDragEvent**(**args*, ***kwargs*)

MouseDragEvents occur when the mouse position changes and at least one mouse button is pressed. Mouse position is mapped to the coordinate space defined in the ioHub configuration file for the Display.

Event Type ID: EventConstants.MOUSE_DRAG

Event Type String: 'MOUSE_DRAG'

time

x position of the Mouse when the event occurred; in display coordinate space.

x_position

x position of the Mouse when the event occurred; in display coordinate space.

button_state

1 if a mouse button press caused the event, 0 if button was released.

button_id

Index of the mouse button that caused the event. MouseConstants.MOUSE_BUTTON_LEFT, MouseConstants.MOUSE_BUTTON_RIGHT and MouseConstants.MOUSE_BUTTON_MIDDLE are int constants representing left, right, and middle buttons of the mouse.

pressed_buttons

All currently pressed button id's logically OR'ed together.

y_position

y position of the Mouse when the event occurred; in display coordinate space.

scroll_x

Horizontal scroll wheel absolute position when the event occurred. macOS only. Always 0 on other OS's.

scroll_y

Vertical scroll wheel absolute position when the event occurred.

`modifiers`

List of the modifiers that were active when the mouse event occurred, provided as a list of the modifier constant labels.

`display_id`

The id of the display that the mouse was over when the event occurred. Only supported on Windows at this time. Always 0 on other OS's.

`window_id`

Window handle reference that the mouse was over when the event occurred (window does not need to have focus).

`event_id`

The id assigned to the device event. Every event generated by iohub during an experiment session is assigned a unique id, starting from 0.

`type`

The type id for the event. This is used to create DeviceEvent objects or dictionary representations of an event based on the data from an event list.

class psychopy.iohub.devices.mouse.**MouseButtonPressEvent**(**args*, ***kwargs*)

MouseButtonPressEvent's are created when a button on the mouse is pressed. The button_state of the event will equal MouseConstants.MOUSE_BUTTON_STATE_PRESSED, and the button that was pressed (button_id) will be MouseConstants.MOUSE_BUTTON_LEFT, MouseConstants.MOUSE_BUTTON_RIGHT, or MouseConstants.MOUSE_BUTTON_MIDDLE, assuming you have a 3 button mouse.

To get the current state of all three buttons on the Mouse Device, the pressed_buttons attribute can be read, which tracks the state of all three mouse buttons as an int that is equal to the sum of any pressed button id's (MouseConstants.MOUSE_BUTTON_LEFT, MouseConstants.MOUSE_BUTTON_RIGHT, or MouseConstants.MOUSE_BUTTON_MIDDLE).

To tell if a given mouse button was depressed when the event occurred, regardless of which button triggered the event, you can use the following:

```
isButtonPressed = event.pressed_buttons &
MouseConstants.MOUSE_BUTTON_xxx == MouseConstants.MOUSE_BUTTON_xxx
```

where xxx is LEFT, RIGHT, or MIDDLE.

For example, if at the time of the event both the left and right mouse buttons were in a pressed state:

```
buttonToCheck=MouseConstants.MOUSE_BUTTON_RIGHT
isButtonPressed = event.pressed_buttons & buttonToCheck ==
buttonToCheck

print isButtonPressed

>> True

buttonToCheck=MouseConstants.MOUSE_BUTTON_LEFT
isButtonPressed = event.pressed_buttons & buttonToCheck ==
buttonToCheck

print isButtonPressed

>> True
```

(continues on next page)

(continued from previous page)

```
buttonToCheck=MouseConstants.MOUSE_BUTTON_MIDDLE
isButtonPressed = event.pressed_buttons & buttonToCheck ==
buttonToCheck

print isButtonPressed

>> False
```

Event Type ID: EventConstants.MOUSE_BUTTON_PRESS

Event Type String: 'MOUSE_BUTTON_PRESS'

:rtype : MouseButtonEvent :param args: :param kwargs:

time

x position of the Mouse when the event occurred; in display coordinate space.

x_position

x position of the Mouse when the event occurred; in display coordinate space.

button_state

1 if a mouse button press caused the event, 0 if button was released.

button_id

Index of the mouse button that caused the event. MouseConstants.MOUSE_BUTTON_LEFT, MouseConstants.MOUSE_BUTTON_RIGHT and MouseConstants.MOUSE_BUTTON_MIDDLE are int constants representing left, right, and middle buttons of the mouse.

pressed_buttons

All currently pressed button id's logically OR'ed together.

y_position

y position of the Mouse when the event occurred; in display coordinate space.

scroll_x

Horizontal scroll wheel absolute position when the event occurred. macOS only. Always 0 on other OS's.

scroll_y

Vertical scroll wheel absolute position when the event occurred.

modifiers

List of the modifiers that were active when the mouse event occurred, provided as a list of the modifier constant labels.

display_id

The id of the display that the mouse was over when the event occurred. Only supported on Windows at this time. Always 0 on other OS's.

window_id

Window handle reference that the mouse was over when the event occurred (window does not need to have focus).

event_id

The id assigned to the device event. Every event generated by iohub during an experiment session is assigned a unique id, starting from 0.

`type`

The type id for the event. This is used to create DeviceEvent objects or dictionary representations of an event based on the data from an event list.

class psychopy.iohub.devices.mouse.**MouseButtonReleaseEvent**(**args*, ***kwargs*)

MouseButtonUpEvent's are created when a button on the mouse is released.

The button_state of the event will equal MouseConstants.MOUSE_BUTTON_STATE_RELEASED, and the button that was pressed (button_id) will be MouseConstants.MOUSE_BUTTON_LEFT, MouseConstants.MOUSE_BUTTON_RIGHT, or MouseConstants.MOUSE_BUTTON_MIDDLE, assuming you have a 3 button mouse.

Event Type ID: EventConstants.MOUSE_BUTTON_RELEASE

Event Type String: 'MOUSE_BUTTON_RELEASE'

:rtype : MouseButtonEvent :param args: :param kwargs:

`time`

x position of the Mouse when the event occurred; in display coordinate space.

`x_position`

x position of the Mouse when the event occurred; in display coordinate space.

`button_state`

1 if a mouse button press caused the event, 0 if button was released.

`button_id`

Index of the mouse button that caused the event. MouseConstants.MOUSE_BUTTON_LEFT, MouseConstants.MOUSE_BUTTON_RIGHT and MouseConstants.MOUSE_BUTTON_MIDDLE are int constants representing left, right, and middle buttons of the mouse.

`pressed_buttons`

All currently pressed button id's logically OR'ed together.

`y_position`

y position of the Mouse when the event occurred; in display coordinate space.

`scroll_x`

Horizontal scroll wheel absolute position when the event occurred. macOS only. Always 0 on other OS's.

`scroll_y`

Vertical scroll wheel absolute position when the event occurred.

`modifiers`

List of the modifiers that were active when the mouse event occurred, provided as a list of the modifier constant labels.

`display_id`

The id of the display that the mouse was over when the event occurred. Only supported on Windows at this time. Always 0 on other OS's.

`window_id`

Window handle reference that the mouse was over when the event occurred (window does not need to have focus).

`event_id`

The id assigned to the device event. Every event generated by iohub during an experiment session is assigned a unique id, starting from 0.

`type`

The type id for the event. This is used to create DeviceEvent objects or dictionary representations of an event based on the data from an event list.

`class psychopy.iohub.devices.mouse.MouseScrollEvent(*args, **kwargs)`

MouseScrollEvent's are generated when the scroll wheel on the Mouse Device (if it has one) is moved. Vertical scrolling is supported on all operating systems, horizontal scrolling is only supported on OS X.

Each MouseScrollEvent provides the number of units the wheel was turned in each supported dimension, as well as the absolute scroll value for of each supported dimension.

Event Type ID: EventConstants.MOUSE_SCROLL

Event Type String: 'MOUSE_SCROLL'

:rtype : MouseScrollEvent :param args: :param kwargs:

`time`

x position of the Mouse when the event occurred; in display coordinate space.

`x_position`

x position of the Mouse when the event occurred; in display coordinate space.

`button_state`

1 if a mouse button press caused the event, 0 if button was released.

`button_id`

Index of the mouse button that caused the event. MouseConstants.MOUSE_BUTTON_LEFT, MouseConstants.MOUSE_BUTTON_RIGHT and MouseConstants.MOUSE_BUTTON_MIDDLE are int constants representing left, right, and middle buttons of the mouse.

`pressed_buttons`

All currently pressed button id's logically OR'ed together.

`y_position`

y position of the Mouse when the event occurred; in display coordinate space.

`scroll_x`

Horizontal scroll wheel absolute position when the event occurred. macOS only. Always 0 on other OS's.

`scroll_dx`

Horizontal scroll wheel position change when the event occurred. macOS only. Always 0 on other OS's.

`scroll_y`

Vertical scroll wheel absolute position when the event occurred.

`scroll_dy`

Vertical scroll wheel position change when the event occurred.

`modifiers`

List of the modifiers that were active when the mouse event occurred, provided as a list of the modifier constant labels.

`display_id`

The id of the display that the mouse was over when the event occurred. Only supported on Windows at this time. Always 0 on other OS's.

window_id

Window handle reference that the mouse was over when the event occurred (window does not need to have focus).

event_id

The id assigned to the device event. Every event generated by iohub during an experiment session is assigned a unique id, starting from 0.

type

The type id for the event. This is used to create DeviceEvent objects or dictionary representations of an event based on the data from an event list.

ioHub Common Eye Tracker Interface

The iohub common eye tracker interface provides a consistent way to configure and collected data from several different eye tracker manufacturers.

Supported Eye Trackers

The following eye trackers are currently supported by iohub.

MouseGaze

MouseGaze simulates an eye tracker using the computer Mouse.

Platforms:

- Windows 7 / 10
- Linux
- macOS

Required Python Version:

- Python 3.6 +

Supported Models:

- Any Mouse. ;)

Additional Software Requirements

None

EyeTracker Class

class psychopy.iohub.devices.eyetracker.hw.mouse.**EyeTracker**

To start iohub with a Mouse Simulated eye tracker, add the full iohub device name as a kwarg passed to launchHubServer:

```
eyetracker.hw.mouse.EyeTracker
```

Examples

A. Start ioHub with the Mouse Simulated eye tracker:

```python
from psychopy.iohub import launchHubServer
from psychopy.core import getTime, wait

iohub_config = {'eyetracker.hw.mouse.EyeTracker': {}}

io = launchHubServer(**iohub_config)

# Get the eye tracker device.
tracker = io.devices.tracker
```

B. Print all eye tracker events received for 2 seconds:

```python
# Check for and print any eye tracker events received...
tracker.setRecordingState(True)

stime = getTime()
while getTime()-stime < 2.0:
    for e in tracker.getEvents():
        print(e)
```

C. Print current eye position for 5 seconds:

```python
# Check for and print current eye position every 100 msec.
stime = getTime()
while getTime()-stime < 5.0:
    print(tracker.getPosition())
    wait(0.1)

tracker.setRecordingState(False)

# Stop the ioHub Server
io.quit()
```

clearEvents(*event_type=None*, *filter_id=None*, *call_proc_events=True*)

Clears any DeviceEvents that have occurred since the last call to the device's getEvents(), or clearEvents() methods.

Note that calling clearEvents() at the device level only clears the given device's event buffer. The ioHub Process's Global Event Buffer is unchanged.

Parameters
: **None**

Returns
: None

enableEventReporting(*enabled=True*)

enableEventReporting is functionally identical to the eye tracker device specific setRecordingState method.

getConfiguration()

Retrieve the configuration settings information used to create the device instance. This will the default settings for the device, found in iohub.devices.<device_name>.default_<device_name>.yaml, updated with any device settings provided via launchHubServer(…).

Changing any values in the returned dictionary has no effect on the device state.

Parameters
`None`

Returns
The dictionary of the device configuration settings used to create the device.

Return type
(dict)

`getEvents(*args, **kwargs)`

Retrieve any DeviceEvents that have occurred since the last call to the device's getEvents() or clearEvents() methods.

Note that calling getEvents() at a device level does not change the Global Event Buffer's contents.

Parameters

- **event_type_id** (*int*) – If specified, provides the ioHub DeviceEvent ID for which events
- **ID** (*should be returned for. Events that have occurred but do not match the event*)
- **class;** (*specified are ignored. Event type ID's can be accessed via the EventConstants*)
- **EventConstants.** (*all available event types are class attributes of*)
- **clearEvents** (*int*) – Can be used to indicate if the events being returned should also be
- **True** (*removed from the device event buffer.*)
- **buffer.** (*being returned. False results in events being left in the device event*)
- **asType** (*str*) – Optional kwarg giving the object type to return events as. Valid values
- **'namedtuple'** (*are*)

Returns
New events that the ioHub has received since the last getEvents() or clearEvents() call to the device. Events are ordered by the ioHub time of each event, older event at index 0. The event object type is determined by the asType parameter passed to the method. By default a namedtuple object is returned for each event.

Return type
(list)

`getLastGazePosition()`

The getLastGazePosition method returns the most recent eye gaze position received from the Eye Tracker. This is the position on the calibrated 2D surface that the eye tracker is reporting as the current eye position. The units are in the units in use by the ioHub Display device.

If binocular recording is being performed, the average position of both eyes is returned.

If no samples have been received from the eye tracker, or the eye tracker is not currently recording data, None is returned.

Parameters
`None`

Returns

If this method is not supported by the eye tracker interface, EyeTrackerConstants.EYETRACKER_INTERFACE_METHOD_NOT_SUPPORTED is returned.

None: If the eye tracker is not currently recording data or no eye samples have been received.

tuple: Latest (gaze_x,gaze_y) position of the eye(s)

Return type

int

`getLastSample()`

The getLastSample method returns the most recent eye sample received from the Eye Tracker. The Eye Tracker must be in a recording state for a sample event to be returned, otherwise None is returned.

Parameters

`None`

Returns

If this method is not supported by the eye tracker interface, EyeTrackerConstants.FUNCTIONALITY_NOT_SUPPORTED is returned.

None: If the eye tracker is not currently recording data.

EyeSample: If the eye tracker is recording in a monocular tracking mode, the latest sample event of this event type is returned.

BinocularEyeSample: If the eye tracker is recording in a binocular tracking mode, the latest sample event of this event type is returned.

Return type

int

`getPosition()`

See getLastGazePosition().

`isRecordingEnabled()`

isRecordingEnabled returns the recording state from the eye tracking device.

Returns

True == the device is recording data; False == Recording is not occurring

Return type

bool

`runSetupProcedure`(*calibration_args={}*)

runSetupProcedure displays a mock calibration procedure. No calibration is actually done.

`setRecordingState`(*recording*)

setRecordingState is used to start or stop the recording of data from the eye tracking device.

`trackerSec()`

Same as trackerTime().

`trackerTime()`

Current eye tracker time.

Returns

current eye tracker time in seconds.

Return type

float

Supported Event Types

MouseGaze generates monocular eye samples. A MonocularEyeSampleEvent is created every 10 or 20 msec depending on the sampling_rate set for the device.

The following fields of the MonocularEyeSample event are supported:

class psychopy.iohub.devices.eyetracker.**BinocularEyeSampleEvent**(**args*, ***kwargs*)

The BinocularEyeSampleEvent event represents the eye position and eye attribute data collected from one frame or reading of an eye tracker device that is recording both eyes of a participant.

Event Type ID: EventConstants.BINOCULAR_EYE_SAMPLE

Event Type String: 'BINOCULAR_EYE_SAMPLE'

time

time of event, in sec.msec format, using psychopy timebase.

gaze_x

The horizontal position of MouseGaze on the computer screen, in Display Coordinate Type Units. Calibration must be done prior to reading (meaningful) gaze data. Uses Gazepoint LPOGX field.

gaze_y

The vertical position of MouseGaze on the computer screen, in Display Coordinate Type Units. Calibration must be done prior to reading (meaningful) gaze data. Uses Gazepoint LPOGY field.

left_pupil_measure_1

MouseGaze pupil diameter, static at 5 mm.

status

Indicates if eye sample contains 'valid' position data. 0 = MouseGaze position is valid. 2 = MouseGaze position is missing (in simulated blink).

MouseGaze also creates basic fixation, saccade, and blink events based on mouse event data.

class psychopy.iohub.devices.eyetracker.**FixationStartEvent**(**args*, ***kwargs*)

A FixationStartEvent is generated when the beginning of an eye fixation (in very general terms, a period of relatively stable eye position) is detected by the eye trackers sample parsing algorithms.

Event Type ID: EventConstants.FIXATION_START

Event Type String: 'FIXATION_START'

time

time of event, in sec.msec format, using psychopy timebase.

eye

EyeTrackerConstants.RIGHT_EYE.

gaze_x

The horizontal 'eye' position on the computer screen at the start of the fixation. Units are same as Window.

gaze_y

The vertical eye position on the computer screen at the start of the fixation. Units are same as Window.

class psychopy.iohub.devices.eyetracker.**FixationEndEvent**(**args*, ***kwargs*)

A FixationEndEvent is generated when the end of an eye fixation (in very general terms, a period of relatively stable eye position) is detected by the eye trackers sample parsing algorithms.

Event Type ID: EventConstants.FIXATION_END

Event Type String: 'FIXATION_END'

`time`

time of event, in sec.msec format, using psychopy timebase.

`eye`

EyeTrackerConstants.RIGHT_EYE.

`start_gaze_x`

The horizontal 'eye' position on the computer screen at the start of the fixation. Units are same as Window.

`start_gaze_y`

The vertical 'eye' position on the computer screen at the start of the fixation. Units are same as Window.

`end_gaze_x`

The horizontal 'eye' position on the computer screen at the end of the fixation. Units are same as Window.

`end_gaze_y`

The vertical 'eye' position on the computer screen at the end of the fixation. Units are same as Window.

`average_gaze_x`

Average calibrated horizontal eye position during the fixation, specified in Display Units.

`average_gaze_y`

Average calibrated vertical eye position during the fixation, specified in Display Units.

`duration`

Duration of the fixation in sec.msec format.

Default Device Settings

```
eyetracker.hw.mouse.EyeTracker:
    #   True = Automatically start reporting events for this device when the experiment␣
↪starts.
    #   False = Do not start reporting events for this device until␣
↪enableEventReporting(True)
    #   is called for the device.
    auto_report_events: False

    # Should eye tracker events be saved to the ioHub DataStore file when the device
    # is recording data ?
    save_events: True

    # Should eye tracker events be sent to the Experiment process when the device
    # is recording data ?
    stream_events: True

    # How many eye events (including samples) should be saved in the ioHub event buffer␣
↪before
    # old eye events start being replaced by new events. When the event buffer reaches
    # the maximum event length of the buffer defined here, older events will start to be␣
↪dropped.
    event_buffer_length: 1024
    runtime_settings:
        # How many samples / second should Mousegaze Generate.
        # 50 or 100 hz are supported.
        sampling_rate: 50
```

(continues on next page)

(continued from previous page)

```
        # MouseGaze always generates Monocular Right eye samples.
        track_eyes: RIGHT_EYE

    controls:
        # Mouse Button used to make a MouseGaze position change.
        # LEFT_BUTTON, MIDDLE_BUTTON, RIGHT_BUTTON.
        move: RIGHT_BUTTON

        # Mouse Button(s) used to make MouseGaze generate a blink event.
        # LEFT_BUTTON, MIDDLE_BUTTON, RIGHT_BUTTON.
        blink: [LEFT_BUTTON, RIGHT_BUTTON]

        # Threshold for saccade generation. Specified in visual degrees.
        saccade_threshold: 0.5

    # MouseGaze creates (minimally populated) fixation, saccade, and blink events.
    monitor_event_types: [MonocularEyeSampleEvent, FixationStartEvent, FixationEndEvent, SaccadeStartEvent, SaccadeEndEvent, BlinkStartEvent, BlinkEndEvent]
```

Last Updated: March, 2021

11.7.3 psychopy.iohub Specific Requirements

Computer Specifications

The design / requirements of your experiment itself can obviously influence what the minimum computer specification should be to provide good timing / performance.

The dual process design when running using psychopy.iohub also influences the minimum suggested specifications as follows:

- Intel i5 or i7 CPU. A minimum of **two** CPU cores is needed.
- 8 GB of RAM
- Windows 7 +, OS X 10.7.5 +, or Linux Kernel 2.6 +

Please see the *Recommended hardware* section for further information that applies to in general.

Usage Considerations

When using psychopy.iohub, the following constrains should be noted:

1. The pyglet graphics backend must be used; pygame is not supported.
2. ioHub devices that report position data use the unit type defined by the Window. However, position data is reported using the full screen area and size the window was created in. Therefore, for accurate window position reporting, the window must be made full screen.
3. On macOS, Assistive Device support must be enabled when using psychopy.iohub:
   - See instructions for OS X 10.7 - 10.8.5
   - See instructions for For OS X 10.9 +

11.7.4 The ioHub Computer Device

Platforms: Windows, macOS, Linux

`class psychopy.iohub.devices.Computer`

Computer provides access to OS and Process level functionality:

- Read the current time in sec.msec format. The time base used is shared by the ioHub and PsychoPy processes.
- Access the iohub and psychopy psutil.Process objects
- Get / set process priority and affinity.
- Read system memory and CPU usage

Computer contains only static methods and class attributes. Therefore all supported functionality can be accessed directly from the Computer class itself; an instance of the class never needs to be created.

`static autoAssignAffinities()`

Auto sets the PsychoPy Process and ioHub Process affinities based on some very simple logic.

It is not known at this time if the implementation of this method makes any sense in terms of actually improving performance. Field tests and feedback will need to occur, based on which the algorithm can be improved.

Currently:

- If the system is detected to have 1 processing unit, or greater

than 8 processing units, nothing is done by the method. * For a system that has two processing units, the PsychoPy Process is assigned to index 0, ioHub Process assigned to 1. * For a system that has four processing units, the PsychoPy Process is assigned to index's 0,1 and the ioHub Process assigned to 2,3. * For a system that has eight processing units, the PsychoPy Process is assigned to index 2,3, ioHub Process assigned to 4,5. All other processes running on the OS are attempted to be assigned to indexes 0,1,6,7.

Parameters
`None`

Returns
None

`core_count = 2`

The number of cpu cores available on the computer. Hyperthreads are NOT included.

`current_process = psutil.Process(pid=1996, name='python.exe', status='running', started='11:32:24')`

Access to the psutil.Process class for the current system Process.

`static getCPUTimeInfo(`*percpu=False*`)`

Return a float representing the current CPU utilization as a percentage.

Parameters
`percpu` (*bool*) – If True, a list of cputimes objects is returned, one for each processing unit for the computer. If False, only a single cputimes object is returned.

Returns
(user=float, system=float, idle=float)

Return type
object

`static getCurrentProcess()`

Get the current / Local process.

On Windows and Linux, this is a psutil.Process class instance.

Parameters
`None`

Returns
Process object for the current system process.

Return type
object

`static getCurrentProcessAffinity()`

Returns a list of 'processor' ID's (from 0 to Computer.processing_unit_count-1) that the current (calling) process is able to run on.

Parameters
`None`

Returns
None

`static getIoHubProcess()`

Get the ioHub Process.

On Windows and Linux, this is a psutil.Process class instance.

Parameters
`None`

Returns
Process object for the ioHub Process.

Return type
object

`static getPhysicalSystemMemoryInfo()`

Return a class containing information about current memory usage.

Parameters
`None`

Returns
(total=long, available=long, percent=float, used=long, free=long)

Return type
vmem

Where:

- vmem.total: the total amount of memory in bytes.
- vmem.available: the available amount of memory in bytes.
- vmem.percent: the percent of memory in use by the system.
- vmem.used: the used amount of memory in bytes.
- vmem.free: the amount of memory that is free in bytes.On Windows,

this is the same as vmem.available.

static getPriority()

Returns the current processes priority as a string.

This method is not supported on OS X.

Returns
'normal', 'high', or 'realtime'

static getProcessAffinities()

Retrieve the current PsychoPy Process affinity list and ioHub Process affinity list.

For example, on a 2 core CPU with hyper-threading, the possible 'processor' list would be [0,1,2,3], and by default both the PsychoPy and ioHub Processes can run on any of these 'processors', so:

```
psychoCPUs,ioHubCPUS=Computer.getProcessAffinities()
print psychoCPUs,ioHubCPUS

>> [0,1,2,3], [0,1,2,3]
```

If Computer.setProcessAffinities was used to set the PsychoPy Process to core 1 (index 0 and 1) and the ioHub Process to core 2 (index 2 and 3), with each using both hyper threads of the given core, the set call would look like:

```
Computer.setProcessAffinities([0,1],[2,3])

psychoCPUs,ioHubCPUS=Computer.getProcessAffinities()
print psychoCPUs,ioHubCPUS

>> [0,1], [2,3]
```

If the ioHub is not being used (i.e self.hub is None), then only the PsychoPy Process affinity list will be returned and None will be returned for the ioHub Process affinity:

```
psychoCPUs,ioHubCPUS=Computer.getProcessAffinities()
print psychoCPUs,ioHubCPUS

>> [0,1,2,3], None
```

But in this case, why are you using the ioHub package at all? ;)

This method is not supported on OS X.

Parameters
`None`

Returns
PsychoPy Process affinity ID list and ioHub Process affinity ID list.

Return type
(list,list) Tuple of two lists

static getProcessAffinityByID(*process_id*)

Returns a list of 'processor' ID's (from 0 to Computer.processing_unit_count-1) that the process with the provided processID is able to run on.

Parameters

- **processID** (*int*) – The system process ID that the affinity should
- **for.** (*be set*)

Returns
list of int processor ID's to set process with the given processID too. An empty list means all processors.

Return type
processorList (list)

`static getProcessingUnitCount()`

Return the number of *processing units* available on the current computer. Processing Units include: cpu's, cpu cores, and hyper threads.

Notes:

- processing_unit_count = num_cpus*num_cores_per_cpu*num_hyperthreads.
- For single core CPU's, num_cores_per_cpu = 1.
- For CPU's that do not support hyperthreading, num_hyperthreads =

1, otherwise num_hyperthreads = 2.

Parameters
`None`

Returns
the number of processing units on the computer.

Return type
int

`static getTime()`

Returns the current sec.msec-msec time of the system.

The underlying timer that is used is based on OS and Python version. Three requirements exist for the ioHub time base implementation:

- The Python interpreter does not apply an offset to the times

returned based on when the timer module being used was loaded or when the timer function first called was first called. * The timer implementation used must be monotonic and report elapsed time between calls, 'not' CPU usage time. * The timer implementation must provide a resolution of 50 usec or better.

Given the above requirements, ioHub selects a timer implementation as follows:

- On Windows, the Windows Query Performance Counter API is used

using ctypes access. * On other OS's, if the Python version being used is 2.6 or lower, time.time is used. For Python 2.7 and above, the timeit.default_timer function is used.

Parameters
`None`

Returns
None

`in_high_priority_mode = False`

True if the current process is currently in high or real-time priority mode (enabled by calling Computer.setPriority()).

`iohub_process = None`

The psutil Process object for the ioHub Process.

`iohub_process_id = None`

The OS process ID of the ioHub Process.

`is_iohub_process = False`

True if the current process is the ioHub Server Process. False if the current process is the Experiment Runtime Process.

`platform = 'win32'`

The name of the current operating system Python is running on.

`processing_unit_count = 2`

Attribute representing the number of *processing units* available on the current computer. This includes cpu's, cpu cores, and hyperthreads.

processing_unit_count = num_cpus * cores_per_cpu * num_hyperthreads

where:

- num_cpus: Number of CPU chips on the motherboard (usually 1 now).
- cores_per_cpu: Number of processing cores per CPU (2,4 is common)
- num_hyperthreads: Hyper-threaded cores = 2, otherwise 1.

`psychopy_process = None`

If Computer class is on the iohub server process, psychopy_process is the psychopy process created from the pid passed to iohub on startup. The iohub server checks that this process exists (server.checkForPsychopyProcess()) and shuts down if it does not.

`pybits = 64`

32 or 64. Note that when a Python 32 bit runtime is used a 64 bit OS sysbits will equal 32.

Type

Python Env. bits

`static setAllOtherProcessesAffinity`(*processor_list*, *exclude_process_id_list=[]*)

Sets the affinity for all OS Processes other than those specified in the exclude_process_id_list, to the processing unit indexes specified in processor_list. Valid values in the processor_list are between 0 to Computer.processing_unit_count-1.

exclude_process_id_list should be a list of OS Process ID integers, or an empty list (indicating to set the affiinty to all processing units).

Note that the OS may not allow the calling process to set the affinity of every other process running on the system. For example, some system level processing can not have their affinity set by a user level application.

However, in general, many processes can have their affinity set by another user process.

Parameters

- **processor_list** (*list*) – list of int processor ID's to set all OS
- **processors.** (*Processes to. An empty list means all*)
- **exclude_process_id_list** (*list*) – A list of process ID's that
- **changed.** (*should not have their process affinity settings*)

Returns

None

`static setCurrentProcessAffinity`(*processorList*)

Sets the list of 'processor' ID's (from 0 to Computer.processing_unit_count-1) that the current (calling) process should only be allowed to run on.

Parameters

- **processorList** (*list*) – list of int processor ID's to set the
- **processors.** (*current Process affinity to. An empty list means all*)

Returns

None

`static setPriority`(*level='normal'*, *disable_gc=False*)

Attempts to change the current processes priority based on level. Supported levels are:

- 'normal': sets the current process priority to

NORMAL_PRIORITY_CLASS on Windows, or to the processes original nice value on Linux. * 'high': sets the current process priority to HIGH_PRIORITY_CLASS on Windows, or to a nice value of -10 value on Linux. * 'realtime': sets the current process priority to REALTIME_PRIORITY_CLASS on Windows, or to a nice value of -18 value on Linux.

If level is 'normal', Python GC is also enabled. If level is 'high' or 'realtime', and disable_gc is True, then the Python garbage collection (GC) thread is suspended.

This method is not supported on OS X.

Returns

Priority level of process when method returns.

`static setProcessAffinities`(*experimentProcessorList*, *ioHubProcessorList*)

Sets the processor affinity for the PsychoPy Process and the ioHub Process.

For example, on a 2 core CPU with hyper-threading, the possible 'processor' list would be [0,1,2,3], and by default both the experiment and ioHub server processes can run on any of these 'processors', so to have both processes have all processors available (which is the default), you would call:

```
Computer.setProcessAffinities([0,1,2,3], [0,1,2,3])

# check the process affinities
psychoCPUs,ioHubCPUS=Computer.getProcessAffinities()
print psychoCPUs,ioHubCPUS

>> [0,1,2,3], [0,1,2,3]
```

based on the above CPU example.

If setProcessAffinities was used to set the experiment process to core 1 (index 0,1) and the ioHub server process to core 2 (index 2,3), with each using both hyper threads of the given core, the set call would look like:

```
Computer.setProcessAffinities([0,1],[2,3])

# check the process affinities
psychoCPUs,ioHubCPUS=Computer.getProcessAffinities()
print psychoCPUs,ioHubCPUS

>> [0,1], [2,3]
```

Parameters

- **experimentProcessorList** (*list*) – list of int processor ID's to set
- **all** (*the PsychoPy Process affinity to. An empty list means*)
- **processors.** (*ioHub Process affinity to. An empty list means all*)
- **ioHubProcessorList** (*list*) – list of int processor ID's to set the
- **processors.**

Returns
None

`static setProcessAffinityByID(process_id, processor_list)`

Sets the list of 'processor' ID's (from 0 to Computer.processing_unit_count-1) that the process with the provided OS Process ID is able to run on.

Parameters

- **processID** (*int*) – The system process ID that the affinity should
- **for.** (*be set*)
- **processorList** (*list*) – list of int processor ID's to set process
- **processors.** (*with the given processID too. An empty list means all*)

Returns
None

`static syncClock(params)`

Sync parameters between Computer.global_clock and a given dict.

Parameters
params (*dict*) – Dict of attributes and values to apply to the computer's global clock. See *psychopy.clock.MonotonicClock* for what attributes to include.

Computer Device Default Configuration Settings

The computer Device is enabled automatically and has no configuration settings in the iohub_config.yaml.

Computer Device Events

The Computer Device does not generate any ioHub Events.

Notes and Considerations

None at this time.

11.7.5 Serial Port Device and Events

TBC

11.8 psychopy.tools - miscellaneous tools

Container for all miscellaneous functions and classes

11.8.1 psychopy.tools.colorspacetools

Tools related to working with various color spaces.

The routines provided in the module are used to transform color coordinates between spaces. Most of the functions here are *vectorized*, allowing for array inputs to convert multiple color values at once.

As of version 2021.0 of PsychoPy, users ought to use the `Color` class for working with color coordinate values.

CIELAB

Conversion functions for the *CIELAB* and *CIELCH* color space.

cielab2rgb(lab[, whiteXYZ, ...])	Transform CIE L*a*b* (1976) color space coordinates to RGB tristimulus values.
cielch2rgb(lch[, whiteXYZ, ...])	Transform CIE *L*C*h** coordinates to RGB tristimulus values.

psychopy.tools.colorspacetools.cielab2rgb

psychopy.tools.colorspacetools.**cielab2rgb**(*lab*, *whiteXYZ=None*, *conversionMatrix=None*, *transferFunc=None*, *clip=False*, ***kwargs*)

Transform CIE L*a*b* (1976) color space coordinates to RGB tristimulus values.

CIE L*a*b* are first transformed into CIE XYZ (1931) color space, then the RGB conversion is applied. By default, the sRGB conversion matrix is used with a reference D65 white point. You may specify your own RGB conversion matrix and white point (in CIE XYZ) appropriate for your display.

Parameters

- **lab** (*tuple, list or ndarray*) – 1-, 2-, 3-D vector of CIE L*a*b* coordinates to convert. The last dimension should be length-3 in all cases specifying a single coordinate.
- **whiteXYZ** (*tuple, list or ndarray*) – 1-D vector coordinate of the white point in CIE-XYZ color space. Must be the same white point needed by the conversion matrix. The default white point is D65 if None is specified, defined as X, Y, Z = 0.9505, 1.0000, 1.0890.
- **conversionMatrix** (*tuple, list or ndarray*) – 3x3 conversion matrix to transform CIE-XYZ to RGB values. The default matrix is sRGB with a D65 white point if None is specified. Note that values must be gamma corrected to appear correctly according to the sRGB standard.
- **transferFunc** (*pyfunc or None*) – Signature of the transfer function to use. If None, values are kept as linear RGB (it's assumed your display is gamma corrected via the hardware CLUT). The TF must be appropriate for the conversion matrix supplied (default is sRGB). Additional arguments to 'transferFunc' can be passed by specifying them as keyword arguments. Gamma functions that come with PsychoPy are 'srgbTF' and 'rec709TF', see their docs for more information.
- **clip** (*bool*) – Make all output values representable by the display. However, colors outside of the display's gamut may not be valid!

Returns
Array of RGB tristimulus values.

Return type
ndarray

Example

Converting a CIE L*a*b* color to linear RGB:

```python
import psychopy.tools.colorspacetools as cst
cielabColor = (53.0, -20.0, 0.0)  # greenish color (L*, a*, b*)
rgbColor = cst.cielab2rgb(cielabColor)
```

Using a transfer function to convert to sRGB:

```python
rgbColor = cst.cielab2rgb(cielabColor, transferFunc=cst.srgbTF)
```

psychopy.tools.colorspacetools.cielch2rgb

psychopy.tools.colorspacetools.**cielch2rgb**(*lch*, *whiteXYZ=None*, *conversionMatrix=None*, *transferFunc=None*, *clip=False*, ***kwargs*)

Transform CIE *L*C*h** coordinates to RGB tristimulus values.

Parameters

- **lch** (*tuple, list or ndarray*) – 1-, 2-, 3-D vector of CIE *L*C*h** coordinates to convert. The last dimension should be length-3 in all cases specifying a single coordinate. The hue angle *h is expected in degrees.
- **whiteXYZ** (*tuple, list or ndarray*) – 1-D vector coordinate of the white point in CIE-XYZ color space. Must be the same white point needed by the conversion matrix. The default white point is D65 if None is specified, defined as X, Y, Z = 0.9505, 1.0000, 1.0890
- **conversionMatrix** (*tuple, list or ndarray*) – 3x3 conversion matrix to transform CIE-XYZ to RGB values. The default matrix is sRGB with a D65 white point if None is specified. Note that values must be gamma corrected to appear correctly according to the sRGB standard.
- **transferFunc** (*pyfunc or None*) – Signature of the transfer function to use. If None, values are kept as linear RGB (it's assumed your display is gamma corrected via the hardware CLUT). The TF must be appropriate for the conversion matrix supplied. Additional arguments to 'transferFunc' can be passed by specifying them as keyword arguments. Gamma functions that come with PsychoPy are 'srgbTF' and 'rec709TF', see their docs for more information.
- **clip** (*boolean*) – Make all output values representable by the display. However, colors outside of the display's gamut may not be valid!

Returns

array of RGB tristimulus values

Return type

ndarray

DKL

Conversion functions for the *Derrington Krauskopf and Lennie* (DKL) color space.

dkl2rgb(dkl[, conversionMatrix])	Convert from DKL color space (Derrington, Krauskopf & Lennie) to RGB.
dklCart2rgb(LUM, LM, S[, conversionMatrix])	Like dkl2rgb except that it uses cartesian coords (LM,S,LUM) rather than spherical coords for DKL (elev, azim, contr).
rgb2dklCart(picture[, conversionMatrix])	Convert an RGB image into Cartesian DKL space.

psychopy.tools.colorspacetools.dkl2rgb

psychopy.tools.colorspacetools.**dkl2rgb**(*dkl*, *conversionMatrix=None*)

Convert from DKL color space (Derrington, Krauskopf & Lennie) to RGB.

Requires a conversion matrix, which will be generated from generic Sony Trinitron phosphors if not supplied (note that this will not be an accurate representation of the color space unless you supply a conversion matrix).

Examples

Converting a single DKL color to RGB:

```
dkl = [90, 0, 1]
rgb = dkl2rgb(dkl, conversionMatrix)
```

psychopy.tools.colorspacetools.dklCart2rgb

psychopy.tools.colorspacetools.**dklCart2rgb**(*LUM*, *LM*, *S*, *conversionMatrix=None*)

Like dkl2rgb except that it uses cartesian coords (LM,S,LUM) rather than spherical coords for DKL (elev, azim, contr).

NB: this may return rgb values >1 or <-1

psychopy.tools.colorspacetools.rgb2dklCart

psychopy.tools.colorspacetools.**rgb2dklCart**(*picture*, *conversionMatrix=None*)

Convert an RGB image into Cartesian DKL space.

HSV

Conversion functions for the *Hue-Saturation-Value* (HSV) color space.

hsv2rgb(hsv_Nx3)	Convert from HSV color space to RGB gun values.
rgb2hsv(rgb)	Convert values from linear RGB to HSV colorspace.

psychopy.tools.colorspacetools.hsv2rgb

psychopy.tools.colorspacetools.**hsv2rgb**(*hsv_Nx3*)

Convert from HSV color space to RGB gun values.

usage:

```
rgb_Nx3 = hsv2rgb(hsv_Nx3)
```

Note that in some uses of HSV space the Hue component is given in radians or cycles (range 0:1]). In this version H is given in degrees (0:360).

Also note that the RGB output ranges -1:1, in keeping with other PsychoPy functions.

psychopy.tools.colorspacetools.rgb2hsv

psychopy.tools.colorspacetools.**rgb2hsv**(*rgb*)

Convert values from linear RGB to HSV colorspace.

Parameters

rgb (*array_like*) – 1-, 2-, 3-D vector of RGB coordinates to convert. The last dimension should be length-3 in all cases, specifying a single coordinate.

Returns

HSV values with the same shape as the input.

Return type

ndarray

LMS

lms2rgb(lms_Nx3[, conversionMatrix])	Convert from cone space (Long, Medium, Short) to RGB.
rgb2lms(rgb_Nx3[, conversionMatrix])	Convert from RGB to cone space (LMS).

psychopy.tools.colorspacetools.lms2rgb

psychopy.tools.colorspacetools.**lms2rgb**(*lms_Nx3*, *conversionMatrix=None*)

Convert from cone space (Long, Medium, Short) to RGB.

Requires a conversion matrix, which will be generated from generic Sony Trinitron phosphors if not supplied (note that you will not get an accurate representation of the color space unless you supply a conversion matrix)

usage:

```
rgb_Nx3 = lms2rgb(dkl_Nx3(el,az,radius), conversionMatrix)
```

psychopy.tools.colorspacetools.rgb2lms

psychopy.tools.colorspacetools.**rgb2lms**(*rgb_Nx3*, *conversionMatrix=None*)

Convert from RGB to cone space (LMS).

Requires a conversion matrix, which will be generated from generic Sony Trinitron phosphors if not supplied (note that you will not get an accurate representation of the color space unless you supply a conversion matrix)

usage:

```
lms_Nx3 = rgb2lms(rgb_Nx3(el,az,radius), conversionMatrix)
```

Gamma/Transfer Functions

Standard gamma functions for converting between *linear RGB* space and *sRGB*.

srgbTF(rgb[, reverse])	Apply sRGB transfer function (or gamma) to linear RGB values.
rec709TF(rgb, **kwargs)	Apply the Rec 709 transfer function (or gamma) to linear RGB values.

psychopy.tools.colorspacetools.srgbTF

psychopy.tools.colorspacetools.**srgbTF**(*rgb*, *reverse=False*, ***kwargs*)

Apply sRGB transfer function (or gamma) to linear RGB values.

Input values must have been transformed using a conversion matrix derived from sRGB primaries relative to D65.

Parameters

- **rgb** (*tuple, list or ndarray of floats*) – Nx3 or NxNx3 array of linear RGB values, last dim must be size == 3 specifying RBG values.
- **reverse** (*boolean*) – If True, the reverse transfer function will convert sRGB -> linear RGB.

Returns

Array of transformed colors with same shape as input.

Return type

ndarray

psychopy.tools.colorspacetools.rec709TF

psychopy.tools.colorspacetools.**rec709TF**(*rgb*, ***kwargs*)

Apply the Rec 709 transfer function (or gamma) to linear RGB values.

This transfer function is defined in the ITU-R BT.709 (2015) recommendation document (https://www.itu.int/rec/R-REC-BT.709-6-201506-I/en) and is commonly used with HDTV televisions.

Parameters

rgb (*tuple, list or ndarray of floats*) – Nx3 or NxNx3 array of linear RGB values, last dim must be size == 3 specifying RBG values.

Returns

Array of transformed colors with same shape as input.

Return type

ndarray

Helpers

Helper functions for working with color coordinates.

rescaleColor(rgb[, convertTo, clip])	Rescale RGB colors.

psychopy.tools.colorspacetools.rescaleColor

psychopy.tools.colorspacetools.**rescaleColor**(*rgb*, *convertTo='signed'*, *clip=False*)

Rescale RGB colors.

This function can be used to convert RGB value triplets from the PsychoPy signed color format to the normalized OpenGL format.

PsychoPy represents colors using values between -1 and 1. However, colors are commonly represented using values between 0 and 1 when working with OpenGL and various other contexts. This function simply rescales values to switch between these formats.

Parameters

- **rgb** (*array_like*) – 1-, 2-, 3-D vector of RGB coordinates to convert. The last dimension should be length-3 in all cases, specifying a single coordinate.
- **convertTo** (*str*) – If 'signed', this function will assume *rgb* is in OpenGL format [0:1] and rescale them to PsychoPy's format [-1:1]. If 'unsigned', input values are treated as OpenGL format and will be rescaled to use PsychoPy's. Default is 'signed'.
- **clip** (*bool*) – Clip values to the range that can be represented on a display. This is an optional step. Default is *False*.

Returns

Rescaled values with the same shape as *rgb*.

Return type

ndarray

Notes

The *convertTo* argument also accepts strings 'opengl' and 'psychopy' as substitutes for 'signed' and 'unsigned', respectively. This might be more explicit in some contexts.

Examples

Convert a signed RGB value to unsigned format:

```
rgb_signed = [-1, 0, 1]
rgb_unsigned = rescaleColor(rgb_signed, convertTo='unsigned')
```

11.8.2 psychopy.tools.coordinatetools

Functions and classes related to coordinate system conversion

cart2pol(x, y[, units])	Convert from cartesian to polar coordinates.
cart2sph(z, y, x)	Convert from cartesian coordinates (x,y,z) to spherical (elevation, azimuth, radius).
pol2cart(theta, radius[, units])	Convert from polar to cartesian coordinates.
sph2cart(*args)	Convert from spherical coordinates (elevation, azimuth, radius) to cartesian (x,y,z).

Function details

psychopy.tools.coordinatetools.**cart2pol**(*x*, *y*, *units='deg'*)

Convert from cartesian to polar coordinates.

Usage

theta, radius = cart2pol(x, y, units='deg') units refers to the units (rad or deg) for theta that should be returned

psychopy.tools.coordinatetools.**cart2sph**(*z*, *y*, *x*)

Convert from cartesian coordinates (x,y,z) to spherical (elevation, azimuth, radius). Output is in degrees.

usage:

array3xN[el,az,rad] = cart2sph(array3xN[x,y,z]) OR elevation, azimuth, radius = cart2sph(x,y,z)

If working in DKL space, z = Luminance, y = S and x = LM

psychopy.tools.coordinatetools.**pol2cart**(*theta*, *radius*, *units='deg'*)

Convert from polar to cartesian coordinates.

usage:

```
x,y = pol2cart(theta, radius, units='deg')
```

psychopy.tools.coordinatetools.**sph2cart**(**args*)

Convert from spherical coordinates (elevation, azimuth, radius) to cartesian (x,y,z).

usage:

array3xN[x,y,z] = sph2cart(array3xN[el,az,rad]) OR x,y,z = sph2cart(elev, azim, radius)

11.8.3 psychopy.tools.filetools

Functions and classes related to file and directory handling

psychopy.tools.filetools.**toFile**(*filename*, *data*)

Save data (of any sort) as a pickle file.

simple wrapper of the cPickle module in core python

psychopy.tools.filetools.**fromFile**(*filename*, *encoding='utf-8-sig'*)

Load data from a psydat, pickle or JSON file.

Parameters

encoding (*str*) – The encoding to use when reading a JSON file. This parameter will be ignored for any other file type.

psychopy.tools.filetools.**mergeFolder**(*src*, *dst*, *pattern=None*)

Merge a folder into another.

Existing files in *dst* folder with the same name will be overwritten. Non-existent files/folders will be created.

psychopy.tools.filetools.**openOutputFile**(*fileName=None*, *append=False*, *fileCollisionMethod='rename'*, *encoding='utf-8-sig'*)

Open an output file (or standard output) for writing.

Parameters

fileName

[None, 'stdout', or str] The desired output file name. If *None* or *stdout*, return *sys.stdout*. Any other string will be considered a filename.

append

[bool, optional] If `True`, append data to an existing file; otherwise, overwrite it with new data. Defaults to `True`, i.e. appending.

fileCollisionMethod

[string, optional] How to handle filename collisions. Valid values are *'rename'*, *'overwrite'*, and *'fail'*. This parameter is ignored if `append` is set to `True`. Defaults to *rename*.

encoding
[string, optional] The encoding to use when writing the file. This parameter will be ignored if *append* is *False* and *fileName* ends with *.psydat* or *.npy* (i.e. if a binary file is to be written). Defaults to `'utf-8'`.

Returns

f
[file] A writable file handle.

`psychopy.tools.filetools.`**`genDelimiter`**`(`*fileName*`)`

Return a delimiter based on a filename.

Parameters

fileName
[string] The output file name.

Returns

delim
[string] A delimiter picked based on the supplied filename. This will be `,` if the filename extension is `.csv`, and a tabulator character otherwise.

11.8.4 `psychopy.tools.gltools`

OpenGL related helper functions.

Shaders

Tools for creating, compiling, using, and inspecting shader programs.

`psychopy.tools.gltools.`**`createProgram`**`()`

Create an empty program object for shaders.

Returns
OpenGL program object handle retrieved from a *glCreateProgram* call.

Return type
int

Examples

Building a program with vertex and fragment shader attachments:

```
myProgram = createProgram()  # new shader object

# compile vertex and fragment shader sources
vertexShader = compileShader(vertShaderSource, GL.GL_VERTEX_SHADER)
fragmentShader = compileShader(fragShaderSource, GL.GL_FRAGMENT_SHADER)

# attach shaders to program
attachShader(myProgram, vertexShader)
attachShader(myProgram, fragmentShader)

# link the shader, makes `myProgram` attachments executable by their
# respective processors and available for use
```

(continues on next page)

(continued from previous page)

```
linkProgram(myProgram)

# optional, validate the program
validateProgram(myProgram)

# optional, detach and discard shader objects
detachShader(myProgram, vertexShader)
detachShader(myProgram, fragmentShader)

deleteObject(vertexShader)
deleteObject(fragmentShader)
```

You can install the program for use in the current rendering state by calling:

```
useProgram(myShader) # OR glUseProgram(myShader)
# set uniforms/attributes and start drawing here ...
```

psychopy.tools.gltools.**createProgramObjectARB**()

Create an empty program object for shaders.

This creates an *Architecture Review Board* (ARB) program variant which is compatible with older GLSL versions and OpenGL coding practices (eg. immediate mode) on some platforms. Use *ARB variants of shader helper functions (eg. *compileShaderObjectARB* instead of *compileShader*) when working with these ARB program objects. This was included for legacy support of existing PsychoPy shaders. However, it is recommended that you use `createShader()` and follow more recent OpenGL design patterns for new code (if possible of course).

Returns

OpenGL program object handle retrieved from a *glCreateProgramObjectARB* call.

Return type

int

Examples

Building a program with vertex and fragment shader attachments:

```
myProgram = createProgramObjectARB()  # new shader object

# compile vertex and fragment shader sources
vertexShader = compileShaderObjectARB(
    vertShaderSource, GL.GL_VERTEX_SHADER_ARB)
fragmentShader = compileShaderObjectARB(
    fragShaderSource, GL.GL_FRAGMENT_SHADER_ARB)

# attach shaders to program
attachObjectARB(myProgram, vertexShader)
attachObjectARB(myProgram, fragmentShader)

# link the shader, makes `myProgram` attachments executable by their
# respective processors and available for use
linkProgramObjectARB(myProgram)

# optional, validate the program
validateProgramARB(myProgram)
```

(continues on next page)

(continued from previous page)

```
# optional, detach and discard shader objects
detachObjectARB(myProgram, vertexShader)
detachObjectARB(myProgram, fragmentShader)

deleteObjectARB(vertexShader)
deleteObjectARB(fragmentShader)
```

Use the program in the current OpenGL state:

```
useProgramObjectARB(myProgram)
```

psychopy.tools.gltools.**compileShader**(*shaderSrc*, *shaderType*)

Compile shader GLSL code and return a shader object. Shader objects can then be attached to programs an made executable on their respective processors.

Parameters

- **shaderSrc** (*str, list of str*) – GLSL shader source code.
- **shaderType** (*GLenum*) – Shader program type (eg. *GL_VERTEX_SHADER*, *GL_FRAGMENT_SHADER*, *GL_GEOMETRY_SHADER*, etc.)

Returns

OpenGL shader object handle retrieved from a *glCreateShader* call.

Return type

int

Examples

Compiling GLSL source code and attaching it to a program object:

```
# GLSL vertex shader source
vertexSource =              '''
    #version 330 core
    layout (location = 0) in vec3 vertexPos;

    void main()
    {
        gl_Position = vec4(vertexPos, 1.0);
    }
    '''
# compile it, specifying `GL_VERTEX_SHADER`
vertexShader = compileShader(vertexSource, GL.GL_VERTEX_SHADER)
attachShader(myProgram, vertexShader)  # attach it to `myProgram`
```

psychopy.tools.gltools.**compileShaderObjectARB**(*shaderSrc*, *shaderType*)

Compile shader GLSL code and return a shader object. Shader objects can then be attached to programs an made executable on their respective processors.

Parameters

- **shaderSrc** (*str, list of str*) – GLSL shader source code text.

- **shaderType** (*GLenum*) – Shader program type. Must be **_ARB* enums such as *GL_VERTEX_SHADER_ARB*, *GL_FRAGMENT_SHADER_ARB*, *GL_GEOMETRY_SHADER_ARB*, etc.

Returns

OpenGL shader object handle retrieved from a *glCreateShaderObjectARB* call.

Return type

int

`psychopy.tools.gltools.`**`embedShaderSourceDefs`**`(`*shaderSrc*, *defs*`)`

Embed preprocessor definitions into GLSL source code.

This function generates and inserts `#define` statements into existing GLSL source code, allowing one to use GLSL preprocessor statements to alter program source at compile time.

Passing `{'MAX_LIGHTS': 8, 'NORMAL_MAP': False}` to *defs* will create and insert the following `#define` statements into *shaderSrc*:

```
#define MAX_LIGHTS 8
#define NORMAL_MAP 0
```

As per the GLSL specification, the `#version` directive must be specified at the top of the file before any other statement (with the exception of comments). If a `#version` directive is present, generated `#define` statements will be inserted starting at the following line. If no `#version` directive is found in *shaderSrc*, the statements will be prepended to *shaderSrc*.

Using preprocessor directives, multiple shader program routines can reside in the same source text if enclosed by `#ifdef` and `#endif` statements as shown here:

```
#ifdef VERTEX
    // vertex shader code here ...
#endif

#ifdef FRAGMENT
    // pixel shader code here ...
#endif
```

Both the vertex and fragment shader can be built from the same GLSL code listing by setting either `VERTEX` or `FRAGMENT` as *True*:

```
vertexShader = gltools.compileShaderObjectARB(
    gltools.embedShaderSourceDefs(glslSource, {'VERTEX': True}),
    GL.GL_VERTEX_SHADER_ARB)
fragmentShader = gltools.compileShaderObjectARB(
    gltools.embedShaderSourceDefs(glslSource, {'FRAGMENT': True}),
    GL.GL_FRAGMENT_SHADER_ARB)
```

In addition, `#ifdef` blocks can be used to prune render code paths. Here, this GLSL snippet shows a shader having diffuse color sampled from a texture is conditional on `DIFFUSE_TEXTURE` being *True*, if not, the material color is used instead:

```
#ifdef DIFFUSE_TEXTURE
    uniform sampler2D diffuseTexture;
#endif
...
#ifdef DIFFUSE_TEXTURE
```

(continues on next page)

(continued from previous page)

```
    // sample color from texture
    vec4 diffuseColor = texture2D(diffuseTexture, gl_TexCoord[0].st);
#else
    // code path for no textures, just output material color
    vec4 diffuseColor = gl_FrontMaterial.diffuse;
#endif
```

This avoids needing to provide two separate GLSL program sources to build shaders to handle cases where a diffuse texture is or isn't used.

Parameters

- **shaderSrc** (*str*) – GLSL shader source code.
- **defs** (*dict*) – Names and values to generate `#define` statements. Keys must all be valid GLSL preprocessor variable names of type *str*. Values can only be *int*, *float*, *str*, *bytes*, or *bool* types. Boolean values *True* and *False* are converted to integers *1* and *0*, respectively.

Returns

GLSL source code with `#define` statements inserted.

Return type

str

Examples

Defining `MAX_LIGHTS` as *8* in a fragment shader program at runtime:

```
fragSrc = embedShaderSourceDefs(fragSrc, {'MAX_LIGHTS': 8})
fragShader = compileShaderObjectARB(fragSrc, GL_FRAGMENT_SHADER_ARB)
```

`psychopy.tools.gltools.`**`deleteObject`**`(`*obj*`)`

Delete a shader or program object.

Parameters

obj (*int*) – Shader or program object handle. Must have originated from a *`createProgram()`*, *`compileShader()`*, *glCreateProgram* or *glCreateShader* call.

`psychopy.tools.gltools.`**`deleteObjectARB`**`(`*obj*`)`

Delete a program or shader object.

Parameters

obj (*int*) – Program handle to attach *shader* to. Must have originated from a *`createProgramObjectARB()`*, `compileShaderObjectARB`, `` `glCreateProgramObjectARB()`` or *glCreateShaderObjectARB* call.

`psychopy.tools.gltools.`**`attachShader`**`(`*program*, *shader*`)`

Attach a shader to a program.

Parameters

- **program** (*int*) – Program handle to attach *shader* to. Must have originated from a *`createProgram()`* or *glCreateProgram* call.
- **shader** (*int*) – Handle of shader object to attach. Must have originated from a *`compileShader()`* or *glCreateShader* call.

psychopy.tools.gltools.**attachObjectARB**(*program*, *shader*)

Attach a shader object to a program.

Parameters

- **program** (*int*) – Program handle to attach *shader* to. Must have originated from a `createProgramObjectARB()` or *glCreateProgramObjectARB* call.
- **shader** (*int*) – Handle of shader object to attach. Must have originated from a `compileShaderObjectARB()` or *glCreateShaderObjectARB* call.

psychopy.tools.gltools.**detachShader**(*program*, *shader*)

Detach a shader object from a program.

Parameters

- **program** (*int*) – Program handle to detach *shader* from. Must have originated from a `createProgram()` or *glCreateProgram* call.
- **shader** (*int*) – Handle of shader object to detach. Must have been previously attached to *program*.

psychopy.tools.gltools.**detachObjectARB**(*program*, *shader*)

Detach a shader object from a program.

Parameters

- **program** (*int*) – Program handle to detach *shader* from. Must have originated from a `createProgramObjectARB()` or *glCreateProgramObjectARB* call.
- **shader** (*int*) – Handle of shader object to detach. Must have been previously attached to *program*.

psychopy.tools.gltools.**linkProgram**(*program*)

Link a shader program. Any attached shader objects will be made executable to run on associated GPU processor units when the program is used.

Parameters

program (*int*) – Program handle to link. Must have originated from a `createProgram()` or *glCreateProgram* call.

Raises

- `ValueError` – Specified *program* handle is invalid.
- `RuntimeError` – Program failed to link. Log will be dumped to *sterr*.

psychopy.tools.gltools.**linkProgramObjectARB**(*program*)

Link a shader program object. Any attached shader objects will be made executable to run on associated GPU processor units when the program is used.

Parameters

program (*int*) – Program handle to link. Must have originated from a `createProgramObjectARB()` or *glCreateProgramObjectARB* call.

Raises

- `ValueError` – Specified *program* handle is invalid.
- `RuntimeError` – Program failed to link. Log will be dumped to *sterr*.

psychopy.tools.gltools.**validateProgram**(*program*)

Check if the program can execute given the current OpenGL state.

Parameters

program (*int*) – Handle of program to validate. Must have originated from a *createProgram()* or *glCreateProgram* call.

psychopy.tools.gltools.**validateProgramARB**(*program*)

Check if the program can execute given the current OpenGL state. If validation fails, information from the driver is dumped giving the reason.

Parameters

program (*int*) – Handle of program object to validate. Must have originated from a *createProgramObjectARB()* or *glCreateProgramObjectARB* call.

psychopy.tools.gltools.**useProgram**(*program*)

Use a program object's executable shader attachments in the current OpenGL rendering state.

In order to install the program object in the current rendering state, a program must have been successfully linked by calling *linkProgram()* or *glLinkProgram*.

Parameters

program (*int*) – Handle of program to use. Must have originated from a *createProgram()* or *glCreateProgram* call and was successfully linked. Passing *0* or *None* disables shader programs.

Examples

Install a program for use in the current rendering state:

```
useProgram(myShader)
```

Disable the current shader program by specifying *0*:

```
useProgram(0)
```

psychopy.tools.gltools.**useProgramObjectARB**(*program*)

Use a program object's executable shader attachments in the current OpenGL rendering state.

In order to install the program object in the current rendering state, a program must have been successfully linked by calling *linkProgramObjectARB()* or *glLinkProgramObjectARB*.

Parameters

program (*int*) – Handle of program object to use. Must have originated from a *createProgramObjectARB()* or *glCreateProgramObjectARB* call and was successfully linked. Passing *0* or *None* disables shader programs.

Examples

Install a program for use in the current rendering state:

```
useProgramObjectARB(myShader)
```

Disable the current shader program by specifying *0*:

```
useProgramObjectARB(0)
```

Notes

Some drivers may support using *glUseProgram* for objects created by calling `createProgramObjectARB()` or *glCreateProgramObjectARB*.

`psychopy.tools.gltools.`**`getInfoLog`**`(`*obj*`)`

Get the information log from a shader or program.

This retrieves a text log from the driver pertaining to the shader or program. For instance, a log can report shader compiler output or validation results. The verbosity and formatting of the logs are platform-dependent, where one driver may provide more information than another.

This function works with both standard and ARB program object variants.

Parameters

obj (*int*) – Program or shader to retrieve a log from. If a shader, the handle must have originated from a `compileShader()`, *glCreateShader*, `createProgramObjectARB()` or *glCreateProgramObjectARB* call. If a program, the handle must have came from a `createProgram()`, `createProgramObjectARB()`, *glCreateProgram* or *glCreateProgramObjectARB* call.

Returns

Information log data. Logs can be empty strings if the driver has no information available.

Return type

str

`psychopy.tools.gltools.`**`getUniformLocations`**`(`*program*, *builtins=False*`)`

Get uniform names and locations from a given shader program object.

This function works with both standard and ARB program object variants.

Parameters

- **program** (*int*) – Handle of program to retrieve uniforms. Must have originated from a `createProgram()`, `createProgramObjectARB()`, *glCreateProgram* or *glCreateProgramObjectARB* call.
- **builtins** (*bool, optional*) – Include built-in GLSL uniforms (eg. *gl_ModelViewProjectionMatrix*). Default is *False*.

Returns

Uniform names and locations.

Return type

dict

`psychopy.tools.gltools.`**`getAttribLocations`**`(`*program*, *builtins=False*`)`

Get attribute names and locations from the specified program object.

This function works with both standard and ARB program object variants.

Parameters

- **program** (*int*) – Handle of program to retrieve attributes. Must have originated from a `createProgram()`, `createProgramObjectARB()`, *glCreateProgram* or *glCreateProgramObjectARB* call.
- **builtins** (*bool, optional*) – Include built-in GLSL attributes (eg. *gl_Vertex*). Default is *False*.

Returns

Attribute names and locations.

Return type
dict

Query

Tools for using OpenGL query objects.

psychopy.tools.gltools.**createQueryObject**(*target=35007*)

Create a GL query object.

Parameters
target (*Glenum or int*) – Target for the query.

Returns
Query object.

Return type
QueryObjectInfo

Examples

Get GPU time elapsed executing rendering/GL calls associated with some stimuli (this is not the difference in absolute time between consecutive *beginQuery* and *endQuery* calls!):

```
# create a new query object
qGPU = createQueryObject(GL_TIME_ELAPSED)

beginQuery(query)
myStim.draw()  # OpenGL calls here
endQuery(query)

# get time elapsed in seconds spent on the GPU
timeRendering = getQueryValue(qGPU) * 1e-9
```

You can also use queries to test if vertices are occluded, as their samples would be rejected during depth testing:

```
drawVAO(shape0, GL_TRIANGLES)  # draw the first object

# check if the object was completely occluded
qOcclusion = createQueryObject(GL_ANY_SAMPLES_PASSED)

# draw the next shape within query context
beginQuery(qOcclusion)
drawVAO(shape1, GL_TRIANGLES)  # draw the second object
endQuery(qOcclusion)

isOccluded = getQueryValue(qOcclusion) == 1
```

This can be leveraged to perform occlusion testing/culling, where you can render a *cheap* version of your mesh/shape, then the more expensive version if samples were passed.

psychopy.tools.gltools.**QueryObjectInfo**(*name*, *target*)

Object for querying information. This includes GPU timing information.

psychopy.tools.gltools.**beginQuery**(*query*)

Begin query.

Parameters

query (*QueryObjectInfo*) – Query object descriptor returned by *createQueryObject()*.

psychopy.tools.gltools.**endQuery**(*query*)

End a query.

Parameters

query (*QueryObjectInfo*) – Query object descriptor returned by *createQueryObject()*, previously passed to *beginQuery()*.

psychopy.tools.gltools.**getQuery**(*query*)

Get the value stored in a query object.

Parameters

query (*QueryObjectInfo*) – Query object descriptor returned by *createQueryObject()*, previously passed to *endQuery()*.

psychopy.tools.gltools.**getAbsTimeGPU**()

Get the absolute GPU time in nanoseconds.

Returns

Time elapsed in nanoseconds since the OpenGL context was fully realized.

Return type

int

Examples

Get the current GPU time in seconds:

```
timeInSeconds = getAbsTimeGPU() * 1e-9
```

Get the GPU time elapsed:

```
t0 = getAbsTimeGPU()
# some drawing commands here ...
t1 = getAbsTimeGPU()
timeElapsed = (t1 - t0) * 1e-9  # take difference, convert to seconds
```

Framebuffer Objects (FBO)

Tools for creating Framebuffer Objects (FBOs).

psychopy.tools.gltools.**createFBO**(*attachments=()*, *sizeHint=None*)

Create a Framebuffer Object.

Parameters

- **attachments** (list or tuple of tuple) – Optional attachments to initialize the Framebuffer with. Attachments are specified as a list of tuples. Each tuple must contain an attachment point (e.g. GL_COLOR_ATTACHMENT0, GL_DEPTH_ATTACHMENT, etc.) and a buffer descriptor type (RenderbufferInfo or TexImage2DInfo). If using a combined depth/stencil format such as GL_DEPTH24_STENCIL8, GL_DEPTH_ATTACHMENT and GL_STENCIL_ATTACHMENT must be passed the same buffer. Alternatively, one can use GL_DEPTH_STENCIL_ATTACHMENT instead. If using multisample buffers, all attachment images must use the same number of samples!. As an example, one may specify attachments as 'attachments=((GL.GL_COLOR_ATTACHMENT0, frameTexture), (GL.GL_DEPTH_STENCIL_ATTACHMENT, depthRenderBuffer))'.

- **sizeHint** (tuple, optional) – Size hint for the FBO. This is used to specify the dimensions of logical buffers attached to the FBO. The size hint is a tuple of two integers (width, height).

Returns
Framebuffer descriptor.

Return type
FramebufferInfo

Notes

- All buffers must have the same number of samples.
- The 'userData' field of the returned descriptor is a dictionary that can be used to store arbitrary data associated with the FBO.
- Framebuffers need a single attachment to be complete.

Examples

Create an empty framebuffer with no attachments:

```
fbo = createFBO()  # invalid until attachments are added
```

Create a render target with multiple color texture attachments:

```
fbo = gt.createFBO(sizeHint=(512, 512))
gt.bindFBO(fbo)
gt.attachImage(  # color
    fbo, GL.GL_COLOR_ATTACHMENT0, gt.createTexImage2D(512, 512))
gt.attachImage(  # normal map
    fbo, GL.GL_COLOR_ATTACHMENT1, gt.createTexImage2D(512, 512))
gt.attachImage(  # depth/stencil
    fbo,
    GL.GL_DEPTH_STENCIL_ATTACHMENT,
    gt.createRenderbuffer(512, 512, GL.GL_DEPTH24_STENCIL8))
print(gt.isFramebufferComplete(fbo))  # True
gt.unbindFBO(None)
```

Examples of userData some custom function might access:

```
fbo.userData['flags'] = ['left_eye', 'clear_before_use']
```

Using a depth only texture (for shadow mapping?):

```
depthTex = createTexImage2D(800, 600,
                            internalFormat=GL.GL_DEPTH_COMPONENT24,
                            pixelFormat=GL.GL_DEPTH_COMPONENT)
fbo = createFBO([(GL.GL_DEPTH_ATTACHMENT, depthTex)])  # is valid

# discard FBO descriptor, just give me the ID
frameBuffer = createFBO().id
```

psychopy.tools.gltools.**attach**(*fbo*, *attachPoint*, *imageBuffer*)

Attach an image to a specified attachment point on the presently bound FBO.

Parameters
fbo (FramebufferInfo) – Framebuffer descriptor to attach buffer to.

:param attachPoint `int`: Attachment point for 'imageBuffer' (e.g. GL.GL_COLOR_ATTACHMENT0). :param imageBuffer: Framebuffer-attachable buffer descriptor. :type imageBuffer: `TexImage2D` or `RenderbufferInfo`

Examples

Attach an image to attachment points on the framebuffer:

```
GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, fbo)
attachImage(GL.GL_COLOR_ATTACHMENT0, colorTex)
attachImage(GL.GL_DEPTH_STENCIL_ATTACHMENT, depthRb)
GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, lastBoundFbo)

# same as above, but using a context manager
with useFBO(fbo):
    attachImage(GL.GL_COLOR_ATTACHMENT0, colorTex)
    attachImage(GL.GL_DEPTH_STENCIL_ATTACHMENT, depthRb)
```

`psychopy.tools.gltools.`**`deleteFBO`**(*fbo*, *deleteAttachments=True*)

Delete a framebuffer.

Parameters

- **fbo** (`FramebufferInfo` or `int`) – Framebuffer descriptor or name to delete.
- **deleteAttachments** (*`bool`, `optional`*) – Delete attachments associated with the framebuffer. Default is *True*.

`psychopy.tools.gltools.`**`blitFBO`**(*srcRect*, *dstRect=None*, *filter=9729*, *mask=16384*)

Copy a block of pixels between framebuffers via blitting. Read and draw framebuffers must be bound prior to calling this function. Beware, the scissor box and viewport are changed when this is called to dstRect.

Parameters

- **srcRect** (`list` of `int`) – List specifying the top-left and bottom-right coordinates of the region to copy from (<X0>, <Y0>, <X1>, <Y1>).
- **dstRect** (`list` of `int` or `None`) – List specifying the top-left and bottom-right coordinates of the region to copy to (<X0>, <Y0>, <X1>, <Y1>). If None, srcRect is used for dstRect.
- **filter** (`int` or `str`) – Interpolation method to use if the image is stretched, default is GL_LINEAR, but can also be GL_NEAREST.
- **mask** (`int` or `str`) – Bitmask specifying which buffers to copy. Default is GL_COLOR_BUFFER_BIT.

Examples

Blitting pixels from on FBO to another:

```
# set buffers for reading and drawing
gt.setReadBuffer(fbo, 'GL_COLOR_ATTACHMENT0')
gt.setDrawBuffer(None, 'GL_BACK')  # default back buffer for window
gt.blitFBO((0 ,0, 512, 512), (0, 0, 512, 512))
```

`psychopy.tools.gltools.`**`useFBO`**(*fbo*)

Context manager for Framebuffer Object bindings. This function yields the framebuffer name as an integer.

:param fbo `int` or `Framebuffer`: OpenGL Framebuffer Object name/ID or descriptor.

Yields

int – OpenGL name of the framebuffer bound in the context.

Examples

Using a framebuffer context manager:

```
# FBO bound somewhere deep in our code
GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, someOtherFBO)

...

# create a new FBO, but we have no idea what the currently bound FBO is
fbo = createFBO()

# use a context to bind attachments
with bindFBO(fbo):
    attach(GL.GL_COLOR_ATTACHMENT0, colorTex)
    attach(GL.GL_DEPTH_ATTACHMENT, depthRb)
    attach(GL.GL_STENCIL_ATTACHMENT, depthRb)
    isComplete = gltools.isComplete()

# someOtherFBO is still bound!
```

Renderbuffers

Tools for creating Renderbuffers.

`psychopy.tools.gltools.`**`createRenderbuffer`**`(`*width*, *height*, *internalFormat=32856*, *samples=1*`)`

Create a new Renderbuffer Object with a specified internal format. A multisample storage buffer is created if samples > 1.

Renderbuffers contain image data and are optimized for use as render targets. See https://www.khronos.org/opengl/wiki/Renderbuffer_Object for more information.

Parameters

- **`width`** (`int`) – Buffer width in pixels.
- **`height`** (`int`) – Buffer height in pixels.
- **`internalFormat`** (`int`) – Format for renderbuffer data (e.g. GL_RGBA8, GL_DEPTH24_STENCIL8).
- **`samples`** (`int`) – Number of samples for multi-sampling, should be >1 and power-of-two. If samples == 1, a single sample buffer is created.

Returns

A descriptor of the created renderbuffer.

Return type

Renderbuffer

Notes

The ‘userData’ field of the returned descriptor is a dictionary that can be used to store arbitrary data associated with the buffer.

psychopy.tools.gltools.**deleteRenderbuffer**(*renderBuffer*)

Free the resources associated with a renderbuffer. This invalidates the renderbuffer's ID.

Textures

Tools for creating textures.

psychopy.tools.gltools.**createTexImage2D**(*width*, *height*, *target=3553*, *level=0*, *internalFormat=32856*, *pixelFormat=6408*, *dataType=5126*, *data=None*, *unpackAlignment=4*, *texParams=None*)

Create a 2D texture in video memory. This can only create a single 2D texture with targets *GL_TEXTURE_2D* or *GL_TEXTURE_RECTANGLE*.

Parameters

- **width** (int) – Texture width in pixels.
- **height** (int) – Texture height in pixels.
- **target** (int) – The target texture should only be either GL_TEXTURE_2D or GL_TEXTURE_RECTANGLE.
- **level** (int) – LOD number of the texture, should be 0 if GL_TEXTURE_RECTANGLE is the target.
- **internalFormat** (int) – Internal format for texture data (e.g. GL_RGBA8, GL_R11F_G11F_B10F).
- **pixelFormat** (int) – Pixel data format (e.g. GL_RGBA, GL_DEPTH_STENCIL)
- **dataType** (int) – Data type for pixel data (e.g. GL_FLOAT, GL_UNSIGNED_BYTE).
- **data** (ctypes or None) – Ctypes pointer to image data. If None is specified, the texture will be created but pixel data will be uninitialized.
- **unpackAlignment** (int) – Alignment requirements of each row in memory. Default is 4.
- **texParams** (dict) – Optional texture parameters specified as *dict*. These values are passed to *glTexParameteri*. Each tuple must contain a parameter name and value. For example, *texParameters={GL.GL_TEXTURE_MIN_FILTER: GL.GL_LINEAR, GL.GL_TEXTURE_MAG_FILTER: GL.GL_LINEAR}*.

Returns

A *TexImage2DInfo* descriptor.

Return type

TexImage2DInfo

Notes

The 'userData' field of the returned descriptor is a dictionary that can be used to store arbitrary data associated with the texture.

Previous textures are unbound after calling 'createTexImage2D'.

Examples

Creating a texture from an image file:

```
import pyglet.gl as GL  # using Pyglet for now

# empty texture
textureDesc = createTexImage2D(1024, 1024, internalFormat=GL.GL_RGBA8)

# load texture data from an image file using Pillow and NumPy
from PIL import Image
import numpy as np
im = Image.open(imageFile)  # 8bpp!
im = im.transpose(Image.FLIP_TOP_BOTTOM)  # OpenGL origin is at bottom
im = im.convert("RGBA")
pixelData = np.array(im).ctypes  # convert to ctypes!

width = pixelData.shape[1]
height = pixelData.shape[0]
textureDesc = gltools.createTexImage2D(
    width,
    height,
    internalFormat=GL.GL_RGBA,
    pixelFormat=GL.GL_RGBA,
    dataType=GL.GL_UNSIGNED_BYTE,
    data=pixelData,
    unpackAlignment=1,
    texParameters=[(GL.GL_TEXTURE_MAG_FILTER, GL.GL_LINEAR),
                   (GL.GL_TEXTURE_MIN_FILTER, GL.GL_LINEAR)])

GL.glBindTexture(GL.GL_TEXTURE_2D, textureDesc.id)
```

psychopy.tools.gltools.**createTexImage2dFromFile**(*imgFile*, *transpose=True*)

Load an image from file directly into a texture.

This is a convenience function to quickly get an image file loaded into a 2D texture. The image is converted to RGBA format. Texture parameters are set for linear interpolation.

Parameters

- **imgFile** (*str*) – Path to the image file.
- **transpose** (*bool*) – Flip the image so it appears upright when displayed in OpenGL image coordinates.

Returns

Texture descriptor.

Return type

TexImage2DInfo

psychopy.tools.gltools.**createTexImage2DMultisample**(*width*, *height*, *target=37120*, *samples=1*, *internalFormat=32856*, *texParameters=()*)

Create a 2D multisampled texture.

Parameters

- **width** (int) – Texture width in pixels.
- **height** (int) – Texture height in pixels.
- **target** (int) – The target texture (e.g. GL_TEXTURE_2D_MULTISAMPLE).

- **samples** (int) – Number of samples for multi-sampling, should be >1 and power-of-two. Work with one sample, but will raise a warning.
- **internalFormat** (int) – Internal format for texture data (e.g. GL_RGBA8, GL_R11F_G11F_B10F).
- **texParameters** (list of tuple of int) – Optional texture parameters specified as a list of tuples. These values are passed to 'glTexParameteri'. Each tuple must contain a parameter name and value. For example, texParameters=[(GL.GL_TEXTURE_MIN_FILTER, GL.GL_LINEAR), (GL.GL_TEXTURE_MAG_FILTER, GL.GL_LINEAR)]

Returns
A TexImage2DMultisampleInfo descriptor.

Return type
TexImage2DMultisampleInfo

psychopy.tools.gltools.**deleteTexture**(*texture*)

Free the resources associated with a texture. This invalidates the texture's ID.

psychopy.tools.gltools.**bindTexture**(*texture*, *target=None*, *unit=None*)

Bind a texture to a target.

Parameters

- **texture** (*GLuint, int, None, TexImage2DInfo or TexImage2DMultisampleInfo*) – Texture to bind. If *None*, the texture will be unbound.
- **target** (*GLenum, int or str, optional*) – Target to bind the texture to. Default is *None*. If *None*, and the texture is a *TexImage2DInfo* or *TexImage2DMultisampleInfo* object, the target will be inferred from the texture object. If specified, the value will override the target inferred from the texture. If *None* and the texture is an integer, the target will default to *GL_TEXTURE_2D*.
- **unit** (*GLenum, int or None, optional*) – Texture unit to bind the texture to, this will also set the active texture unit. Default is *None*. If *None*, the texture will be bound to the currently active texture unit.

psychopy.tools.gltools.**createCubeMap**(*width*, *height*, *target=34067*, *level=0*, *internalFormat=6408*, *pixelFormat=6408*, *dataType=5121*, *data=None*, *unpackAlignment=4*, *texParams=None*)

Create a cubemap.

Parameters

- **name** (*int* or *GLuint*) – OpenGL handle for the cube map. Is *0* if uninitialized.
- **target** (int) – The target texture should only be *GL_TEXTURE_CUBE_MAP*.
- **width** (int) – Texture width in pixels.
- **height** (int) – Texture height in pixels.
- **level** (int) – LOD number of the texture.
- **internalFormat** (int) – Internal format for texture data (e.g. GL_RGBA8, GL_R11F_G11F_B10F).
- **pixelFormat** (int) – Pixel data format (e.g. GL_RGBA, GL_DEPTH_STENCIL)
- **dataType** (int) – Data type for pixel data (e.g. GL_FLOAT, GL_UNSIGNED_BYTE).

- **data** (*list or tuple*) – List of six ctypes pointers to image data for each cubemap face. Image data is assigned to a face by index [+X, -X, +Y, -Y, +Z, -Z]. All images must have the same size as specified by *width* and *height*.
- **unpackAlignment** (int) – Alignment requirements of each row in memory. Default is 4.
- **texParams** (list of tuple of int) – Optional texture parameters specified as *dict*. These values are passed to *glTexParameteri*. Each tuple must contain a parameter name and value. For example, *texParameters={ GL.GL_TEXTURE_MIN_FILTER: GL.GL_LINEAR, GL.GL_TEXTURE_MAG_FILTER: GL.GL_LINEAR}*. These can be changed and will be updated the next time this instance is passed to `bindTexture()`.

Vertex Buffer/Array Objects

Tools for creating and working with Vertex Buffer Objects (VBOs) and Vertex Array Objects (VAOs).

psychopy.tools.gltools.**VertexArrayInfo**(*name=0*, *count=0*, *activeAttribs=None*, *indexBuffer=None*, *attribDivisors=None*, *isLegacy=False*, *userData=None*)

Vertex array object (VAO) descriptor.

This class only stores information about the VAO it refers to, it does not contain any actual array data associated with the VAO. Calling `createVAO()` returns instances of this class.

If *isLegacy* is *True*, attribute binding states are using deprecated (but still supported) pointer definition calls (eg. *glVertexPointer*). This is to ensure backwards compatibility. The keys stored in *activeAttribs* must be *GLenum* types such as *GL_VERTEX_ARRAY*.

Parameters

- **name** (*int*) – OpenGL handle for the VAO.
- **count** (*int*) – Number of vertex elements. If *indexBuffer* is not *None*, count corresponds to the number of elements in the index buffer.
- **activeAttribs** (*dict*) – Attributes and buffers defined as part of this VAO state. Keys are attribute pointer indices or capabilities (ie. GL_VERTEX_ARRAY). Modifying these values will not update the VAO state.
- **indexBuffer** (*VertexBufferInfo, optional*) – Buffer object for indices.
- **attribDivisors** (*dict, optional*) – Divisors for each attribute.
- **isLegacy** (*bool*) – Array pointers were defined using the deprecated OpenGL API. If *True*, the VAO may work with older GLSL shaders versions and the fixed-function pipeline.
- **userData** (*dict or None, optional*) – Optional user defined data associated with this VAO.

psychopy.tools.gltools.**createVAO**(*attribBuffers*, *indexBuffer=None*, *attribDivisors=None*, *legacy=False*)

Create a Vertex Array object (VAO). VAOs store buffer binding states, reducing CPU overhead when drawing objects with vertex data stored in VBOs.

Define vertex attributes within a VAO state by passing a mapping for generic attribute indices and VBO buffers.

Parameters

- **attribBuffers** (*dict*) – Attributes and associated VBOs to add to the VAO state. Keys are vertex attribute pointer indices, values are VBO descriptors to define. Values can be *tuples* where the first value is the buffer descriptor, the second is the number of attribute components (*int*, either 2, 3 or 4), the third is the offset (*int*), and the last is whether to normalize the array (*bool*).
- **indexBuffer** (*VertexBufferInfo*) – Optional index buffer.

- **attribDivisors** (*dict*) – Attribute divisors to set. Keys are vertex attribute pointer indices, values are the number of instances that will pass between updates of an attribute. Setting attribute divisors is only permitted if *legacy* is *False*.
- **legacy** (*bool, optional*) – Use legacy attribute pointer functions when setting the VAO state. This is for compatibility with older GL implementations. Key specified to *attribBuffers* must be *GLenum* types such as *GL_VERTEX_ARRAY* to indicate the capability to use.

Examples

Create a vertex array object and enable buffer states within it:

```
vao = createVAO({0: vertexPos, 1: texCoords, 2: vertexNormals})
```

Using an interleaved vertex buffer, all attributes are in the same buffer (*vertexAttr*). We need to specify offsets for each attribute by passing a buffer in a *tuple* with the second value specifying the offset:

```
# buffer with interleaved layout `00011222` per-attribute
vao = createVAO(
    {0: (vertexAttr, 3),            # size 3, offset 0
     1: (vertexAttr, 2, 3),         # size 2, offset 3
     2: (vertexAttr, 3, 5, True)})  # size 3, offset 5, normalize
```

You can mix interleaved and single-use buffers:

```
vao = createVAO(
    {0: (vertexAttr, 3, 0), 1: (vertexAttr, 3, 3), 2: vertexColors})
```

Specifying an optional index array, this is used for indexed drawing of primitives:

```
vao = createVAO({0: vertexPos}, indexBuffer=indices)
```

The returned *VertexArrayInfo* instance will have attribute `isIndexed==True`.

Drawing vertex arrays using a VAO, will use the *indexBuffer* if available:

```
# draw the array
drawVAO(vao, mode=GL.GL_TRIANGLES)
```

Use legacy attribute pointer bindings when building a VAO for compatibility with the fixed-function pipeline and older GLSL versions:

```
attribBuffers = {GL_VERTEX_ARRAY: vertexPos, GL_NORMAL_ARRAY: normals}
vao = createVAO(attribBuffers, legacy=True)
```

If you wish to used instanced drawing, you can specify attribute divisors this way:

```
vao = createVAO(
    {0: (vertexAttr, 3, 0), 1: (vertexAttr, 3, 3), 2: vertexColors},
    attribDivisors={2: 1})
```

`psychopy.tools.gltools.`**`drawVAO`**(*vao*, *mode=4*, *start=0*, *count=None*, *instanceCount=None*, *flush=False*)

Draw a vertex array object. Uses *glDrawArrays* or *glDrawElements* if *instanceCount* is *None*, or else *glDrawArraysInstanced* or *glDrawElementsInstanced* is used.

Parameters

- **vao** (*VertexArrayObject*) – Vertex Array Object (VAO) to draw.

- **mode** (*int, optional*) – Drawing mode to use (e.g. GL_TRIANGLES, GL_QUADS, GL_POINTS, etc.) for rasterization. Default is *GL_TRIANGLES*. Strings can be used for convenience (e.g. 'GL_TRIANGLES', 'GL_QUADS', 'GL_POINTS').
- **start** (*int, optional*) – Starting index for array elements. Default is *0* which is the beginning of the array.
- **count** (*int, optional*) – Number of indices to draw from *start*. Must not exceed *vao.count - start*.
- **instanceCount** (*int or None*) – Number of instances to draw. If >0 and not *None*, instanced drawing will be used.
- **flush** (*bool, optional*) – Flush queued drawing commands before returning.

Examples

Creating a VAO and drawing it:

```
# draw the VAO, renders the mesh
drawVAO(vaoDesc, GL.GL_TRIANGLES)
```

psychopy.tools.gltools.**deleteVAO**(*vao*)

Delete a Vertex Array Object (VAO). This does not delete array buffers bound to the VAO.

Parameters

vao (*VertexArrayInfo*) – VAO to delete. All fields in the descriptor except *userData* will be reset.

psychopy.tools.gltools.**VertexBufferInfo**(*name=0*, *target=34962*, *usage=35044*, *dataType=5126*, *size=0*, *stride=0*, *shape=(0,)*, *userData=None*)

Vertex buffer object (VBO) descriptor.

This class only stores information about the VBO it refers to, it does not contain any actual array data associated with the VBO. Calling `createVBO()` returns instances of this class.

It is recommended to use *gltools* functions `bindVBO()`, `unbindVBO()`, `mapBuffer()`, etc. when working with these objects.

Parameters

- **name** (*GLuint or int*) – OpenGL handle for the buffer.
- **target** (*GLenum or int, optional*) – Target used when binding the buffer (e.g. *GL_VERTEX_ARRAY* or *GL_ELEMENT_ARRAY_BUFFER*). Default is *GL_VERTEX_ARRAY*)
- **usage** (*GLenum or int, optional*) – Usage type for the array (i.e. *GL_STATIC_DRAW*).
- **dataType** (*Glenum, optional*) – Data type of array. Default is *GL_FLOAT*.
- **size** (*int, optional*) – Size of the buffer in bytes.
- **stride** (*int, optional*) – Number of bytes between adjacent attributes. If *0*, values are assumed to be tightly packed.
- **shape** (*tuple or list, optional*) – Shape of the array used to create this VBO.
- **userData** (*dict, optional*) – Optional user defined data associated with the VBO. If *None*, *userData* will be initialized as an empty dictionary.

psychopy.tools.gltools.**createVBO**(*data*, *target=34962*, *dataType=None*, *usage=35044*)

Create an array buffer object (VBO).

Creates a VBO using input data, usually as a *ndarray* or *list*. Attributes common to one vertex should occupy a single row of the *data* array.

Parameters

- **data** (*array_like*) – A 2D array of values to write to the array buffer. The data type of the VBO is inferred by the type of the array. If the input is a Python *list* or *tuple* type, the data type of the array will be *GL_DOUBLE*.
- **target** (int or str, optional) – Target used when binding the buffer (e.g. *GL_VERTEX_ARRAY* or *GL_ELEMENT_ARRAY_BUFFER*). Default is *GL_VERTEX_ARRAY*. Strings may also be used to specify the target, where the following are valid: 'array' (for *GL_VERTEX_ARRAY*) or 'element_array' (for *GL_ELEMENT_ARRAY_BUFFER*).
- **dataType** (*Glenum or None, optional*) – Data type of array. Input data will be recast to an appropriate type if necessary. Default is *None*. If *None*, the data type will be inferred from the input data.
- **usage** (*GLenum or int, optional*) – Usage hint for the array (i.e. *GL_STATIC_DRAW*). This will hint to the GL driver how the buffer will be used so it can optimize memory allocation and access. Default is *GL_STATIC_DRAW*.

Returns

A descriptor with vertex buffer information.

Return type

VertexBufferInfo

Examples

Creating a vertex buffer object with vertex data:

```
# vertices of a triangle
verts = [[ 1.0,  1.0, 0.0],   # v0
         [ 0.0, -1.0, 0.0],   # v1
         [-1.0,  1.0, 0.0]]   # v2

# load vertices to graphics device, return a descriptor
vboDesc = createVBO(verts)
```

Drawing triangles or quads using vertex buffer data:

```
nIndices, vSize = vboDesc.shape  # element size

bindVBO(vboDesc)
setVertexAttribPointer(
    GL_VERTEX_ARRAY, vSize, vboDesc.dataType, legacy=True)
enableVertexAttribArray(GL_VERTEX_ARRAY, legacy=True)

if vSize == 3:
    drawMode = GL_TRIANGLES
elif vSize == 4:
    drawMode = GL_QUADS
```

(continues on next page)

(continued from previous page)

```
glDrawArrays(drawMode, 0, nIndices)
glFlush()

disableVertexAttribArray(GL_VERTEX_ARRAY, legacy=True)
unbindVBO()
```

Custom data can be associated with this vertex buffer by specifying *userData*:

```
myVBO = createVBO(data)
myVBO.userData['startIdx'] = 14  # first index to draw with

# use it later
nIndices, vSize = vboDesc.shape  # element size
startIdx = myVBO.userData['startIdx']
endIdx = nIndices - startIdx
glDrawArrays(GL_TRIANGLES, startIdx, endIdx)
glFlush()
```

`psychopy.tools.gltools.`**`bindVBO`**`(`*vbo*`)`

Bind a VBO to the current GL state.

Parameters

vbo (*VertexBufferInfo*) – VBO descriptor to bind.

Returns

True is the binding state was changed. Returns *False* if the state was not changed due to the buffer already being bound.

Return type

bool

`psychopy.tools.gltools.`**`unbindVBO`**`(`*vbo*`)`

Unbind a vertex buffer object (VBO).

Parameters

vbo (*VertexBufferInfo*) – VBO descriptor to unbind.

`psychopy.tools.gltools.`**`mapBuffer`**`(`*vbo*, *start=0*, *length=None*, *read=True*, *write=True*, *noSync=False*`)`

Map a vertex buffer object to client memory. This allows you to modify its contents.

If planning to update VBO vertex data, make sure the VBO *usage* types are *GL_DYNAMIC_** or *GL_STREAM_** or else serious performance issues may arise.

> **Warning**
>
> Modifying buffer data must be done carefully, or else system stability may be affected. Do not use the returned view *ndarray* outside of successive *mapBuffer()* and *unmapBuffer()* calls. Do not use the mapped buffer for rendering until after *unmapBuffer()* is called.

Parameters

- **vbo** (*VertexBufferInfo*) – Vertex buffer to map to client memory.
- **start** (*int*) – Initial index of the sub-range of the buffer to modify.

- **length** (*int or None*) – Number of elements of the sub-array to map from *offset*. If *None*, all elements to from *offset* to the end of the array are mapped.
- **read** (*bool, optional*) – Allow data to be read from the buffer (sets *GL_MAP_READ_BIT*). This is ignored if *noSync* is *True*.
- **write** (*bool, optional*) – Allow data to be written to the buffer (sets *GL_MAP_WRITE_BIT*).
- **noSync** (*bool, optional*) – If *True*, GL will not wait until the buffer is free (i.e. not being processed by the GPU) to map it (sets *GL_MAP_UNSYNCHRONIZED_BIT*). The contents of the previous storage buffer are discarded and the driver returns a new one. This prevents the CPU from stalling until the buffer is available.

Returns

View of the data. The type of the returned array is one which best matches the data type of the buffer.

Return type

ndarray

Examples

Map a buffer and edit it:

```
arr = mapBuffer(vbo)
arr[:, :] += 2.0  # add 2 to all values
unmapBuffer(vbo)  # call when done
# Don't ever modify `arr` after calling `unmapBuffer`. Delete it if
# necessary to prevent it form being used.
del arr
```

Modify a sub-range of data by specifying *start* and *length*, indices correspond to values, not byte offsets:

```
arr = mapBuffer(vbo, start=12, end=24)
arr[:, :] *= 10.0
unmapBuffer(vbo)
```

psychopy.tools.gltools.**unmapBuffer**(*vbo*)

Unmap a previously mapped buffer. Must be called after *mapBuffer()* is called and before any drawing operations which use the buffer are called. Failing to call this before using the buffer could result in a system error.

Parameters

vbo (*VertexBufferInfo*) – Vertex buffer descriptor.

Returns

True if the buffer has been successfully modified. If *False*, the data was corrupted for some reason and needs to be resubmitted.

Return type

bool

psychopy.tools.gltools.**deleteVBO**(*vbo*)

Delete a vertex buffer object (VBO).

Parameters

vbo (*VertexBufferInfo*) – Descriptor of VBO to delete.

psychopy.tools.gltools.**setVertexAttribPointer**(*index*, *vbo*, *size=None*, *offset=0*, *normalize=False*, *legacy=False*)

Define an array of vertex attribute data with a VBO descriptor.

In modern OpenGL implementations, attributes are 'generic', where an attribute pointer index does not correspond to any special vertex property. Usually the usage for an attribute is defined in the shader program. It is recommended that shader programs define attributes using the *layout* parameters:

```
layout (location = 0) in vec3 position;
layout (location = 1) in vec2 texCoord;
layout (location = 2) in vec3 normal;
```

Setting attribute pointers can be done like this:

```
setVertexAttribPointer(0, posVbo)
setVertexAttribPointer(1, texVbo)
setVertexAttribPointer(2, normVbo)
```

For compatibility with older OpenGL specifications, some drivers will alias vertex pointers unless they are explicitly defined in the shader. This allows VAOs the be used with the fixed-function pipeline or older GLSL versions.

On nVidia graphics drivers (and maybe others), the following attribute pointer indices are aliased with reserved GLSL names:

- gl_Vertex - 0
- gl_Normal - 2
- gl_Color - 3
- gl_SecondaryColor - 4
- gl_FogCoord - 5
- gl_MultiTexCoord0 - 8
- gl_MultiTexCoord1 - 9
- gl_MultiTexCoord2 - 10
- gl_MultiTexCoord3 - 11
- gl_MultiTexCoord4 - 12
- gl_MultiTexCoord5 - 13
- gl_MultiTexCoord6 - 14
- gl_MultiTexCoord7 - 15

Specifying *legacy* as *True* will allow for old-style pointer definitions. You must specify the capability as a *GLenum* associated with the pointer in this case:

```
setVertexAttribPointer(GL_VERTEX_ARRAY, posVbo, legacy=True)
setVertexAttribPointer(GL_TEXTURE_COORD_ARRAY, texVbo, legacy=True)
setVertexAttribPointer(GL_NORMAL_ARRAY, normVbo, legacy=True)
```

Parameters

- **index** (*int*) – Index of the attribute to modify. If *legacy=True*, this value should be a *GLenum* type corresponding to the capability to bind the buffer to, such as *GL_VERTEX_ARRAY*, *GL_TEXTURE_COORD_ARRAY*, *GL_NORMAL_ARRAY*, etc.
- **vbo** (*VertexBufferInfo*) – VBO descriptor.
- **size** (*int, optional*) – Number of components per vertex attribute, can be either 1, 2, 3, or 4. If *None* is specified, the component size will be inferred from the *shape* of the VBO. You must specify this value if the VBO is interleaved.
- **offset** (*int, optional*) – Starting index of the attribute in the buffer.
- **normalize** (*bool, optional*) – Normalize fixed-point format values when accessed.
- **legacy** (*bool, optional*) – Use legacy vertex attributes (ie. *GL_VERTEX_ARRAY*, *GL_TEXTURE_COORD_ARRAY*, etc.) for backwards compatibility.

Examples

Define a generic attribute from a vertex buffer descriptor:

```
# set the vertex location attribute
setVertexAttribPointer(0, vboDesc)  # 0 is vertex in our shader
GL.glColor3f(1.0, 0.0, 0.0)  # red triangle

# draw the triangle
nIndices, vSize = vboDesc.shape  # element size
GL.glDrawArrays(GL.GL_TRIANGLES, 0, nIndices)
```

If our VBO has interleaved attributes, we can specify *offset* to account for that:

```
# define interleaved vertex attributes
#        |     Position    | Texture |   Normals   |
vQuad = [[ -1.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0],  # v0
         [ -1.0,  1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0],  # v1
         [  1.0,  1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0],  # v2
         [  1.0, -1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0]]  # v3

# create a VBO with interleaved attributes
vboInterleaved = createVBO(np.asarray(vQuad, dtype=np.float32))

# ... before rendering, set the attribute pointers
GL.glBindBuffer(vboInterleaved.target, vboInterleaved.name)
gltools.setVertexAttribPointer(
    0, vboInterleaved, size=3, offset=0)  # vertex pointer
gltools.setVertexAttribPointer(
    8, vboInterleaved, size=2, offset=3)  # texture pointer
gltools.setVertexAttribPointer(
    3, vboInterleaved, size=3, offset=5)  # normals pointer

# Note, we specified `bind=False` since we are managing the binding
# state. It is recommended that you do this when setting up interleaved
# buffers to avoid re-binding the same buffer.

# draw red, full screen quad
GL.glColor3f(1.0, 0.0, 0.0)
```

(continues on next page)

(continued from previous page)

```
GL.glDrawArrays(GL.GL_QUADS, 0, vboInterleaved.shape[1])

# call these when done if `enable=True`
gltools.disableVertexAttribArray(0)
gltools.disableVertexAttribArray(8)
gltools.disableVertexAttribArray(1)

# unbind the buffer
GL.glBindBuffer(vboInterleaved.target, 0)
```

`psychopy.tools.gltools.`**`enableVertexAttribArray`**`(`*index*, *legacy=False*`)`

Enable a vertex attribute array. Attributes will be used for use by subsequent draw operations. Be sure to call *`disableVertexAttribArray()`* on the same attribute to prevent currently enabled attributes from affecting later rendering.

Parameters

- **`index`** (*`int`*) – Index of the attribute to enable. If *legacy=True*, this value should be a *GLenum* type corresponding to the capability to bind the buffer to, such as *GL_VERTEX_ARRAY*, *GL_TEXTURE_COORD_ARRAY*, *GL_NORMAL_ARRAY*, etc.
- **`legacy`** (*`bool`*, *`optional`*) – Use legacy vertex attributes (ie. *GL_VERTEX_ARRAY*, *GL_TEXTURE_COORD_ARRAY*, etc.) for backwards compatibility.

`psychopy.tools.gltools.`**`disableVertexAttribArray`**`(`*index*, *legacy=False*`)`

Disable a vertex attribute array.

Parameters

- **`index`** (*`int`*) – Index of the attribute to enable. If *legacy=True*, this value should be a *GLenum* type corresponding to the capability to bind the buffer to, such as *GL_VERTEX_ARRAY*, *GL_TEXTURE_COORD_ARRAY*, *GL_NORMAL_ARRAY*, etc.
- **`legacy`** (*`bool`*, *`optional`*) – Use legacy vertex attributes (ie. *GL_VERTEX_ARRAY*, *GL_TEXTURE_COORD_ARRAY*, etc.) for backwards compatibility.

Materials and Lighting

Tools for specifying the appearance of faces and shading. Note that these tools use the legacy OpenGL pipeline which may not be available on your platform. Use fragment/vertex shaders instead for newer applications.

`psychopy.tools.gltools.`**`createMaterial`**`(`*params=()*, *textures=()*, *face=1032*`)`

Create a new material.

Parameters

- **`params`** (`list` of `tuple`, optional) – List of material modes and values. Each mode is assigned a value as (mode, color). Modes can be GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_EMISSION, GL_SHININESS or GL_AMBIENT_AND_DIFFUSE. Colors must be a tuple of 4 floats which specify reflectance values for each RGBA component. The value of GL_SHININESS should be a single float. If no values are specified, an empty material will be created.
- **`textures`** (`list` of `tuple`, optional) – List of texture units and TexImage2D descriptors. These will be written to the ‘textures’ field of the returned descriptor. For example, [(GL.GL_TEXTURE0, texDesc0), (GL.GL_TEXTURE1, texDesc1)]. The number of texture units per-material is GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS.

- **face** (int, optional) – Faces to apply material to. Values can be GL_FRONT_AND_BACK, GL_FRONT and GL_BACK. The default is GL_FRONT_AND_BACK.

Returns
A descriptor with material properties.

Return type
Material

Examples

Creating a new material with given properties:

```
# The values for the material below can be found at
# http://devernay.free.fr/cours/opengl/materials.html

# create a gold material
gold = createMaterial([
    (GL.GL_AMBIENT, (0.24725, 0.19950, 0.07450, 1.0)),
    (GL.GL_DIFFUSE, (0.75164, 0.60648, 0.22648, 1.0)),
    (GL.GL_SPECULAR, (0.628281, 0.555802, 0.366065, 1.0)),
    (GL.GL_SHININESS, 0.4 * 128.0)])
```

Use the material when drawing:

```
useMaterial(gold)
drawVAO( ... )  # all meshes will be gold
useMaterial(None)  # turn off material when done
```

Create a red plastic material, but define reflectance and shine later:

```
red_plastic = createMaterial()

# you need to convert values to ctypes!
red_plastic.values[GL_AMBIENT] = (GLfloat * 4)(0.0, 0.0, 0.0, 1.0)
red_plastic.values[GL_DIFFUSE] = (GLfloat * 4)(0.5, 0.0, 0.0, 1.0)
red_plastic.values[GL_SPECULAR] = (GLfloat * 4)(0.7, 0.6, 0.6, 1.0)
red_plastic.values[GL_SHININESS] = 0.25 * 128.0

# set and draw
useMaterial(red_plastic)
drawVertexbuffers( ... )  # all meshes will be red plastic
useMaterial(None)
```

psychopy.tools.gltools.**useMaterial**(*material*, *useTextures=True*)

Use a material for proceeding vertex draws.

Parameters

- **material** (Material or None) – Material descriptor to use. Default material properties are set if None is specified. This is equivalent to disabling materials.
- **useTextures** (bool) – Enable textures. Textures specified in a material descriptor's 'texture' attribute will be bound and their respective texture units will be enabled. Note, when disabling materials, the value of useTextures must match the previous call. If there are no textures attached to the material, useTexture will be silently ignored.

Return type
None

Notes

1. If a material mode has a value of None, a color with all components 0.0 will be assigned.
2. Material colors and shininess values are accessible from shader programs after calling 'useMaterial'. Values can be accessed via built-in 'gl_FrontMaterial' and 'gl_BackMaterial' structures (e.g. gl_FrontMaterial.diffuse).

Examples

Use a material when drawing:

```
useMaterial(metalMaterials.gold)
drawVAO( ... )  # all meshes drawn will be gold
useMaterial(None)  # turn off material when done
```

`psychopy.tools.gltools.`**`createLight`**`(`*params=()*`)`

Create a point light source.

`psychopy.tools.gltools.`**`useLights`**`(`*lights*, *setupOnly=False*`)`

Use specified lights in successive rendering operations. All lights will be transformed using the present modelview matrix.

Parameters

- **`lights`** (`List` of `Light` or None) – Descriptor of a light source. If None, lighting is disabled.
- **`setupOnly`** (`bool`, optional) – Do not enable lighting or lights. Specify True if lighting is being computed via fragment shaders.

`psychopy.tools.gltools.`**`setAmbientLight`**`(`*color*`)`

Set the global ambient lighting for the scene when lighting is enabled. This is equivalent to GL.glLightModelfv(GL.GL_LIGHT_MODEL_AMBIENT, color) and does not contribute to the GL_MAX_LIGHTS limit.

Parameters
`color` (`tuple`) – Ambient lighting RGBA intensity for the whole scene.

Notes

If unset, the default value is (0.2, 0.2, 0.2, 1.0) when GL_LIGHTING is enabled.

Meshes

Tools for loading or procedurally generating meshes (3D models).

`psychopy.tools.gltools.`**`ObjMeshInfo`**`(`*vertexPos=None*, *texCoords=None*, *normals=None*, *faces=None*, *extents=None*, *mtlFile=None*`)`

Descriptor for mesh data loaded from a Wavefront OBJ file.

`psychopy.tools.gltools.`**`loadObjFile`**`(`*objFile*`)`

Load a Wavefront OBJ file (*.obj).

Loads vertex, normals, and texture coordinates from the provided **.obj* file into arrays. These arrays can be processed then loaded into vertex buffer objects (VBOs) for rendering. The **.obj* file must at least specify vertex position data to be loaded successfully. Normals and texture coordinates are optional.

Faces can be either triangles or quads, but not both. Faces are grouped by their materials. Index arrays are generated for each material present in the file.

Data from the returned *ObjMeshInfo* object can be used to create vertex buffer objects and arrays for rendering. See *Examples* below for details on how to do this.

Parameters
`objFile` (`str`) – Path to the **.OBJ* file to load.

Returns
Mesh data.

Return type
ObjMeshInfo

See also

`loadMtlFile`
Load a **.mtl* file.

Notes

1. This importer should work fine for most sanely generated files. Export your model with Blender for best results, even if you used some other package to create it.
2. The mesh cannot contain both triangles and quads.

Examples

Loading a **.obj* mode from file:

```
objModel = loadObjFile('/path/to/file.obj')
# load the material (*.mtl) file, textures are also loaded
mtllib = loadMtl('/path/to/' + objModel.mtlFile)
```

Creating separate vertex buffer objects (VBOs) for each vertex attribute:

```
vertexPosVBO = createVBO(objModel.vertexPos)
texCoordVBO = createVBO(objModel.texCoords)
normalsVBO = createVBO(objModel.normals)
```

Create vertex array objects (VAOs) to draw the mesh. We create VAOs for each face material:

```
objVAOs = {}  # dictionary for VAOs
# for each material create a VAO
# keys are material names, values are index buffers
for material, faces in objModel.faces.items():
    # convert index buffer to VAO
    indexBuffer =                 gltools.createVBO(
            faces.flatten(),  # flatten face index for element array
            target=GL.GL_ELEMENT_ARRAY_BUFFER,
            dataType=GL.GL_UNSIGNED_INT)

    # see `setVertexAttribPointer` for more information about attribute
    # pointer indices
```

(continues on next page)

(continued from previous page)

```
    objVAOs[material] = gltools.createVAO(
        {0: vertexPosVBO,  # 0 = gl_Vertex
         8: texCoordVBO,   # 8 = gl_MultiTexCoord0
         2: normalsVBO},   # 2 = gl_Normal
         indexBuffer=indexBuffer)

    # if using legacy attribute pointers, do this instead ...
    # objVAOs[key] = createVAO({GL_VERTEX_ARRAY: vertexPosVBO,
    #                           GL_TEXTURE_COORD_ARRAY: texCoordVBO,
    #                           GL_NORMAL_ARRAY: normalsVBO},
    #                           indexBuffer=indexBuffer,
    #                           legacy=True)  # this needs to be `True`
```

To render the VAOs using *objVAOs* created above, do the following:

```
for material, vao in objVAOs.items():
    useMaterial(mtllib[material])
    drawVAO(vao)

useMaterial(None)  # disable materials when done
```

Optionally, you can create a single-storage, interleaved VBO by using *numpy.hstack*. On some GL implementations, using single-storage buffers offers better performance:

```
interleavedData = numpy.hstack(
    (objModel.vertexPos, objModel.texCoords, objModel.normals))
vertexData = createVBO(interleavedData)
```

Creating VAOs with interleaved, single-storage buffers require specifying additional information, such as *size* and *offset*:

```
objVAOs = {}
for key, val in objModel.faces.items():
    indexBuffer =                 gltools.createVBO(
            faces.flatten(),
            target=GL.GL_ELEMENT_ARRAY_BUFFER,
            dataType=GL.GL_UNSIGNED_INT)

    objVAOs[key] = createVAO({0: (vertexData, 3, 0),  # size=3, offset=0
                              8: (vertexData, 2, 3),  # size=2, offset=3
                              2: (vertexData, 3, 5),  # size=3, offset=5
                              indexBuffer=val)
```

Drawing VAOs with interleaved buffers is exactly the same as shown before with separate buffers.

psychopy.tools.gltools.**loadMtlFile**(*mtllib*, *texParams=None*)

Load a material library file (*.mtl).

Parameters

- **mtllib** (*str*) – Path to the material library file.
- **texParams** (*list or tuple*) – Optional texture parameters for loaded textures. Texture parameters are specified as a list of tuples. Each item specifies the option and parameter. For instance, *[(GL.GL_TEXTURE_MAG_FILTER, GL.GL_LINEAR), . . .]*. By default, linear

filtering is used for both the minifying and magnification filter functions. This is adequate for most uses.

Returns

Dictionary of materials. Where each key is the material name found in the file, and values are *Material* namedtuple objects.

Return type

dict

See also

loadObjFile

Load an **.OBJ* file.

Examples

Load material associated with an **.OBJ* file:

```
objModel = loadObjFile('/path/to/file.obj')
# load the material (*.mtl) file, textures are also loaded
mtllib = loadMtl('/path/to/' + objModel.mtlFile)
```

Use a material when rendering vertex arrays:

```
useMaterial(mtllib[material])
drawVAO(vao)
useMaterial(None)  # disable materials when done
```

psychopy.tools.gltools.**createUVSphere**(*radius=0.5*, *sectors=16*, *stacks=16*, *flipFaces=False*)

Create a UV sphere.

Procedurally generate a UV sphere by specifying its radius, and number of stacks and sectors. The poles of the resulting sphere will be aligned with the Z-axis.

Surface normals and texture coordinates are automatically generated. The returned normals are computed to produce smooth shading.

Parameters

- **radius** (*float, optional*) – Radius of the sphere in scene units (usually meters). Default is 0.5.
- **sectors** (*int, optional*) – Number of longitudinal and latitudinal sub-divisions. Default is 16 for both.
- **stacks** (*int, optional*) – Number of longitudinal and latitudinal sub-divisions. Default is 16 for both.
- **flipFaces** (*bool, optional*) – If *True*, normals and face windings will be set to point inward towards the center of the sphere. Texture coordinates will remain the same. Default is *False*.

Returns

Vertex attribute arrays (position, texture coordinates, and normals) and triangle indices.

Return type

tuple

Examples

Create a UV sphere and VAO to render it:

```
vertices, textureCoords, normals, faces =             gltools.
↪createUVSphere(sectors=32, stacks=32)

vertexVBO = gltools.createVBO(vertices)
texCoordVBO = gltools.createVBO(textureCoords)
normalsVBO = gltools.createVBO(normals)
indexBuffer = gltools.createVBO(
    faces.flatten(),
    target=GL.GL_ELEMENT_ARRAY_BUFFER,
    dataType=GL.GL_UNSIGNED_INT)

vao = gltools.createVAO({0: vertexVBO, 8: texCoordVBO, 2: normalsVBO},
    indexBuffer=indexBuffer)

# in the rendering loop
gltools.drawVAO(vao, GL.GL_TRIANGLES)
```

The color of the sphere can be changed by calling *glColor**:

```
glColor4f(1.0, 0.0, 0.0, 1.0)  # red
gltools.drawVAO(vao, GL.GL_TRIANGLES)
```

Raw coordinates can be transformed prior to uploading to VBOs. Here we can rotate vertex positions and normals so the equator rests on Z-axis:

```
r = mt.rotationMatrix(90.0, (1.0, 0, 0.0))  # 90 degrees about +X axis
vertices = mt.applyMatrix(r, vertices)
normals = mt.applyMatrix(r, normals)
```

psychopy.tools.gltools.**createPlane**(*size=(1.0, 1.0)*)

Create a plane.

Procedurally generate a plane (or quad) mesh by specifying its size. Texture coordinates are computed automatically, with origin at the bottom left of the plane. The generated plane is perpendicular to the +Z axis, origin of the plane is at its center.

Parameters

size (*tuple or float*) – Dimensions of the plane. If a single value is specified, the plane will be square. Provide a tuple of floats to specify the width and length of the plane (eg. *size=(0.2, 1.3)*).

Returns

Vertex attribute arrays (position, texture coordinates, and normals) and triangle indices.

Return type

tuple

Examples

Create a plane mesh and draw it:

```
vertices, textureCoords, normals, faces = gltools.createPlane()
```

(continues on next page)

(continued from previous page)

```
vertexVBO = gltools.createVBO(vertices)
texCoordVBO = gltools.createVBO(textureCoords)
normalsVBO = gltools.createVBO(normals)
indexBuffer = gltools.createVBO(
    faces.flatten(),
    target=GL.GL_ELEMENT_ARRAY_BUFFER,
    dataType=GL.GL_UNSIGNED_INT)

vao = gltools.createVAO({0: vertexVBO, 8: texCoordVBO, 2: normalsVBO},
    indexBuffer=indexBuffer)

# in the rendering loop
gltools.drawVAO(vao, GL.GL_TRIANGLES)
```

psychopy.tools.gltools.**createMeshGridFromArrays**(*xvals*, *yvals*, *zvals=None*, *tessMode='diag'*, *computeNormals=True*)

Create a mesh grid using coordinates from arrays.

Generates a mesh using data in provided in 2D arrays of vertex coordinates. Triangle faces are automatically computed by this function by joining adjacent vertices at neighbouring indices in the array. Texture coordinates are generated covering the whole mesh, with origin at the bottom left.

Parameters

- **xvals** (*array_like*) – NxM arrays of X and Y coordinates. Both arrays must have the same shape. the resulting mesh will have a single vertex for each X and Y pair. Faces will be generated to connect adjacent coordinates in the array.
- **yvals** (*array_like*) – NxM arrays of X and Y coordinates. Both arrays must have the same shape. the resulting mesh will have a single vertex for each X and Y pair. Faces will be generated to connect adjacent coordinates in the array.
- **zvals** (*array_like, optional*) – NxM array of Z coordinates for each X and Y. Must have the same shape as X and Y. If not specified, the Z coordinates will be filled with zeros.
- **tessMode** (*str, optional*) – Tessellation mode. Specifies how faces are generated. Options are 'center', 'radial', and 'diag'. Default is 'diag'. Modes 'radial' and 'center' work best with odd numbered array dimensions.
- **computeNormals** (*bool, optional*) – Compute normals for the generated mesh. If *False*, all normals are set to face in the +Z direction. Presently, computing normals is a slow operation and may not be needed for some meshes.

Returns

Vertex attribute arrays (position, texture coordinates, and normals) and triangle indices.

Return type

tuple

Examples

Create a 3D sine grating mesh using 2D arrays:

```
x = np.linspace(0, 1.0, 32)
y = np.linspace(1.0, 0.0, 32)
xx, yy = np.meshgrid(x, y)
zz = np.tile(np.sin(np.linspace(0.0, 32., 32)) * 0.02, (32, 1))
```

(continues on next page)

(continued from previous page)

```
vertices, textureCoords, normals, faces =             gltools.
↪createMeshGridFromArrays(xx, yy, zz)
```

psychopy.tools.gltools.**createMeshGrid**(*size=(1.0, 1.0)*, *subdiv=0*, *tessMode='diag'*)

Create a grid mesh.

Procedurally generate a grid mesh by specifying its size and number of sub-divisions. Texture coordinates are computed automatically. The origin is at the center of the mesh. The generated grid is perpendicular to the +Z axis, origin of the grid is at its center.

Parameters

- **size** (*tuple or float*) – Dimensions of the mesh. If a single value is specified, the plane will be square. Provide a tuple of floats to specify the width and length of the plane (eg. *size=(0.2, 1.3)*).
- **subdiv** (*int, optional*) – Number of subdivisions. Zero subdivisions are applied by default, and the resulting mesh will only have vertices at the corners.
- **tessMode** (*str, optional*) – Tessellation mode. Specifies how faces are subdivided. Options are 'center', 'radial', and 'diag'. Default is 'diag'. Modes 'radial' and 'center' work best with an odd number of subdivisions.

Returns

Vertex attribute arrays (position, texture coordinates, and normals) and triangle indices.

Return type

tuple

Examples

Create a grid mesh and draw it:

```
vertices, textureCoords, normals, faces = gltools.createPlane()

vertexVBO = gltools.createVBO(vertices)
texCoordVBO = gltools.createVBO(textureCoords)
normalsVBO = gltools.createVBO(normals)
indexBuffer = gltools.createVBO(
    faces.flatten(),
    target=GL.GL_ELEMENT_ARRAY_BUFFER,
    dataType=GL.GL_UNSIGNED_INT)

vao = gltools.createVAO({0: vertexVBO, 8: texCoordVBO, 2: normalsVBO},
    indexBuffer=indexBuffer)

# in the rendering loop
gltools.drawVAO(vao, GL.GL_TRIANGLES)
```

Randomly displace vertices off the plane of the grid by setting the *Z* value per vertex:

```
vertices, textureCoords, normals, faces =             gltools.
↪createMeshGrid(subdiv=11)

numVerts = vertices.shape[0]
```

(continues on next page)

(continued from previous page)

```
vertices[:, 2] = np.random.uniform(-0.02, 0.02, (numVerts,)))  # Z

# you must recompute surface normals to get correct shading!
normals = gltools.calculateVertexNormals(vertices, faces)

# create a VAO as shown in the previous example here to draw it ...
```

`psychopy.tools.gltools.`**`createBox`**(*size=(1.0, 1.0, 1.0)*, *flipFaces=False*)

Create a box mesh.

Create a box mesh by specifying its *size* in three dimensions (x, y, z), or a single value (*float*) to create a cube. The resulting box will be centered about the origin. Texture coordinates and normals are automatically generated for each face.

Setting *flipFaces=True* will make faces and normals point inwards, this allows boxes to be viewed and lit correctly from the inside.

Parameters

- **size** (*tuple or float*) – Dimensions of the mesh. If a single value is specified, the box will be a cube. Provide a tuple of floats to specify the width, length, and height of the box (eg. *size=(0.2, 1.3, 2.1)*).
- **flipFaces** (*bool, optional*) – If *True*, normals and face windings will be set to point inward towards the center of the box. Texture coordinates will remain the same. Default is *False*.

Returns

Vertex attribute arrays (position, texture coordinates, and normals) and triangle indices.

Return type

tuple

Examples

Create a box mesh and draw it:

```
vertices, textureCoords, normals, faces = gltools.createBox()

vertexVBO = gltools.createVBO(vertices)
texCoordVBO = gltools.createVBO(textureCoords)
normalsVBO = gltools.createVBO(normals)
indexBuffer = gltools.createVBO(
    faces.flatten(),
    target=GL.GL_ELEMENT_ARRAY_BUFFER,
    dataType=GL.GL_UNSIGNED_INT)

vao = gltools.createVAO({0: vertexVBO, 8: texCoordVBO, 2: normalsVBO},
    indexBuffer=indexBuffer)

# in the rendering loop
gltools.drawVAO(vao, GL.GL_TRIANGLES)
```

`psychopy.tools.gltools.`**`transformMeshPosOri`**(*vertices*, *normals*, *pos=(0.0, 0.0, 0.0)*, *ori=(0.0, 0.0, 0.0, 1.0)*)

Transform a mesh.

Transform mesh vertices and normals to a new position and orientation using a position coordinate and rotation quaternion. Values *vertices* and *normals* must be the same shape. This is intended to be used when editing raw vertex data prior to rendering. Do not use this to change the configuration of an object while rendering.

Parameters

- **vertices** (*array_like*) – Nx3 array of vertices.
- **normals** (*array_like*) – Nx3 array of normals.
- **pos** (*array_like, optional*) – Position vector to transform mesh vertices. If Nx3, *vertices* will be transformed by corresponding rows of *pos*.
- **ori** (*array_like, optional*) – Orientation quaternion in form [x, y, z, w]. If Nx4, *vertices* and *normals* will be transformed by corresponding rows of *ori*.

Returns

Transformed vertices and normals.

Return type

tuple

Examples

Create and re-orient a plane to face upwards:

```
vertices, textureCoords, normals, faces = createPlane()

# rotation quaternion
qr = quatFromAxisAngle((1., 0., 0.), -90.0)  # -90 degrees about +X axis

# transform the normals and points
vertices, normals = transformMeshPosOri(vertices, normals, ori=qr)
```

Any *create** primitive generating function can be used inplace of *createPlane*.

psychopy.tools.gltools.**calculateVertexNormals**(*vertices*, *faces*, *shading='smooth'*)

Calculate vertex normals given vertices and triangle faces.

Finds all faces sharing a vertex index and sets its normal to either the face normal if *shading='flat'* or the average normals of adjacent faces if *shading='smooth'*. Note, this function does not convert between flat and smooth shading. Flat shading only works correctly if each vertex belongs to exactly one face.

The direction of the normals are determined by the winding order of triangles, assumed counter clock-wise (OpenGL default). Most 3D model editing software exports using this convention. If not, winding orders can be reversed by calling:

```
faces = numpy.fliplr(faces)
```

In some case when using 'smooth', creases may appear if vertices are at the same location, but do not share the same index. This may be desired in some cases, however one may use the `smoothCreases()` function computing normals to smooth out creases.

Parameters

- **vertices** (*array_like*) – Nx3 vertex positions.
- **faces** (*array_like*) – Nx3 vertex indices.
- **shading** (*str, optional*) – Shading mode. Options are 'smooth' and 'flat'. Flat only works with meshes where no vertex index is shared across faces, if not, the returned normals will be invalid.

Returns
Vertex normals array with the shame shape as *vertices*. Computed normals are normalized.

Return type
ndarray

Examples

Recomputing vertex normals for a UV sphere:

```
# create a sphere and discard normals
vertices, textureCoords, _, faces = gltools.createUVSphere()
normals = gltools.calculateVertexNormals(vertices, faces)
```

Miscellaneous

Miscellaneous tools for working with OpenGL.

`psychopy.tools.gltools.`**`getIntegerv`**`(`*parName*`)`

Get a single integer parameter value, return it as a Python integer.

Parameters
pName (*int*) – OpenGL property enum to query (e.g. GL_MAJOR_VERSION).

Return type
int

`psychopy.tools.gltools.`**`getFloatv`**`(`*parName*`)`

Get a single float parameter value, return it as a Python float.

Parameters
pName (*float*) – OpenGL property enum to query.

Return type
float

`psychopy.tools.gltools.`**`getString`**`(`*parName*`)`

Get a single string parameter value, return it as a Python UTF-8 string.

Parameters
pName (*int*) – OpenGL property enum to query (e.g. GL_VENDOR).

Return type
str

`psychopy.tools.gltools.`**`getOpenGLInfo`**`()`

Get general information about the OpenGL implementation on this machine. This should provide a consistent means of doing so regardless of the OpenGL interface we are using.

Returns are dictionary with the following fields:

```
vendor, renderer, version, majorVersion, minorVersion, doubleBuffer,
maxTextureSize, stereo, maxSamples, extensions
```

Supported extensions are returned as a list in the 'extensions' field. You can check if a platform supports an extension by checking the membership of the extension name in that list.

Return type
OpenGLInfo

psychopy.tools.gltools.**getModelViewMatrix**()

Get the present model matrix from the OpenGL matrix stack.

Returns
4x4 model/view matrix.

Return type
ndarray

psychopy.tools.gltools.**getProjectionMatrix**()

Get the present projection matrix from the OpenGL matrix stack.

Returns
4x4 projection matrix.

Return type
ndarray

Examples

Working with Framebuffer Objects (FBOs):

Creating an empty framebuffer with no attachments:

```
fbo = createFBO()  # invalid until attachments are added
```

Create a render target with multiple color texture attachments:

```
colorTex = createTexImage2D(1024,1024)  # empty texture
depthRb = createRenderbuffer(800,600,internalFormat=GL.GL_DEPTH24_STENCIL8)

GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, fbo.id)
attach(GL.GL_COLOR_ATTACHMENT0, colorTex)
attach(GL.GL_DEPTH_ATTACHMENT, depthRb)
attach(GL.GL_STENCIL_ATTACHMENT, depthRb)
# or attach(GL.GL_DEPTH_STENCIL_ATTACHMENT, depthRb)
GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, 0)
```

Attach FBO images using a context. This automatically returns to the previous FBO binding state when complete. This is useful if you don't know the current binding state:

```
with useFBO(fbo):
    attach(GL.GL_COLOR_ATTACHMENT0, colorTex)
    attach(GL.GL_DEPTH_ATTACHMENT, depthRb)
    attach(GL.GL_STENCIL_ATTACHMENT, depthRb)
```

How to set userData some custom function might access:

```
fbo.userData['flags'] = ['left_eye', 'clear_before_use']
```

Binding an FBO for drawing/reading:

```
GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, fb.id)
```

Depth-only framebuffers are valid, sometimes need for generating shadows:

```
depthTex = createTexImage2D(800, 600,
                            internalFormat=GL.GL_DEPTH_COMPONENT24,
                            pixelFormat=GL.GL_DEPTH_COMPONENT)
fbo = createFBO([(GL.GL_DEPTH_ATTACHMENT, depthTex)])
```

Deleting a framebuffer when done with it. This invalidates the framebuffer's ID and makes it available for use:

```
deleteFBO(fbo)
```

11.8.5 `psychopy.tools.imagetools`

Functions and classes related to image handling

array2image(a)	Takes an array and returns an image object (PIL).
image2array(im)	Takes an image object (PIL) and returns a numpy array.
makeImageAuto(inarray)	Combines float_uint8 and image2array operations ie.

Function details

`psychopy.tools.imagetools.`**`array2image`**(*a*)

Takes an array and returns an image object (PIL).

`psychopy.tools.imagetools.`**`image2array`**(*im*)

Takes an image object (PIL) and returns a numpy array.

`psychopy.tools.imagetools.`**`makeImageAuto`**(*inarray*)

Combines float_uint8 and image2array operations ie. scales a numeric array from -1:1 to 0:255 and converts to PIL image format.

11.8.6 `psychopy.tools.mathtools`

Assorted math functions for working with vectors, matrices, and quaternions. These functions are intended to provide basic support for common mathematical operations associated with displaying stimuli (e.g. animation, posing, rendering, etc.)

For tools related to view transformations, see *`viewtools`*. Various math functions for working with vectors, matrices, and quaternions.

Rigid Body

Classes for working with rigid body transformations.

RigidBodyPose([pos, ori, dtype])	Class for representing rigid body poses.
BoundingBox([extents, dtype])	Class for representing object bounding boxes.

psychopy.tools.mathtools.RigidBodyPose

class `psychopy.tools.mathtools.`**`RigidBodyPose`**(*pos=(0.0, 0.0, 0.0)*, *ori=(0.0, 0.0, 0.0, 1.0)*, *dtype=None*)

Class for representing rigid body poses.

This class is an abstract representation of a rigid body pose, where the position of the body in a scene is represented by a vector/coordinate and the orientation with a quaternion. Pose can be manipulated and interacted with using class methods and attributes. Rigid body poses assume a right-handed coordinate system (-Z is forward and +Y is up).

Poses can be converted to 4x4 transformation matrices with *getModelMatrix*. One can use these matrices when rendering to transform the vertices of a model associated with the pose by passing them to OpenGL. Matrices are cached internally to avoid recomputing them if *pos* and *ori* attributes have not been updated.

Operators * and ~ can be used on *RigidBodyPose* objects to combine and invert poses. For instance, you can multiply (*) poses to get a new pose which is the combination of both orientations and translations by:

```
newPose = rb1 * rb2
```

Likewise, a pose can be inverted by using the ~ operator:

```
invPose = ~rb
```

Multiplying a pose by its inverse will result in an identity pose with no translation and default orientation where *pos=[0, 0, 0]* and *ori=[0, 0, 0, 1]*:

```
identityPose = ~rb * rb
```

> ⚠ **Warning**
>
> This class is experimental and may result in undefined behavior.

Parameters

- **pos** (*array_like*) – Position vector *[x, y, z]* for the origin of the rigid body.
- **ori** (*array_like*) – Orientation quaternion *[x, y, z, w]* where *x*, *y*, *z* are imaginary and *w* is real.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. Default is *None* which uses the default data configured by *setDefaultPrecision*.

__init__(*pos=(0.0, 0.0, 0.0)*, *ori=(0.0, 0.0, 0.0, 1.0)*, *dtype=None*)

Parameters

- **pos** (*array_like*) – Position vector *[x, y, z]* for the origin of the rigid body.
- **ori** (*array_like*) – Orientation quaternion *[x, y, z, w]* where *x*, *y*, *z* are imaginary and *w* is real.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. Default is *None* which uses the default data configured by *setDefaultPrecision*.

Methods

__init__([pos, ori, dtype])	
alignTo(alignTo)	Align this pose to another point or pose.
clear()	Clear the pose, setting position and orientation to zero.
copy()	Get a new *RigidBodyPose* object which copies the position and orientation of this one.
distanceTo(v)	Get the distance to a pose or point in scene units.
getModelMatrix([inverse, out])	Get the present rigid body transformation as a 4x4 matrix.
getNormalMatrix([out])	Get the present normal matrix.
getOriAxisAngle([degrees])	Get the axis and angle of rotation for the rigid body.
getViewMatrix([inverse, out])	Convert this pose into a view matrix.
getYawPitchRoll([degrees])	Get the yaw, pitch and roll angles for this pose relative to the -Z world axis.
interp(end, s)	Interpolate between poses.
invert()	Invert this pose.
inverted()	Get a pose which is the inverse of this one.
isEqual(other)	Check if poses have similar orientation and position.
setIdentity()	Clear rigid body transformations (alias for *clear*).
setOriAxisAngle(axis, angle[, degrees])	Set the orientation of the rigid body using an *axis* and *angle*.
transform(v[, out])	Transform a vector using this pose.
transformNormal(n)	Rotate a normal vector with respect to this pose.

Attributes

at	Vector defining the forward direction (-Z) of this pose.
bounds	Bounding box associated with this pose.
dtype	Data type used for computations and arrays (*numpy.dtype*).
inverseModelMatrix	Inverse of the pose as a 4x4 model matrix (read-only).
inverseViewMatrix	The inverse of the 4x4 view matrix for this pose (read-only).
modelMatrix	Pose as a 4x4 model matrix (read-only).
normalMatrix	The 4x4 normal transformation matrix (read-only).
ori	Orientation quaternion (X, Y, Z, W).
pos	Position vector (X, Y, Z).
posOri	The position (x, y, z) and orientation (x, y, z, w).
up	Vector defining the up direction (+Y) of this pose.
viewMatrix	The 4x4 view matrix for this pose (read-only).

psychopy.tools.mathtools.BoundingBox

class psychopy.tools.mathtools.**BoundingBox**(*extents=None*, *dtype=None*)

Class for representing object bounding boxes.

A bounding box is a construct which represents a 3D rectangular volume about some pose, defined by its minimum and maximum extents in the reference frame of the pose. The axes of the bounding box are aligned to the axes of the world or the associated pose.

Bounding boxes are primarily used for visibility testing; to determine if the extents of an object associated with a pose (eg. the vertices of a model) falls completely outside of the viewing frustum. If so, the model can be culled during rendering to avoid wasting CPU/GPU resources on objects not visible to the viewer.

`__init__`(*extents=None*, *dtype=None*)

Methods

`__init__`([extents, dtype])	
`clear`()	Clear a bounding box, invalidating it.
`fit`(verts)	Fit the bounding box to vertices.

Attributes

`dtype`	Data type used for computations and arrays (*numpy.dtype*).
`extents`	
`isValid`	*True* if the bounding box is valid.

Vectors

Tools for working with 2D and 3D vectors.

`length`(v[, squared, out, dtype])	Get the length of a vector.
`normalize`(v[, out, dtype])	Normalize a vector or quaternion.
`orthogonalize`(v, n[, out, dtype])	Orthogonalize a vector relative to a normal vector.
`reflect`(v, n[, out, dtype])	Reflection of a vector.
`dot`(v0, v1[, out, dtype])	Dot product of two vectors.
`cross`(v0, v1[, out, dtype])	Cross product of 3D vectors.
`project`(v0, v1[, out, dtype])	Project a vector onto another.
`perp`(v, n[, norm, out, dtype])	Project *v* to be a perpendicular axis of *n*.
`lerp`(v0, v1, t[, out, dtype])	Linear interpolation (LERP) between two vectors/coordinates.
`distance`(v0, v1[, out, dtype])	Get the distance between vectors/coordinates.
`angleTo`(v, point[, degrees, out, dtype])	Get the relative angle to a point from a vector.
`bisector`(v0, v1[, norm, out, dtype])	Get the angle bisector.
`surfaceNormal`(tri[, norm, out, dtype])	Compute the surface normal of a given triangle.
`surfaceBitangent`(tri, uv[, norm, out, dtype])	Compute the bitangent vector of a given triangle.
`surfaceTangent`(tri, uv[, norm, out, dtype])	Compute the tangent vector of a given triangle.
`vertexNormal`(faceNorms[, norm, out, dtype])	Compute a vertex normal from shared triangles.
`fixTangentHandedness`(tangents, normals, ...)	Ensure the handedness of tangent vectors are all the same.
`ortho3Dto2D`(p, orig, normal, up[, right, dtype])	Get the planar coordinates of an orthogonal projection of a 3D point onto a 2D plane.
`transform`(pos, ori, points[, out, dtype])	Transform points using a position and orientation.
`scale`(sf, points[, out, dtype])	Scale points by a factor.

psychopy.tools.mathtools.length

psychopy.tools.mathtools.**length**(*v*, *squared=False*, *out=None*, *dtype=None*)

Get the length of a vector.

Parameters

- **v** (*array_like*) – Vector to normalize, can be Nx2, Nx3, or Nx4. If a 2D array is specified, rows are treated as separate vectors.
- **squared** (*bool, optional*) – If True the squared length is returned. The default is False.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

Length of vector *v*.

Return type

float or ndarray

psychopy.tools.mathtools.normalize

psychopy.tools.mathtools.**normalize**(*v*, *out=None*, *dtype=None*)

Normalize a vector or quaternion.

v

[array_like] Vector to normalize, can be Nx2, Nx3, or Nx4. If a 2D array is specified, rows are treated as separate vectors. All vectors should have nonzero length.

out

[ndarray, optional] Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.

dtype

[dtype or str, optional] Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

Normalized vector *v*.

Return type

ndarray

Notes

- If the vector has length is zero, a vector of all zeros is returned after normalization.

Examples

Normalize a vector:

```
v = [1., 2., 3., 4.]
vn = normalize(v)
```

The *normalize* function is vectorized. It's considerably faster to normalize large arrays of vectors than to call *normalize* separately for each one:

```
v = np.random.uniform(-1.0, 1.0, (1000, 4,))  # 1000 length 4 vectors
vn = np.zeros((1000, 4))  # place to write values
normalize(v, out=vn)  # very fast!

# don't do this!
for i in range(1000):
    vn[i, :] = normalize(v[i, :])
```

psychopy.tools.mathtools.orthogonalize

psychopy.tools.mathtools.**orthogonalize**(*v*, *n*, *out=None*, *dtype=None*)

Orthogonalize a vector relative to a normal vector.

This function ensures that *v* is perpendicular (or orthogonal) to *n*.

Parameters

- **v** (*array_like*) – Vector to orthogonalize, can be Nx2, Nx3, or Nx4. If a 2D array is specified, rows are treated as separate vectors.
- **n** (*array_like*) – Normal vector, must have same shape as *v*.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

Orthogonalized vector *v* relative to normal vector *n*.

Return type

ndarray

Warning

If *v* and *n* are the same, the direction of the perpendicular vector is indeterminate. The resulting vector is degenerate (all zeros).

psychopy.tools.mathtools.reflect

psychopy.tools.mathtools.**reflect**(*v*, *n*, *out=None*, *dtype=None*)

Reflection of a vector.

Get the reflection of *v* relative to normal *n*.

Parameters

- **v** (*array_like*) – Vector to reflect, can be Nx2, Nx3, or Nx4. If a 2D array is specified, rows are treated as separate vectors.
- **n** (*array_like*) – Normal vector, must have same shape as *v*.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.

- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns
Reflected vector *v* off normal *n*.

Return type
ndarray

psychopy.tools.mathtools.dot

psychopy.tools.mathtools.**dot**(*v0*, *v1*, *out=None*, *dtype=None*)

Dot product of two vectors.

The behaviour of this function depends on the format of the input arguments:

- If *v0* and *v1* are 1D, the dot product is returned as a scalar and *out* is ignored.
- If *v0* and *v1* are 2D, a 1D array of dot products between corresponding row vectors are returned.
- If either *v0* and *v1* are 1D and 2D, an array of dot products between each row of the 2D vector and the 1D vector are returned.

Parameters

- **v0** (*array_like*) – Vector(s) to compute dot products of (e.g. [x, y, z]). *v0* must have equal or fewer dimensions than *v1*.
- **v1** (*array_like*) – Vector(s) to compute dot products of (e.g. [x, y, z]). *v0* must have equal or fewer dimensions than *v1*.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns
Dot product(s) of *v0* and *v1*.

Return type
ndarray

psychopy.tools.mathtools.cross

psychopy.tools.mathtools.**cross**(*v0*, *v1*, *out=None*, *dtype=None*)

Cross product of 3D vectors.

The behavior of this function depends on the dimensions of the inputs:

- If *v0* and *v1* are 1D, the cross product is returned as 1D vector.
- If *v0* and *v1* are 2D, a 2D array of cross products between corresponding row vectors are returned.
- If either *v0* and *v1* are 1D and 2D, an array of cross products between each row of the 2D vector and the 1D vector are returned.

Parameters

- **v0** (*array_like*) – Vector(s) in form [x, y, z] or [x, y, z, 1].

- **v1** (*array_like*) – Vector(s) in form [x, y, z] or [x, y, z, 1].
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

Cross product of *v0* and *v1*.

Return type

ndarray

Notes

- If input vectors are 4D, the last value of cross product vectors is always set to one.
- If input vectors *v0* and *v1* are Nx3 and *out* is Nx4, the cross product is computed and the last column of *out* is filled with ones.

Examples

Find the cross product of two vectors:

```
a = normalize([1, 2, 3])
b = normalize([3, 2, 1])
c = cross(a, b)
```

If input arguments are 2D, the function returns the cross products of corresponding rows:

```
# create two 6x3 arrays with random numbers
shape = (6, 3,)
a = normalize(np.random.uniform(-1.0, 1.0, shape))
b = normalize(np.random.uniform(-1.0, 1.0, shape))
cprod = np.zeros(shape)  # output has the same shape as inputs
cross(a, b, out=cprod)
```

If a 1D and 2D vector are specified, the cross product of each row of the 2D array and the 1D array is returned as a 2D array:

```
a = normalize([1, 2, 3])
b = normalize(np.random.uniform(-1.0, 1.0, (6, 3,)))
cprod = np.zeros(a.shape)
cross(a, b, out=cprod)
```

psychopy.tools.mathtools.project

`psychopy.tools.mathtools.`**`project`**(*v0*, *v1*, *out=None*, *dtype=None*)

Project a vector onto another.

Parameters

- **v0** (*array_like*) – Vector can be Nx2, Nx3, or Nx4. If a 2D array is specified, rows are treated as separate vectors.
- **v1** (*array_like*) – Vector to project onto *v0*.

- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

Projection of vector *v0* on *v1*.

Return type

ndarray or float

psychopy.tools.mathtools.perp

psychopy.tools.mathtools.**perp**(*v*, *n*, *norm=True*, *out=None*, *dtype=None*)

Project *v* to be a perpendicular axis of *n*.

Parameters

- **v** (*array_like*) – Vector to project [x, y, z], may be Nx3.
- **n** (*array_like*) – Normal vector [x, y, z], may be Nx3.
- **norm** (*bool*) – Normalize the resulting axis. Default is *True*.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

Perpendicular axis of *n* from *v*.

Return type

ndarray

Examples

Determine the local *up* (y-axis) of a surface or plane given *normal*:

```
normal = [0., 0.70710678, 0.70710678]
up = [1., 0., 0.]

yaxis = perp(up, normal)
```

Do a cross product to get the x-axis perpendicular to both:

```
xaxis = cross(yaxis, normal)
```

psychopy.tools.mathtools.lerp

psychopy.tools.mathtools.**lerp**(*v0*, *v1*, *t*, *out=None*, *dtype=None*)

Linear interpolation (LERP) between two vectors/coordinates.

Parameters

- **v0** (*array_like*) – Initial vector/coordinate. Can be 2D where each row is a point.

- **v1** (*array_like*) – Final vector/coordinate. Must be the same shape as *v0*.
- **t** (*float*) – Interpolation weight factor [0, 1].
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

Vector at *t* with same shape as *v0* and *v1*.

Return type

ndarray

Examples

Find the coordinate of the midpoint between two vectors:

```
u = [0., 0., 0.]
v = [0., 0., 1.]
midpoint = lerp(u, v, 0.5)  # 0.5 to interpolate half-way between points
```

psychopy.tools.mathtools.distance

psychopy.tools.mathtools.**distance**(*v0*, *v1*, *out=None*, *dtype=None*)

Get the distance between vectors/coordinates.

The behaviour of this function depends on the format of the input arguments:

- If *v0* and *v1* are 1D, the distance is returned as a scalar and *out* is ignored.
- If *v0* and *v1* are 2D, an array of distances between corresponding row vectors are returned.
- If either *v0* and *v1* are 1D and 2D, an array of distances between each row of the 2D vector and the 1D vector are returned.

Parameters

- **v0** (*array_like*) – Vectors to compute the distance between.
- **v1** (*array_like*) – Vectors to compute the distance between.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

Distance between vectors *v0* and *v1*.

Return type

ndarray

psychopy.tools.mathtools.angleTo

`psychopy.tools.mathtools.`**`angleTo`**`(`*v*, *point*, *degrees=True*, *out=None*, *dtype=None*`)`

Get the relative angle to a point from a vector.

The behaviour of this function depends on the format of the input arguments:

- If *v0* and *v1* are 1D, the angle is returned as a scalar and *out* is ignored.
- If *v0* and *v1* are 2D, an array of angles between corresponding row vectors are returned.
- If either *v0* and *v1* are 1D and 2D, an array of angles between each row of the 2D vector and the 1D vector are returned.

Parameters

- **v** (*array_like*) – Direction vector [x, y, z].
- **point** (*array_like*) – Point(s) to compute angle to from vector *v*.
- **degrees** (*bool, optional*) – Return the resulting angles in degrees. If *False*, angles will be returned in radians. Default is *True*.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns

Distance between vectors *v0* and *v1*.

Return type

ndarray

psychopy.tools.mathtools.bisector

`psychopy.tools.mathtools.`**`bisector`**`(`*v0*, *v1*, *norm=False*, *out=None*, *dtype=None*`)`

Get the angle bisector.

Computes a vector which bisects the angle between *v0* and *v1*. Input vectors *v0* and *v1* must be non-zero.

Parameters

- **v0** (*array_like*) – Vectors to bisect [x, y, z]. Must be non-zero in length and have the same shape. Inputs can be Nx3 where the bisector for corresponding rows will be returned.
- **v1** (*array_like*) – Vectors to bisect [x, y, z]. Must be non-zero in length and have the same shape. Inputs can be Nx3 where the bisector for corresponding rows will be returned.
- **norm** (*bool, optional*) – Normalize the resulting bisector. Default is *False*.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns

Bisecting vector [x, y, z].

Return type
ndarray

psychopy.tools.mathtools.surfaceNormal

psychopy.tools.mathtools.**surfaceNormal**(*tri*, *norm=True*, *out=None*, *dtype=None*)

Compute the surface normal of a given triangle.

Parameters

- **tri** (*array_like*) – Triangle vertices as 2D (3x3) array [p0, p1, p2] where each vertex is a length 3 array [vx, xy, vz]. The input array can be 3D (Nx3x3) to specify multiple triangles.
- **norm** (*bool, optional*) – Normalize computed surface normals if True, default is True.
- **out** (*ndarray, optional*) – Optional output array. Must have one fewer dimensions than *tri*. The shape of the last dimension must be 3.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns
Surface normal of triangle *tri*.

Return type
ndarray

Examples

Compute the surface normal of a triangle:

```
vertices = [[-1., 0., 0.], [0., 1., 0.], [1, 0, 0]]
norm = surfaceNormal(vertices)
```

Find the normals for multiple triangles, and put results in a pre-allocated array:

```
vertices = [[[-1., 0., 0.], [0., 1., 0.], [1, 0, 0]],  # 2x3x3
            [[1., 0., 0.], [0., 1., 0.], [-1, 0, 0]]]
normals = np.zeros((2, 3))  # normals for two triangles
surfaceNormal(vertices, out=normals)
```

psychopy.tools.mathtools.surfaceBitangent

psychopy.tools.mathtools.**surfaceBitangent**(*tri*, *uv*, *norm=True*, *out=None*, *dtype=None*)

Compute the bitangent vector of a given triangle.

This function can be used to generate bitangent vertex attributes for normal mapping. After computing bitangents, one may orthogonalize them with vertex normals using the *orthogonalize()* function, or within the fragment shader. Uses texture coordinates at each triangle vertex to determine the direction of the vector.

Parameters

- **tri** (*array_like*) – Triangle vertices as 2D (3x3) array [p0, p1, p2] where each vertex is a length 3 array [vx, xy, vz]. The input array can be 3D (Nx3x3) to specify multiple triangles.
- **uv** (*array_like*) – Texture coordinates associated with each face vertex as a 2D array (3x2) where each texture coordinate is length 2 array [u, v]. The input array can be 3D (Nx3x2) to specify multiple texture coordinates if multiple triangles are specified.

- **norm** (*bool, optional*) – Normalize computed bitangents if `True`, default is `True`.
- **out** (*ndarray, optional*) – Optional output array. Must have one fewer dimensions than *tri*. The shape of the last dimension must be 3.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns
Surface bitangent of triangle *tri*.

Return type
ndarray

Examples

Computing the bitangents for two triangles from vertex and texture coordinates (UVs):

```
# array of triangle vertices (2x3x3)
tri = np.asarray([
    [(-1.0, 1.0, 0.0), (-1.0, -1.0, 0.0), (1.0, -1.0, 0.0)],   # 1
    [(-1.0, 1.0, 0.0), (-1.0, -1.0, 0.0), (1.0, -1.0, 0.0)]])  # 2

# array of triangle texture coordinates (2x3x2)
uv = np.asarray([
    [(0.0, 1.0), (0.0, 0.0), (1.0, 0.0)],   # 1
    [(0.0, 1.0), (0.0, 0.0), (1.0, 0.0)]])  # 2

bitangents = surfaceBitangent(tri, uv, norm=True)  # bitangets (2x3)
```

psychopy.tools.mathtools.surfaceTangent

`psychopy.tools.mathtools.`**`surfaceTangent`**(*tri*, *uv*, *norm=True*, *out=None*, *dtype=None*)

Compute the tangent vector of a given triangle.

This function can be used to generate tangent vertex attributes for normal mapping. After computing tangents, one may orthogonalize them with vertex normals using the `orthogonalize()` function, or within the fragment shader. Uses texture coordinates at each triangle vertex to determine the direction of the vector.

Parameters

- **tri** (*array_like*) – Triangle vertices as 2D (3x3) array [p0, p1, p2] where each vertex is a length 3 array [vx, xy, vz]. The input array can be 3D (Nx3x3) to specify multiple triangles.
- **uv** (*array_like*) – Texture coordinates associated with each face vertex as a 2D array (3x2) where each texture coordinate is length 2 array [u, v]. The input array can be 3D (Nx3x2) to specify multiple texture coordinates if multiple triangles are specified. If so *N* must be the same size as the first dimension of *tri*.
- **norm** (*bool, optional*) – Normalize computed tangents if `True`, default is `True`.
- **out** (*ndarray, optional*) – Optional output array. Must have one fewer dimensions than *tri*. The shape of the last dimension must be 3.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns
Surface normal of triangle *tri*.

Return type
ndarray

Examples

Compute surface normals, tangents, and bitangents for a list of triangles:

```
# triangle vertices (2x3x3)
vertices = [[[-1., 0., 0.], [0., 1., 0.], [1, 0, 0]],
            [[1., 0., 0.], [0., 1., 0.], [-1, 0, 0]]]

# array of triangle texture coordinates (2x3x2)
uv = np.asarray([
    [(0.0, 1.0), (0.0, 0.0), (1.0, 0.0)],   # 1
    [(0.0, 1.0), (0.0, 0.0), (1.0, 0.0)]])  # 2

normals = surfaceNormal(vertices)
tangents = surfaceTangent(vertices, uv)
bitangents = cross(normals, tangents)  # or use `surfaceBitangent`
```

Orthogonalize a surface tangent with a vertex normal vector to get the vertex tangent and bitangent vectors:

```
vertexTangent = orthogonalize(faceTangent, vertexNormal)
vertexBitangent = cross(vertexTangent, vertexNormal)
```

Ensure computed vectors have the same handedness, if not, flip the tangent vector (important for applications like normal mapping):

```
# tangent, bitangent, and normal are 2D
tangent[dot(cross(normal, tangent), bitangent) < 0.0, :] *= -1.0
```

psychopy.tools.mathtools.vertexNormal

psychopy.tools.mathtools.**vertexNormal**(*faceNorms*, *norm=True*, *out=None*, *dtype=None*)

Compute a vertex normal from shared triangles.

This function computes a vertex normal by averaging the surface normals of the triangles it belongs to. If model has no vertex normals, first use *surfaceNormal()* to compute them, then run *vertexNormal()* to compute vertex normal attributes.

While this function is mainly used to compute vertex normals, it can also be supplied triangle tangents and bitangents.

Parameters

- **faceNorms** (*array_like*) – An array (Nx3) of surface normals.
- **norm** (*bool, optional*) – Normalize computed normals if True, default is True.
- **out** (*ndarray, optional*) – Optional output array.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns
Vertex normal.

Return type
ndarray

Examples

Compute a vertex normal from the face normals of the triangles it belongs to:

```
normals = [[1., 0., 0.], [0., 1., 0.]]  # adjacent face normals
vertexNorm = vertexNormal(normals)
```

psychopy.tools.mathtools.fixTangentHandedness

psychopy.tools.mathtools.**fixTangentHandedness**(*tangents*, *normals*, *bitangents*, *out=None*, *dtype=None*)

Ensure the handedness of tangent vectors are all the same.

Often 3D computed tangents may not have the same handedness due to how texture coordinates are specified. This function takes input surface vectors are ensures that tangents have the same handedness. Use this function if you notice that normal mapping shading appears reversed with respect to the incident light direction. The output array of corrected tangents can be used inplace of the original.

Parameters

- **tangents** (*array_like*) – Input Nx3 arrays of triangle tangents, normals and bitangents. All arrays must have the same size.
- **normals** (*array_like*) – Input Nx3 arrays of triangle tangents, normals and bitangents. All arrays must have the same size.
- **bitangents** (*array_like*) – Input Nx3 arrays of triangle tangents, normals and bitangents. All arrays must have the same size.
- **out** (*ndarray, optional*) – Optional output array for tangents. If not specified, a new array of tangents will be allocated.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns
Array of tangents with handedness corrected.

Return type
ndarray

psychopy.tools.mathtools.ortho3Dto2D

psychopy.tools.mathtools.**ortho3Dto2D**(*p*, *orig*, *normal*, *up*, *right=None*, *dtype=None*)

Get the planar coordinates of an orthogonal projection of a 3D point onto a 2D plane.

This function gets the nearest point on the plane which a 3D point falls on the plane.

Parameters

- **p** (*array_like*) – Point to be projected on the plane.
- **orig** (*array_like*) – Origin of the plane to test [x, y, z].
- **normal** (*array_like*) – Normal vector of the plane [x, y, z], must be normalized.

- **up** (*array_like*) – Normalized up (+Y) direction of the plane's coordinate system. Must be perpendicular to *normal*.
- **right** (*array_like, optional*) – Perpendicular right (+X) axis. If not provided, the axis will be computed via the cross product between *normal* and *up*.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

Coordinates on the plane [X, Y] where the 3D point projects towards perpendicularly.

Return type

ndarray

Examples

This function can be used with `intersectRayPlane()` to find the location on the plane the ray intersects:

```
# plane information
planeOrigin = [0, 0, 0]
planeNormal = [0, 0, 1]  # must be normalized
planeUpAxis = perp([0, 1, 0], planeNormal)  # must also be normalized

# ray
rayDir = [0, 0, -1]
rayOrigin = [0, 0, 5]

# get the intersect in 3D world space
pnt = intersectRayPlane(rayOrigin, rayDir, planeOrigin, planeNormal)

# get the 2D coordinates on the plane the intersect occurred
planeX, planeY = ortho3Dto2D(pnt, planeOrigin, planeNormal, planeUpAxis)
```

psychopy.tools.mathtools.transform

`psychopy.tools.mathtools.`**`transform`**(*pos*, *ori*, *points*, *out=None*, *dtype=None*)

Transform points using a position and orientation. Points are rotated then translated.

Parameters

- **pos** (*array_like*) – Position vector in form [x, y, z] or [x, y, z, 1].
- **ori** (*array_like*) – Orientation quaternion in form [x, y, z, w] where w is real and x, y, z are imaginary components.
- **points** (*array_like*) – Point(s) [x, y, z] to transform.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

Transformed points.

Return type
ndarray

Examples

Transform points by a position coordinate and orientation quaternion:

```
# rigid body pose
ori = quatFromAxisAngle([0., 0., -1.], 90.0, degrees=True)
pos = [0., 1.5, -3.]
# points to transform
points = np.array([[0., 1., 0., 1.], [-1., 0., 0., 1.]])  # [x, y, z, 1]
outPoints = np.zeros_like(points)  # output array
transform(pos, ori, points, out=outPoints)  # do the transformation
```

You can get the same results as the previous example using a matrix by doing the following:

```
R = rotationMatrix(90., [0., 0., -1])
T = translationMatrix([0., 1.5, -3.])
M = concatenate([R, T])
applyMatrix(M, points, out=outPoints)
```

If you are defining transformations with quaternions and coordinates, you can skip the costly matrix creation process by using *transform*.

Notes

- In performance tests, *applyMatrix* is noticeably faster than *transform* for very large arrays, however this is only true if you are applying the same transformation to all points.
- If the input arrays for *points* or *pos* is Nx4, the last column is ignored.

psychopy.tools.mathtools.scale

`psychopy.tools.mathtools.`**`scale`**`(`*sf*, *points*, *out=None*, *dtype=None*`)`

Scale points by a factor.

This is useful for converting points between units, and to stretch or compress points along a given axis. Scaling can be uniform which the same factor is applied along all axes, or anisotropic along specific axes.

Parameters

- **sf** (*array_like or float*) – Scaling factor. If scalar, all points will be scaled uniformly by that factor. If a vector, scaling will be anisotropic along an axis.
- **points** (*array_like*) – Point(s) [x, y, z] to scale.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns
Scaled points.

Return type
ndarray

Examples

Apply uniform scaling to points, here we scale to convert points in centimeters to meters:

```
CM_TO_METERS = 1.0 / 100.0
pointsCM = [[1, 2, 3], [4, 5, 6], [-1, 1, 0]]
pointsM = scale(CM_TO_METERS, pointsCM)
```

Anisotropic scaling along the X and Y axis:

```
pointsM = scale((SCALE_FACTOR_X, SCALE_FACTOR_Y), pointsCM)
```

Scale only on the X axis:

```
pointsM = scale((SCALE_FACTOR_X,), pointsCM)
```

Apply scaling on the Z axis only:

```
pointsM = scale((1.0, 1.0, SCALE_FACTOR_Z), pointsCM)
```

Quaternions

Tools for working with *quaternions*. Quaternions are used primarily here to represent rotations in 3D space.

articulate(boneVecs, boneOris[, dtype])	Articulate an armature.
slerp(q0, q1, t[, shortest, out, dtype])	Spherical linear interpolation (SLERP) between two quaternions.
quatToAxisAngle(q[, degrees, dtype])	Convert a quaternion to *axis* and *angle* representation.
quatFromAxisAngle(axis, angle[, degrees, dtype])	Create a quaternion to represent a rotation about *axis* vector by *angle*.
quatYawPitchRoll(q[, degrees, out, dtype])	Get the yaw, pitch, and roll of a quaternion's orientation relative to the world -Z axis.
alignTo(v, t[, out, dtype])	Compute a quaternion which rotates one vector to align with another.
quatMagnitude(q[, squared, out, dtype])	Get the magnitude of a quaternion.
multQuat(q0, q1[, out, dtype])	Multiply quaternion *q0* and *q1*.
accumQuat(qlist[, out, dtype])	Accumulate quaternion rotations.
invertQuat(q[, out, dtype])	Get the multiplicative inverse of a quaternion.
applyQuat(q, points[, out, dtype])	Rotate points/coordinates using a quaternion.
quatToMatrix(q[, out, dtype])	Create a 4x4 rotation matrix from a quaternion.

psychopy.tools.mathtools.articulate

psychopy.tools.mathtools.**articulate**(*boneVecs*, *boneOris*, *dtype=None*)

Articulate an armature.

This function is used for forward kinematics and posing by specifying a list of 'bones'. A bone has a length and orientation, where sequential bones are linked end-to-end. Returns the transformed origins of the bones in scene coordinates and their orientations.

There are many applications for forward kinematics such as posing armatures and stimuli for display (eg. mocap data). Another application is for getting the location of the end effector of coordinate measuring hardware, where encoders measure the joint angles and the length of linking members are known. This can be used for computing pose from "Sword of Damocles"[1] like hardware or some other haptic input devices which the participant wears

[1] Sutherland, I. E. (1968). "A head-mounted three dimensional display". Proceedings of AFIPS 68, pp. 757-764

(eg. a glove that measures joint angles in the hand). The computed pose of the joints can be used to interact with virtual stimuli.

Parameters

- **boneVecs** (*array_like*) – Bone lengths [x, y, z] as an Nx3 array.
- **boneOris** (*array_like*) – Orientation of the bones as quaternions in form [x, y, z, w], relative to the previous bone.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

Array of bone origins and orientations. The first origin is root position which is always at [0, 0, 0]. Use `transform()` to reposition the armature, or create a transformation matrix and use *applyMatrix* to translate and rotate the whole armature into position.

Return type

tuple

References

Examples

Compute the orientations and origins of segments of an arm:

```
# bone lengths
boneLengths = [[0., 1., 0.], [0., 1., 0.], [0., 1., 0.]]

# create quaternions for joints
shoulder = mt.quatFromAxisAngle('-y', 45.0)
elbow = mt.quatFromAxisAngle('+z', 45.0)
wrist = mt.quatFromAxisAngle('+z', 45.0)

# articulate the parts of the arm
boxPos, boxOri = mt.articulate(pos, [shoulder, elbow, wrist])

# assign positions and orientations to 3D objects
shoulderModel.thePose.posOri = (boxPos[0, :], boxOri[0, :])
elbowModel.thePose.posOri = (boxPos[1, :], boxOri[1, :])
wristModel.thePose.posOri = (boxPos[2, :], boxOri[2, :])
```

psychopy.tools.mathtools.slerp

`psychopy.tools.mathtools.`**`slerp`**(*q0*, *q1*, *t*, *shortest=True*, *out=None*, *dtype=None*)

Spherical linear interpolation (SLERP) between two quaternions.

The behaviour of this function depends on the types of arguments:

- If *q0* and *q1* are both 1-D and *t* is scalar, the interpolation at *t* is returned.
- If *q0* and *q1* are both 2-D Nx4 arrays and *t* is scalar, an Nx4 array is returned with each row containing the interpolation at *t* for each quaternion pair at matching row indices in *q0* and *q1*.

Parameters

- **q0** (*array_like*) – Initial quaternion in form [x, y, z, w] where w is real and x, y, z are imaginary components.
- **q1** (*array_like*) – Final quaternion in form [x, y, z, w] where w is real and x, y, z are imaginary components.
- **t** (*float*) – Interpolation weight factor within interval 0.0 and 1.0.
- **shortest** (*bool, optional*) – Ensure interpolation occurs along the shortest arc along the 4-D hypersphere (default is *True*).
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns
Quaternion [x, y, z, w] at *t*.

Return type
ndarray

Examples

Interpolate between two orientations:

```
q0 = quatFromAxisAngle(90.0, degrees=True)
q1 = quatFromAxisAngle(-90.0, degrees=True)
# halfway between 90 and -90 is 0.0 or quaternion [0. 0. 0. 1.]
qr = slerp(q0, q1, 0.5)
```

Example of smooth rotation of an object with fixed angular velocity:

```
degPerSec = 10.0  # rotate a stimulus at 10 degrees per second

# initial orientation, axis rotates in the Z direction
qr = quatFromAxisAngle([0., 0., -1.], 0.0, degrees=True)
# amount to rotate every second
qv = quatFromAxisAngle([0., 0., -1.], degPerSec, degrees=True)

# ---- within main experiment loop ----
# `frameTime` is the time elapsed in seconds from last `slerp`.
qr = multQuat(qr, slerp((0., 0., 0., 1.), qv, degPerSec * frameTime))
_, angle = quatToAxisAngle(qr)  # discard axis, only need angle

# myStim is a GratingStim or anything with an 'ori' argument which
# accepts angle in degrees
myStim.ori = angle
myStim.draw()
```

psychopy.tools.mathtools.quatToAxisAngle

psychopy.tools.mathtools.**quatToAxisAngle**(*q*, *degrees=True*, *dtype=None*)

Convert a quaternion to *axis* and *angle* representation.

This allows you to use quaternions to set the orientation of stimuli that have an *ori* property.

Parameters

- **q** (*tuple, list or ndarray of float*) – Quaternion in form [x, y, z, w] where w is real and x, y, z are imaginary components.
- **degrees** (*bool, optional*) – Indicate *angle* is to be returned in degrees, otherwise *angle* will be returned in radians.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

Axis and angle of quaternion in form ([ax, ay, az], angle). If *degrees* is *True*, the angle returned is in degrees, radians if *False*.

Return type

tuple

Examples

Using a quaternion to rotate a stimulus a fixed angle each frame:

```
# initial orientation, axis rotates in the Z direction
qr = quatFromAxisAngle([0., 0., -1.], 0.0, degrees=True)
# rotation per-frame, here it's 0.1 degrees per frame
qf = quatFromAxisAngle([0., 0., -1.], 0.1, degrees=True)

# ---- within main experiment loop ----
# myStim is a GratingStim or anything with an 'ori' argument which
# accepts angle in degrees
qr = multQuat(qr, qf)  # cumulative rotation
_, angle = quatToAxisAngle(qr)  # discard axis, only need angle
myStim.ori = angle
myStim.draw()
```

psychopy.tools.mathtools.quatFromAxisAngle

psychopy.tools.mathtools.**quatFromAxisAngle**(*axis*, *angle*, *degrees=True*, *dtype=None*)

Create a quaternion to represent a rotation about *axis* vector by *angle*.

Parameters

- **axis** (*tuple, list, ndarray or str*) – Axis vector components or axis name. If a vector, input must be length 3 [x, y, z]. A string can be specified for rotations about world axes (eg. *'+x'*, *'-z'*, *'+y'*, etc.)
- **angle** (*float*) – Rotation angle in radians (or degrees if *degrees* is *True*. Rotations are right-handed about the specified *axis*.
- **degrees** (*bool, optional*) – Indicate *angle* is in degrees, otherwise *angle* will be treated as radians.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

Quaternion [x, y, z, w].

Return type
ndarray

Examples

Create a quaternion from specified *axis* and *angle*:

```
axis = [0., 0., -1.]  # rotate about -Z axis
angle = 90.0  # angle in degrees
ori = quatFromAxisAngle(axis, angle, degrees=True)  # using degrees!
```

psychopy.tools.mathtools.quatYawPitchRoll

psychopy.tools.mathtools.**quatYawPitchRoll**(*q*, *degrees=True*, *out=None*, *dtype=None*)

Get the yaw, pitch, and roll of a quaternion's orientation relative to the world -Z axis.

You can multiply the quaternion by the inverse of some other one to make the returned values referenced to a local coordinate system.

Parameters

- **q** (*tuple, list or ndarray of float*) – Quaternion in form [x, y, z, w] where w is real and x, y, z are imaginary components.
- **degrees** (*bool, optional*) – Indicate angles are to be returned in degrees, otherwise they will be returned in radians.
- **out** (*ndarray*) – Optional output array. Must have same *shape* and *dtype* as what is expected to be returned by this function of *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns
Yaw, pitch and roll [yaw, pitch, roll] of quaternion *q*.

Return type
ndarray

psychopy.tools.mathtools.alignTo

psychopy.tools.mathtools.**alignTo**(*v*, *t*, *out=None*, *dtype=None*)

Compute a quaternion which rotates one vector to align with another.

Parameters

- **v** (*array_like*) – Vector [x, y, z] to rotate. Can be Nx3, but must have the same shape as *t*.
- **t** (*array_like*) – Target [x, y, z] vector to align to. Can be Nx3, but must have the same shape as *v*.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns
Quaternion which rotates *v* to *t*.

Return type
ndarray

Examples

Rotate some vectors to align with other vectors, inputs should be normalized:

```
vec = [[1, 0, 0], [0, 1, 0], [1, 0, 0]]
targets = [[0, 1, 0], [0, -1, 0], [-1, 0, 0]]

qr = alignTo(vec, targets)
vecRotated = applyQuat(qr, vec)

numpy.allclose(vecRotated, targets)  # True
```

Get matrix which orients vertices towards a point:

```
point = [5, 6, 7]
vec = [0, 0, -1]  # initial facing is -Z (forward in GL)

targetVec = normalize(point - vec)
qr = alignTo(vec, targetVec)  # get rotation to align

M = quatToMatrix(qr)  # 4x4 transformation matrix
```

psychopy.tools.mathtools.quatMagnitude

psychopy.tools.mathtools.**quatMagnitude**(*q*, *squared=False*, *out=None*, *dtype=None*)

Get the magnitude of a quaternion.

A quaternion is normalized if its magnitude is 1.

Parameters

- **q** (*array_like*) – Quaternion(s) in form [x, y, z, w] where w is real and x, y, z are imaginary components.
- **squared** (*bool, optional*) – If True return the squared magnitude. If you are just checking if a quaternion is normalized, the squared magnitude will suffice to avoid the square root operation.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns
Magnitude of quaternion *q*.

Return type
float or ndarray

psychopy.tools.mathtools.multQuat

psychopy.tools.mathtools.**multQuat**(*q0*, *q1*, *out=None*, *dtype=None*)

Multiply quaternion *q0* and *q1*.

The orientation of the returned quaternion is the combination of the input quaternions.

Parameters

- **q0** (*array_like*) – Quaternions to multiply in form [x, y, z, w] where w is real and x, y, z are imaginary components. If 2D (Nx4) arrays are specified, quaternions are multiplied row-wise between each array.
- **q1** (*array_like*) – Quaternions to multiply in form [x, y, z, w] where w is real and x, y, z are imaginary components. If 2D (Nx4) arrays are specified, quaternions are multiplied row-wise between each array.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

Combined orientations of *q0* amd *q1*.

Return type

ndarray

Notes

- Quaternions are normalized prior to multiplication.

Examples

Combine the orientations of two quaternions:

```
a = quatFromAxisAngle([0, 0, -1], 45.0, degrees=True)
b = quatFromAxisAngle([0, 0, -1], 90.0, degrees=True)
c = multQuat(a, b)  # rotates 135 degrees about -Z axis
```

psychopy.tools.mathtools.accumQuat

psychopy.tools.mathtools.**accumQuat**(*qlist*, *out=None*, *dtype=None*)

Accumulate quaternion rotations.

Chain multiplies an Nx4 array of quaternions, accumulating their rotations. This function can be used for computing the orientation of joints in an armature for forward kinematics. The first quaternion is treated as the 'root' and the last is the orientation of the end effector.

Parameters

- **q** (*array_like*) – Nx4 array of quaternions to accumulate, where each row is a quaternion.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified. In this case, the same shape as *qlist*.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns
Nx4 array of quaternions.

Return type
ndarray

Examples

Get the orientation of joints in an armature if we know their relative angles:

```
shoulder = quatFromAxisAngle('-x', 45.0)  # rotate shoulder down 45 deg
elbow = quatFromAxisAngle('+x', 45.0)  # rotate elbow up 45 deg
wrist = quatFromAxisAngle('-x', 45.0)  # rotate wrist down 45 deg
finger = quatFromAxisAngle('+x', 0.0)  # keep finger in-line with wrist

armRotations = accumQuat([shoulder, elbow, wrist, finger])
```

psychopy.tools.mathtools.invertQuat

psychopy.tools.mathtools.**invertQuat**(*q*, *out=None*, *dtype=None*)

Get the multiplicative inverse of a quaternion.

This gives a quaternion which rotates in the opposite direction with equal magnitude. Multiplying a quaternion by its inverse returns an identity quaternion as both orientations cancel out.

Parameters

- **q** (*ndarray, list, or tuple of float*) – Quaternion to invert in form [x, y, z, w] where w is real and x, y, z are imaginary components. If *q* is 2D (Nx4), each row is treated as a separate quaternion and inverted.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns
Inverse of quaternion *q*.

Return type
ndarray

Examples

Show that multiplying a quaternion by its inverse returns an identity quaternion where [x=0, y=0, z=0, w=1]:

```
angle = 90.0
axis = [0., 0., -1.]
q = quatFromAxisAngle(axis, angle, degrees=True)
qinv = invertQuat(q)
qr = multQuat(q, qinv)
qi = np.array([0., 0., 0., 1.])  # identity quaternion
print(np.allclose(qi, qr))   # True
```

Notes

- Quaternions are normalized prior to inverting.

psychopy.tools.mathtools.applyQuat

psychopy.tools.mathtools.**applyQuat**(*q*, *points*, *out=None*, *dtype=None*)

Rotate points/coordinates using a quaternion.

This is similar to using *applyMatrix* with a rotation matrix. However, it is computationally less intensive to use *applyQuat* if one only wishes to rotate points.

Parameters

- **q** (*array_like*) – Quaternion to invert in form [x, y, z, w] where w is real and x, y, z are imaginary components.
- **points** (*array_like*) – 2D array of vectors or points to transform, where each row is a single point. Only the x, y, and z components (the first three columns) are rotated. Additional columns are copied.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

Transformed points.

Return type

ndarray

Examples

Rotate points using a quaternion:

```
points = [[1., 0., 0.], [0., -1., 0.]]
quat = quatFromAxisAngle(-90.0, [0., 0., -1.], degrees=True)
pointsRotated = applyQuat(quat, points)
# [[0. 1. 0.]
#  [1. 0. 0.]]
```

Show that you get the same result as a rotation matrix:

```
axis = [0., 0., -1.]
angle = -90.0
rotMat = rotationMatrix(axis, angle)[:3, :3]  # rotation sub-matrix only
rotQuat = quatFromAxisAngle(angle, axis, degrees=True)
points = [[1., 0., 0.], [0., -1., 0.]]
isClose = np.allclose(applyMatrix(rotMat, points),  # True
                      applyQuat(rotQuat, points))
```

Specifying an array to *q* where each row is a quaternion transforms points in corresponding rows of *points*:

```
points = [[1., 0., 0.], [0., -1., 0.]]
quats = [quatFromAxisAngle(-90.0, [0., 0., -1.], degrees=True),
```

(continues on next page)

(continued from previous page)

```
         quatFromAxisAngle(45.0, [0., 0., -1.], degrees=True)]
applyQuat(quats, points)
```

psychopy.tools.mathtools.quatToMatrix

psychopy.tools.mathtools.**quatToMatrix**(*q*, *out=None*, *dtype=None*)

Create a 4x4 rotation matrix from a quaternion.

Parameters

- **q** (*tuple, list or ndarray of float*) – Quaternion to convert in form [x, y, z, w] where w is real and x, y, z are imaginary components.
- **out** (*ndarray or None*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

4x4 rotation matrix in row-major order.

Return type

ndarray or None

Examples

Convert a quaternion to a rotation matrix:

```
point = [0., 1., 0., 1.]  # 4-vector form [x, y, z, 1.0]
ori = [0., 0., 0., 1.]
rotMat = quatToMatrix(ori)
# rotate 'point' using matrix multiplication
newPoint = np.matmul(rotMat.T, point)  # returns [-1., 0., 0., 1.]
```

Rotate all points in an array (each row is a coordinate):

```
points = np.asarray([[0., 0., 0., 1.],
                     [0., 1., 0., 1.],
                     [1., 1., 0., 1.]])
newPoints = points.dot(rotMat)
```

Notes

- Quaternions are normalized prior to conversion.

Matrices

Tools to creating and using affine transformation matrices.

matrixToQuat(m[, out, dtype])	Convert a rotation matrix to a quaternion.
matrixFromEulerAngles(rx, ry, rz[, degrees, ...])	Construct a 4x4 rotation matrix from Euler angles.
scaleMatrix(s[, out, dtype])	Create a scaling matrix.
rotationMatrix(angle[, axis, out, dtype])	Create a rotation matrix.
translationMatrix(t[, out, dtype])	Create a translation matrix.
invertMatrix(m[, out, dtype])	Invert a square matrix.
isOrthogonal(m)	Check if a square matrix is orthogonal.
isAffine(m)	Check if a 4x4 square matrix describes an affine transformation.
multMatrix(matrices[, reverse, out, dtype])	Chain multiplication of two or more matrices.
concatenate(matrices[, out, dtype])	Concatenate matrix transformations.
normalMatrix(modelMatrix[, out, dtype])	Get the normal matrix from a model matrix.
forwardProject(objPos, modelView, proj[, ...])	Project a point in a scene to a window coordinate.
reverseProject(winPos, modelView, proj[, ...])	Unproject window coordinates into object or scene coordinates.
lookAt(eyePos, centerPos[, upVec, out, dtype])	Create a transformation matrix to orient a view towards some point.
applyMatrix(m, points[, out, dtype])	Apply a matrix over a 2D array of points.
posOriToMatrix(pos, ori[, out, dtype])	Convert a rigid body pose to a 4x4 transformation matrix.

psychopy.tools.mathtools.matrixToQuat

psychopy.tools.mathtools.**matrixToQuat**(*m*, *out=None*, *dtype=None*)

Convert a rotation matrix to a quaternion.

Parameters

- **m** (*array_like*) – 3x3 rotation matrix (row-major). A 4x4 affine transformation matrix may be provided, assuming the top-left 3x3 sub-matrix is orthonormal and is a rotation group.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

Rotation quaternion.

Return type

ndarray

Notes

- Depending on the input, returned quaternions may not be exactly the same as the one used to construct the rotation matrix (i.e. by calling *quatToMatrix*), typically when a large rotation angle is used. However, the returned quaternion should result in the same rotation when applied to points.

Examples

Converting a rotation matrix from the OpenGL matrix stack to a quaternion:

```
glRotatef(45., -1, 0, 0)

m = np.zeros((4, 4), dtype='float32')  # store the matrix
GL.glGetFloatv(
    GL.GL_MODELVIEW_MATRIX,
    m.ctypes.data_as(ctypes.POINTER(ctypes.c_float)))

qr = matrixToQuat(m.T)  # must be transposed
```

Interpolation between two 4x4 transformation matrices:

```
interpWeight = 0.5

posStart = mStart[:3, 3]
oriStart = matrixToQuat(mStart)

posEnd = mEnd[:3, 3]
oriEnd = matrixToQuat(mEnd)

oriInterp = slerp(qStart, qEnd, interpWeight)
posInterp = lerp(posStart, posEnd, interpWeight)

mInterp = posOriToMatrix(posInterp, oriInterp)
```

psychopy.tools.mathtools.matrixFromEulerAngles

psychopy.tools.mathtools.**matrixFromEulerAngles**(*rx*, *ry*, *rz*, *degrees=True*, *out=None*, *dtype=None*)

Construct a 4x4 rotation matrix from Euler angles.

Rotations are combined by first rotating about the X axis, then Y, and finally Z.

Parameters

- **rx** (*float*) – Rotation angles (pitch, yaw, and roll).
- **ry** (*float*) – Rotation angles (pitch, yaw, and roll).
- **rz** (*float*) – Rotation angles (pitch, yaw, and roll).
- **degrees** (*bool, optional*) – Rotation angles are specified in degrees. If *False*, they will be assumed as radians. Default is *True*.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

4x4 rotation matrix.

Return type

ndarray

Examples

Demonstration of how a combination of axis-angle rotations is equivalent to a single call of *matrixFromEulerAngles*:

```
m1 = matrixFromEulerAngles(90., 45., 135.))

# construct rotation matrix from 3 orthogonal rotations
rx = rotationMatrix(90., (1, 0, 0))  # x-axis
ry = rotationMatrix(45., (0, 1, 0))  # y-axis
rz = rotationMatrix(135., (0, 0, 1))  # z-axis
m2 = concatenate([rz, ry, rx])  # note the order

print(numpy.allclose(m1, m2))  # True
```

Not only does *matrixFromEulerAngles* require less code, it also is considerably more efficient than constructing and multiplying multiple matrices.

psychopy.tools.mathtools.scaleMatrix

psychopy.tools.mathtools.**scaleMatrix**(*s*, *out=None*, *dtype=None*)

Create a scaling matrix.

The resulting matrix is the same as a generated by a *glScale* call.

Parameters

- **s** (*array_like, float or int*) – Scaling factor(s). If *s* is scalar (float), scaling will be uniform. Providing a vector of scaling values [sx, sy, sz] will result in an anisotropic scaling matrix if any of the values differ.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

4x4 scaling matrix in row-major order.

Return type

ndarray

psychopy.tools.mathtools.rotationMatrix

psychopy.tools.mathtools.**rotationMatrix**(*angle*, *axis=(0.0, 0.0, -1.0)*, *out=None*, *dtype=None*)

Create a rotation matrix.

The resulting matrix will rotate points about *axis* by *angle*. The resulting matrix is similar to that produced by a *glRotate* call.

Parameters

- **angle** (*float*) – Rotation angle in degrees.
- **axis** (*array_like or str*) – Axis vector components or axis name. If a vector, input must be length 3. A string can be specified for rotations about world axes (eg. *'+x'*, *'-z'*, *'+y'*, etc.)

- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

4x4 scaling matrix in row-major order. Will be the same array as *out* if specified, if not, a new array will be allocated.

Return type

ndarray

Notes

- Vector *axis* is normalized before creating the matrix.

psychopy.tools.mathtools.translationMatrix

psychopy.tools.mathtools.**translationMatrix**(*t*, *out=None*, *dtype=None*)

Create a translation matrix.

The resulting matrix is the same as generated by a *glTranslate* call.

Parameters

- **t** (*ndarray, tuple, or list of float*) – Translation vector [tx, ty, tz].
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

4x4 translation matrix in row-major order. Will be the same array as *out* if specified, if not, a new array will be allocated.

Return type

ndarray

psychopy.tools.mathtools.invertMatrix

psychopy.tools.mathtools.**invertMatrix**(*m*, *out=None*, *dtype=None*)

Invert a square matrix.

Parameters

- **m** (*array_like*) – Square matrix to invert. Inputs can be 4x4, 3x3 or 2x2.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

Matrix which is the inverse of *m*

Return type
ndarray

psychopy.tools.mathtools.isOrthogonal

psychopy.tools.mathtools.**isOrthogonal**(*m*)

Check if a square matrix is orthogonal.

If a matrix is orthogonal, its columns form an orthonormal basis and is non-singular. An orthogonal matrix is invertible by simply taking the transpose of the matrix.

Parameters
m (*array_like*) – Square matrix, either 2x2, 3x3 or 4x4.

Returns
True if the matrix is orthogonal.

Return type
bool

psychopy.tools.mathtools.isAffine

psychopy.tools.mathtools.**isAffine**(*m*)

Check if a 4x4 square matrix describes an affine transformation.

Parameters
m (*array_like*) – 4x4 transformation matrix.

Returns
True if the matrix is affine.

Return type
bool

psychopy.tools.mathtools.multMatrix

psychopy.tools.mathtools.**multMatrix**(*matrices*, *reverse=False*, *out=None*, *dtype=None*)

Chain multiplication of two or more matrices.

Multiply a sequence of matrices together, reducing to a single product matrix. For instance, specifying *matrices* the sequence of matrices (A, B, C, D) will return the product (((AB)C)D). If *reverse=True*, the product will be (A(B(CD))).

Alternatively, a 3D array can be specified to *matrices* as a stack, where an index along axis 0 references a 2D slice storing matrix values. The product of the matrices along the axis will be returned. This is a bit more efficient than specifying separate matrices in a sequence, but the difference is negligible when only a few matrices are being multiplied.

Parameters

- **matrices** (*list, tuple or ndarray*) – Sequence or stack of matrices to multiply. All matrices must have the same dimensions.
- **reverse** (*bool, optional*) – Multiply matrices right-to-left. This is useful when dealing with transformation matrices, where the order of operations for transforms will appear the same as the order the matrices are specified. Default is 'False'. When *True*, this function behaves similarly to *concatenate()*.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.

- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns
Matrix product.

Return type
ndarray

Notes

- You may use *numpy.matmul* when dealing with only two matrices instead of *multMatrix*.
- If a single matrix is specified, the returned product will have the same values.

Examples

Chain multiplication of SRT matrices:

```
translate = translationMatrix((0.035, 0, -0.5))
rotate = rotationMatrix(90.0, (0, 1, 0))
scale = scaleMatrix(2.0)

SRT = multMatrix((translate, rotate, scale))
```

Same as above, but matrices are in a 3x4x4 array:

```
matStack = np.array((translate, rotate, scale))

# or ...
# matStack = np.zeros((3, 4, 4))
# matStack[0, :, :] = translate
# matStack[1, :, :] = rotate
# matStack[2, :, :] = scale

SRT = multMatrix(matStack)
```

Using *reverse=True* allows you to specify transformation matrices in the order which they will be applied:

```
SRT = multMatrix(np.array((scale, rotate, translate)), reverse=True)
```

psychopy.tools.mathtools.concatenate

psychopy.tools.mathtools.**concatenate**(*matrices*, *out=None*, *dtype=None*)

Concatenate matrix transformations.

Chain multiply matrices describing transform operations into a single matrix product, that when applied, transforms points and vectors with each operation in the order they're specified.

Parameters

- **matrices** (*list or tuple*) – List of matrices to concatenate. All matrices must all have the same size, usually 4x4 or 3x3.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.

- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns
Matrix product.

Return type
ndarray

See also

- multMatrix : Chain multiplication of matrices.

Notes

- This function should only be used for combining transformation matrices. Use *multMatrix* for general matrix chain multiplication.

Examples

Create an SRT (scale, rotate, and translate) matrix to convert model-space coordinates to world-space:

```
S = scaleMatrix([2.0, 2.0, 2.0])  # scale model 2x
R = rotationMatrix(-90., [0., 0., -1])  # rotate -90 about -Z axis
T = translationMatrix([0., 0., -5.])  # translate point 5 units away

# product matrix when applied to points will scale, rotate and transform
# in that order.
SRT = concatenate([S, R, T])

# transform a point in model-space coordinates to world-space
pointModel = np.array([0., 1., 0., 1.])
pointWorld = np.matmul(SRT, pointModel.T)  # point in WCS
# ... or ...
pointWorld = matrixApply(SRT, pointModel)
```

Create a model-view matrix from a world-space pose represented by an orientation (quaternion) and position (vector). The resulting matrix will transform model-space coordinates to eye-space:

```
# eye pose as quaternion and vector
stimOri = quatFromAxisAngle([0., 0., -1.], -45.0)
stimPos = [0., 1.5, -5.]

# create model matrix
R = quatToMatrix(stimOri)
T = translationMatrix(stimPos)
M = concatenate(R, T)  # model matrix

# create a view matrix, can also be represented as 'pos' and 'ori'
eyePos = [0., 1.5, 0.]
eyeFwd = [0., 0., -1.]
eyeUp = [0., 1., 0.]
V = lookAt(eyePos, eyeFwd, eyeUp)  # from viewtools
```

(continues on next page)

(continued from previous page)

```

# modelview matrix
MV = concatenate([M, V])
```

You can put the created matrix in the OpenGL matrix stack as shown below. Note that the matrix must have a 32-bit floating-point data type and needs to be loaded transposed since OpenGL takes matrices in column-major order:

```
GL.glMatrixMode(GL.GL_MODELVIEW)

# pyglet
MV = np.asarray(MV, dtype='float32')  # must be 32-bit float!
ptrMV = MV.ctypes.data_as(ctypes.POINTER(ctypes.c_float))
GL.glLoadTransposeMatrixf(ptrMV)

# PyOpenGL
MV = np.asarray(MV, dtype='float32')
GL.glLoadTransposeMatrixf(MV)
```

Furthermore, you can convert a point from model-space to homogeneous clip-space by concatenating the projection, view, and model matrices:

```
# compute projection matrix, functions here are from 'viewtools'
screenWidth = 0.52
screenAspect = w / h
scrDistance = 0.55
frustum = computeFrustum(screenWidth, screenAspect, scrDistance)
P = perspectiveProjectionMatrix(*frustum)

# multiply model-space points by MVP to convert them to clip-space
MVP = concatenate([M, V, P])
pointModel = np.array([0., 1., 0., 1.])
pointClipSpace = np.matmul(MVP, pointModel.T)
```

psychopy.tools.mathtools.normalMatrix

psychopy.tools.mathtools.**normalMatrix**(*modelMatrix*, *out=None*, *dtype=None*)

Get the normal matrix from a model matrix.

Parameters

- **modelMatrix** (*array_like*) – 4x4 homogeneous model matrix.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

Normal matrix.

Return type

ndarray

psychopy.tools.mathtools.forwardProject

psychopy.tools.mathtools.**forwardProject**(*objPos*, *modelView*, *proj*, *viewport=None*, *out=None*, *dtype=None*)

Project a point in a scene to a window coordinate.

This function is similar to *gluProject* and can be used to find the window coordinate which a point projects to.

Parameters

- **objPos** (*array_like*) – Object coordinates (x, y, z). If an Nx3 array of coordinates is specified, where each row contains a window coordinate this function will return an array of projected coordinates with the same size.
- **modelView** (*array_like*) – 4x4 combined model and view matrix for returned value to be object coordinates. Specify only the view matrix for a coordinate in the scene.
- **proj** (*array_like*) – 4x4 projection matrix used for rendering.
- **viewport** (*array_like*) – Viewport rectangle for the window [x, y, w, h]. If not specified, the returned values will be in normalized device coordinates.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

Normalized device or viewport coordinates [x, y, z] of the point. The *z* component is similar to the depth buffer value for the object point.

Return type

ndarray

psychopy.tools.mathtools.reverseProject

psychopy.tools.mathtools.**reverseProject**(*winPos*, *modelView*, *proj*, *viewport=None*, *out=None*, *dtype=None*)

Unproject window coordinates into object or scene coordinates.

This function works like *gluUnProject* and can be used to find to an object or scene coordinate at the point on-screen (mouse coordinate or pixel). The coordinate can then be used to create a direction vector from the viewer's eye location. Another use of this function is to convert depth buffer samples to object or scene coordinates. This is the inverse operation of *forwardProject()*.

Parameters

- **winPos** (*array_like*) – Window coordinates (x, y, z). If *viewport* is not specified, these should be normalized device coordinates. If an Nx3 array of coordinates is specified, where each row contains a window coordinate this function will return an array of unprojected coordinates with the same size. Usually, you only need to specify the *x* and *y* coordinate, leaving *z* as zero. However, you can specify *z* if sampling from a depth map or buffer to convert a depth sample to an actual location.
- **modelView** (*array_like*) – 4x4 combined model and view matrix for returned value to be object coordinates. Specify only the view matrix for a coordinate in the scene.
- **proj** (*array_like*) – 4x4 projection matrix used for rendering.

- **viewport** (*array_like*) – Viewport rectangle for the window [x, y, w, h]. Do not specify one if *winPos* is in already in normalized device coordinates.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

Object or scene coordinates.

Return type

ndarray

psychopy.tools.mathtools.lookAt

psychopy.tools.mathtools.**lookAt**(*eyePos*, *centerPos*, *upVec=(0.0, 1.0, 0.0)*, *out=None*, *dtype=None*)

Create a transformation matrix to orient a view towards some point.

Based on the same algorithm as 'gluLookAt'. This does not generate a projection matrix, but rather the matrix to transform the observer's view in the scene.

For more information see: https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/gluLookAt.xml

Parameters

- **eyePos** (*list of float or ndarray*) – Eye position in the scene.
- **centerPos** (*list of float or ndarray*) – Position of the object center in the scene.
- **upVec** (*list of float or ndarray, optional*) – Vector defining the up vector. Default is +Y is up.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for arrays, can either be 'float32' or 'float64'. If *None* is specified, the data type is inferred by *out*. If *out* is not provided, the default is 'float64'.

Returns

4x4 view matrix

Return type

ndarray

Notes

- This function was moved from *viewtools* in version 2025.1.0.
- The returned matrix is row-major. Values are floats with 32-bits of precision stored as a contiguous (C-order) array.

psychopy.tools.mathtools.applyMatrix

psychopy.tools.mathtools.**applyMatrix**(*m*, *points*, *out=None*, *dtype=None*)

Apply a matrix over a 2D array of points.

This function behaves similarly to the following *Numpy* statement:

```
points[:, :] = points.dot(m.T)
```

Transformation matrices specified to *m* must have dimensions 4x4, 3x4, 3x3 or 2x2. With the exception of 4x4 matrices, input *points* must have the same number of columns as the matrix has rows. 4x4 matrices can be used to transform both Nx4 and Nx3 arrays.

Parameters

- **m** (*array_like*) – Matrix with dimensions 2x2, 3x3, 3x4 or 4x4.
- **points** (*array_like*) – 2D array of points/coordinates to transform. Each row should have length appropriate for the matrix being used. If not, a square submatrix will be taken from the input matrix with dimensions equal to the number of columns in *points*.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

Transformed coordinates.

Return type

ndarray

Notes

- Input (*points*) and output (*out*) arrays cannot be the same instance for this function.
- In the case of 4x4 input matrices, this function performs optimizations based on whether the input matrix is affine, greatly improving performance when working with Nx3 arrays.

Examples

Construct a matrix and transform a point:

```
# identity 3x3 matrix for this example
M = [[1.0, 0.0, 0.0],
     [0.0, 1.0, 0.0],
     [0.0, 0.0, 1.0]]

pnt = [1.0, 0.0, 0.0]

pntNew = applyMatrix(M, pnt)
```

Construct an SRT matrix (scale, rotate, transform) and transform an array of points:

```
S = scaleMatrix([5.0, 5.0, 5.0])  # scale 5x
R = rotationMatrix(180., [0., 0., -1])  # rotate 180 degrees
T = translationMatrix([0., 1.5, -3.])  # translate point up and away
M = concatenate([S, R, T])  # create transform matrix

# points to transform
points = np.array([[0., 1., 0., 1.], [-1., 0., 0., 1.]]) # [x, y, z, w]
newPoints = applyMatrix(M, points)  # apply the transformation
```

Convert CIE-XYZ colors to sRGB:

```
sRGBMatrix = [[3.2404542, -1.5371385, -0.4985314],
              [-0.969266,  1.8760108,  0.041556 ],
              [0.0556434, -0.2040259,  1.0572252]]

colorsRGB = applyMatrix(sRGBMatrix, colorsXYZ)
```

psychopy.tools.mathtools.posOriToMatrix

psychopy.tools.mathtools.**posOriToMatrix**(*pos*, *ori*, *out=None*, *dtype=None*)

Convert a rigid body pose to a 4x4 transformation matrix.

A pose is represented by a position coordinate *pos* and orientation quaternion *ori*.

Parameters

- **pos** (*ndarray, tuple, or list of float*) – Position vector [x, y, z].
- **ori** (*tuple, list or ndarray of float*) – Orientation quaternion in form [x, y, z, w] where w is real and x, y, z are imaginary components.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

4x4 transformation matrix.

Return type

ndarray

Collisions

Tools for determining whether a vector intersects a solid or bounding volume.

fitBBox(points[, dtype])	Fit an axis-aligned bounding box around points.
computeBBoxCorners(extents[, dtype])	Get the corners of an axis-aligned bounding box.
intersectRayPlane(rayOrig, rayDir, ...[, dtype])	Get the point which a ray intersects a plane.
intersectRaySphere(rayOrig, rayDir[, ...])	Calculate the points which a ray/line intersects a sphere (if any).
intersectRayAABB(rayOrig, rayDir, ...[, dtype])	Find the point a ray intersects an axis-aligned bounding box (AABB).
intersectRayOBB(rayOrig, rayDir, ...[, dtype])	Find the point a ray intersects an oriented bounding box (OBB).
intersectRayTriangle(rayOrig, rayDir, tri[, ...])	Get the intersection of a ray and triangle(s).

psychopy.tools.mathtools.fitBBox

psychopy.tools.mathtools.**fitBBox**(*points*, *dtype=None*)

Fit an axis-aligned bounding box around points.

This computes the minimum and maximum extents for a bounding box to completely enclose *points*. Keep in mind the output in bounds are axis-aligned and may not optimally fits the points (i.e. fits the points with the minimum required volume). However, this should work well enough for applications such as visibility testing (see *~psychopy.tools.viewtools.volumeVisible* for more information..

Parameters

- `points` (*array_like*) – Nx3 or Nx4 array of points to fit the bounding box to.
- `dtype` (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

Extents (mins, maxs) as a 2x3 array.

Return type

ndarray

See also

`computeBBoxCorners`
Convert bounding box extents to corners.

psychopy.tools.mathtools.computeBBoxCorners

psychopy.tools.mathtools.**computeBBoxCorners**(*extents*, *dtype=None*)

Get the corners of an axis-aligned bounding box.

Parameters

- `extents` (*array_like*) – 2x3 array indicating the minimum and maximum extents of the bounding box.
- `dtype` (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

8x4 array of points defining the corners of the bounding box.

Return type

ndarray

Examples

Compute the corner points of a bounding box:

```
minExtent = [-1, -1, -1]
maxExtent = [1, 1, 1]
corners = computeBBoxCorners([minExtent, maxExtent])

# [[ 1.  1.  1.  1.]
#  [-1.  1.  1.  1.]
#  [ 1. -1.  1.  1.]
#  [-1. -1.  1.  1.]
#  [ 1.  1. -1.  1.]
#  [-1.  1. -1.  1.]
```

(continues on next page)

(continued from previous page)

```
#  [ 1. -1. -1.  1.]
#  [-1. -1. -1.  1.]]
```

psychopy.tools.mathtools.intersectRayPlane

psychopy.tools.mathtools.**intersectRayPlane**(*rayOrig*, *rayDir*, *planeOrig*, *planeNormal*, *dtype=None*)

Get the point which a ray intersects a plane.

Parameters

- **rayOrig** (*array_like*) – Origin of the line in space [x, y, z].
- **rayDir** (*array_like*) – Direction vector of the line [x, y, z].
- **planeOrig** (*array_like*) – Origin of the plane to test [x, y, z].
- **planeNormal** (*array_like*) – Normal vector of the plane [x, y, z].
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

Position (*ndarray*) in space which the line intersects the plane and the distance the intersect occurs from the origin (*float*). *None* is returned if the line does not intersect the plane at a single point or at all.

Return type

tuple or None

Examples

Find the point in the scene a ray intersects the plane:

```
# plane information
planeOrigin = [0, 0, 0]
planeNormal = [0, 0, 1]
planeUpAxis = perp([0, 1, 0], planeNormal)

# ray
rayDir = [0, 0, -1]
rayOrigin = [0, 0, 5]

# get the intersect and distance in 3D world space
pnt, dist = intersectRayPlane(rayOrigin, rayDir, planeOrigin, planeNormal)
```

psychopy.tools.mathtools.intersectRaySphere

psychopy.tools.mathtools.**intersectRaySphere**(*rayOrig*, *rayDir*, *sphereOrig=(0.0, 0.0, 0.0)*, *sphereRadius=1.0*, *dtype=None*)

Calculate the points which a ray/line intersects a sphere (if any).

Get the 3D coordinate of the point which the ray intersects the sphere and the distance to the point from *orig*. The nearest point is returned if the line intersects the sphere at multiple locations. All coordinates should be in world/scene units.

Parameters

- **rayOrig** (*array_like*) – Origin of the ray in space [x, y, z].
- **rayDir** (*array_like*) – Direction vector of the ray [x, y, z], should be normalized.
- **sphereOrig** (*array_like*) – Origin of the sphere to test [x, y, z].
- **sphereRadius** (*float*) – Sphere radius to test in scene units.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

Coordinate in world space of the intersection and distance in scene units from *orig*. Returns *None* if there is no intersection.

Return type

tuple

psychopy.tools.mathtools.intersectRayAABB

`psychopy.tools.mathtools.`**`intersectRayAABB`**(*rayOrig*, *rayDir*, *boundsOffset*, *boundsExtents*, *dtype=None*)

Find the point a ray intersects an axis-aligned bounding box (AABB).

Parameters

- **rayOrig** (*array_like*) – Origin of the ray in space [x, y, z].
- **rayDir** (*array_like*) – Direction vector of the ray [x, y, z], should be normalized.
- **boundsOffset** (*array_like*) – Offset of the bounding box in the scene [x, y, z].
- **boundsExtents** (*array_like*) – Minimum and maximum extents of the bounding box.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

Coordinate in world space of the intersection and distance in scene units from *rayOrig*. Returns *None* if there is no intersection.

Return type

tuple

Examples

Get the point on an axis-aligned bounding box that the cursor is over and place a 3D stimulus there. The eye location is defined by *RigidBodyPose* object *camera*:

```
# get the mouse position on-screen
mx, my = mouse.getPos()

# find the point which the ray intersects on the box
result = intersectRayAABB(
    camera.pos,
    camera.transformNormal(win.coordToRay((mx, my))),
    myStim.pos,
    myStim.thePose.bounds.extents)
```

(continues on next page)

(continued from previous page)

```
# if the ray intersects, set the position of the cursor object to it
if result is not None:
    cursorModel.thePose.pos = result[0]
    cursorModel.draw()  # don't draw anything if there is no intersect
```

Note that if the model is rotated, the bounding box may not be aligned anymore with the axes. Use *intersectRayOBB* if your model rotates.

psychopy.tools.mathtools.intersectRayOBB

`psychopy.tools.mathtools.`**`intersectRayOBB`**`(`*rayOrig*, *rayDir*, *modelMatrix*, *boundsExtents*, *dtype=None*`)`

Find the point a ray intersects an oriented bounding box (OBB).

Parameters

- **`rayOrig`** (*`array_like`*) – Origin of the ray in space [x, y, z].
- **`rayDir`** (*`array_like`*) – Direction vector of the ray [x, y, z], should be normalized.
- **`modelMatrix`** (*`array_like`*) – 4x4 model matrix of the object and bounding box.
- **`boundsExtents`** (*`array_like`*) – Minimum and maximum extents of the bounding box.
- **`dtype`** (*`dtype or str, optional`*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

Coordinate in world space of the intersection and distance in scene units from *rayOrig*. Returns *None* if there is no intersection.

Return type

tuple

Examples

Get the point on an oriented bounding box that the cursor is over and place a 3D stimulus there. The eye location is defined by *RigidBodyPose* object *camera*:

```
# get the mouse position on-screen
mx, my = mouse.getPos()

# find the point which the ray intersects on the box
result = intersectRayOBB(
    camera.pos,
    camera.transformNormal(win.coordToRay((mx, my))),
    myStim.thePose.getModelMatrix(),
    myStim.thePose.bounds.extents)

# if the ray intersects, set the position of the cursor object to it
if result is not None:
    cursorModel.thePose.pos = result[0]
    cursorModel.draw()  # don't draw anything if there is no intersect
```

psychopy.tools.mathtools.intersectRayTriangle

psychopy.tools.mathtools.**intersectRayTriangle**(*rayOrig*, *rayDir*, *tri*, *dtype=None*)

Get the intersection of a ray and triangle(s).

This function can be used to achieve 'pixel-perfect' ray picking/casting on meshes defined with triangles. However, high-poly meshes may lead to performance issues.

Parameters

- **rayOrig** (*array_like*) – Origin of the ray in space [x, y, z].
- **rayDir** (*array_like*) – Direction vector of the ray [x, y, z], should be normalized.
- **tri** (*array_like*) – Triangle vertices as 2D (3x3) array [p0, p1, p2] where each vertex is a length 3 array [vx, xy, vz]. The input array can be 3D (Nx3x3) to specify multiple triangles.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

Coordinate in world space of the intersection, distance in scene units from *rayOrig*, and the barycentric coordinates on the triangle [x, y]. Returns *None* if there is no intersection.

Return type

tuple

Distortion

Functions for generating barrel/pincushion distortion meshes to correct image distortion. Such distortion is usually introduced by lenses in the optical path between the viewer and the display.

lensCorrection(xys[, coefK, distCenter, ...])	Lens correction (or distortion) using the division model with even polynomial terms.
lensCorrectionSpherical(xys[, coefK, ...])	Simple lens correction.

psychopy.tools.mathtools.lensCorrection

psychopy.tools.mathtools.**lensCorrection**(*xys*, *coefK=(1.0,)*, *distCenter=(0.0, 0.0)*, *out=None*, *dtype=None*)

Lens correction (or distortion) using the division model with even polynomial terms.

Calculate new vertex positions or texture coordinates to apply radial warping, such as 'pincushion' and 'barrel' distortion. This is to compensate for optical distortion introduced by lenses placed in the optical path of the viewer and the display (such as in an HMD).

See references[1]_ for implementation details.

Parameters

- **xys** (*array_like*) – Nx2 list of vertex positions or texture coordinates to distort. Works correctly only if input values range between -1.0 and 1.0.
- **coefK** (*array_like or float*) – Distortion coefficients K_n. Specifying multiple values will add more polynomial terms to the distortion formula. Positive values will produce 'barrel' distortion, whereas negative will produce 'pincushion' distortion. In most cases, two or three coefficients are adequate, depending on the degree of distortion.

- **distCenter** (*array_like, optional*) – X and Y coordinate of the distortion center (eg. (0.2, -0.4)).
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

Array of distorted vertices.

Return type

ndarray

Notes

- At this time tangential distortion (i.e. due to a slant in the display) cannot be corrected for.

References

Examples

Creating a lens correction mesh with barrel distortion (eg. for HMDs):

```
vertices, textureCoords, normals, faces = gltools.createMeshGrid(
    subdiv=11, tessMode='center')

# recompute vertex positions
vertices[:, :2] = mt.lensCorrection(vertices[:, :2], coefK=(5., 5.))
```

psychopy.tools.mathtools.lensCorrectionSpherical

psychopy.tools.mathtools.**lensCorrectionSpherical**(*xys*, *coefK=1.0*, *aspect=1.0*, *out=None*, *dtype=None*)

Simple lens correction.

Lens correction for a spherical lenses with distortion centered at the middle of the display. See references[1]_ for implementation details.

Parameters

- **xys** (*array_like*) – Nx2 list of vertex positions or texture coordinates to distort. Assumes the output will be rendered to normalized device coordinates where points range from -1.0 to 1.0.
- **coefK** (*float*) – Distortion coefficient. Use positive numbers for pincushion distortion and negative for barrel distortion.
- **aspect** (*float*) – Aspect ratio of the target window or buffer (width / height).
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns

Array of distorted vertices.

Return type
ndarray

References

Examples

Creating a lens correction mesh with barrel distortion (eg. for HMDs):

```
vertices, textureCoords, normals, faces = gltools.createMeshGrid(
    subdiv=11, tessMode='center')

# recompute vertex positions
vertices[:, :2] = mt.lensCorrection2(vertices[:, :2], coefK=2.0)
```

Miscellaneous

Miscellaneous and helper functions.

zeroFix(a[, inplace, threshold])	Fix zeros in an array.

psychopy.tools.mathtools.zeroFix

psychopy.tools.mathtools.**zeroFix**(*a*, *inplace=False*, *threshold=None*)

Fix zeros in an array.

This function truncates very small numbers in an array to zero and removes any negative zeros.

Parameters

- **a** (*ndarray*) – Input array, must be a Numpy array.
- **inplace** (*bool*) – Fix an array inplace. If *True*, the input array will be modified, otherwise a new array will be returned with same *dtype* and shape with the fixed values.
- **threshold** (*float or None*) – Threshold for truncation. If *None*, the machine epsilon value for the input array *dtype* will be used. You can specify a custom threshold as a float.

Returns
Output array with zeros fixed.

Return type
ndarray

Performance and Optimization

Most functions listed here are very fast, however they are optimized to work on arrays of values (vectorization). Calling functions repeatedly (for instance within a loop), should be avoided as the CPU overhead associated with each function call (not to mention the loop itself) can be considerable.

For example, one may want to normalize a bunch of randomly generated vectors by calling *normalize()* on each row:

```
v = np.random.uniform(-1.0, 1.0, (1000, 4,))  # 1000 length 4 vectors
vn = np.zeros((1000, 4))  # place to write values

# don't do this!
```

(continues on next page)

(continued from previous page)

```
for i in range(1000):
    vn[i, :] = normalize(v[i, :])
```

The same operation is completed in considerably less time by passing the whole array to the function like so:

```
normalize(v, out=vn)  # very fast!
vn = normalize(v)  # also fast if `out` is not provided
```

Specifying an output array to *out* will improve performance by reducing overhead associated with allocating memory to store the result (functions do this automatically if *out* is not provided). However, *out* should only be provided if the output array is reused multiple times. Furthermore, the function still returns a value if *out* is provided, but the returned value is a reference to *out*, not a copy of it. If *out* is not provided, the function will return the result with a freshly allocated array.

Data Types

Sub-routines used by the functions here will perform arithmetic using 64-bit floating-point precision unless otherwise specified via the *dtype* argument. This functionality is helpful in certain applications where input and output arrays demand a specific type (eg. when working with data passed to and from OpenGL functions).

If a *dtype* is specified, input arguments will be coerced to match that type and all floating-point arithmetic will use the precision of the type. If input arrays have the same type as *dtype*, they will automatically pass-through without being recast as a different type. As a performance consideration, all input arguments should have matching types and *dtype* set accordingly.

Most functions have an *out* argument, where one can specify an array to write values to. The value of *dtype* is ignored if *out* is provided, and all input arrays will be converted to match the *dtype* of *out* (if not already). This ensures that the type of the destination array is used for all arithmetic.

11.8.7 psychopy.tools.monitorunittools

Functions and classes related to unit conversion respective to a particular monitor

convertToPix(vertices, pos, units, win)	Takes vertices and position, combines and converts to pixels from any unit
cm2deg(cm, monitor[, correctFlat])	Convert size in cm to size in degrees for a given Monitor object
cm2pix(cm, monitor)	Convert size in cm to size in pixels for a given Monitor object.
deg2cm(degrees, monitor[, correctFlat])	Convert size in degrees to size in pixels for a given Monitor object.
deg2pix(degrees, monitor[, correctFlat])	Convert size in degrees to size in pixels for a given Monitor object
pix2cm(pixels, monitor)	Convert size in pixels to size in cm for a given Monitor object
pix2deg(pixels, monitor[, correctFlat])	Convert size in pixels to size in degrees for a given Monitor object

Function details

psychopy.tools.monitorunittools.**convertToPix**(*vertices*, *pos*, *units*, *win*)

Takes vertices and position, combines and converts to pixels from any unit

The reason that *pos* and *vertices* are provided separately is that it allows the conversion from deg to apply flat-screen correction to each separately.

The reason that these use function args rather than relying on self.pos is that some stimuli use other terms (e.g. ElementArrayStim uses fieldPos).

`psychopy.tools.monitorunittools.`**`cm2deg`**(*cm*, *monitor*, *correctFlat=False*)

Convert size in cm to size in degrees for a given Monitor object

`psychopy.tools.monitorunittools.`**`cm2pix`**(*cm*, *monitor*)

Convert size in cm to size in pixels for a given Monitor object.

`psychopy.tools.monitorunittools.`**`deg2cm`**(*degrees*, *monitor*, *correctFlat=False*)

Convert size in degrees to size in pixels for a given Monitor object.

If *correctFlat == False* then the screen will be treated as if all points are equal distance from the eye. This means that each "degree" will be the same size irrespective of its position.

If *correctFlat == True* then the *degrees* argument must be an Nx2 matrix for X and Y values (the two cannot be calculated separately in this case).

With *correctFlat == True* the positions may look strange because more eccentric vertices will be spaced further apart.

`psychopy.tools.monitorunittools.`**`deg2pix`**(*degrees*, *monitor*, *correctFlat=False*)

Convert size in degrees to size in pixels for a given Monitor object

`psychopy.tools.monitorunittools.`**`pix2cm`**(*pixels*, *monitor*)

Convert size in pixels to size in cm for a given Monitor object

`psychopy.tools.monitorunittools.`**`pix2deg`**(*pixels*, *monitor*, *correctFlat=False*)

Convert size in pixels to size in degrees for a given Monitor object

11.8.8 `psychopy.tools.movietools`

Classes and functions for working with movies in PsychoPy.

Overview

MovieFileWriter(filename, size, fps[, ...])	Create movies from a sequence of images.
MovieFileWriter.filename	The name (path) of the movie file (*str*).
MovieFileWriter.fps	Output frames per second (*float*).
MovieFileWriter.size	The size *(w, h)* of the movie in pixels (*tuple* or *str*).
MovieFileWriter.codec	The codec to use for encoding the movie (*str*).
MovieFileWriter.pixelFormat	Pixel format for frames being added to the movie (*str*).
MovieFileWriter.encoderLib	The library to use for writing the movie (*str*).
MovieFileWriter.encoderOpts	Encoder options (*dict*).
MovieFileWriter.frameRate	Output frames per second (*float*).
MovieFileWriter.frameSize	The size *(w, h)* of the movie in pixels (*tuple*).
MovieFileWriter.lastVideoFile	The name of the last video file written to disk (*str* or *None*).
MovieFileWriter.isOpen	Whether the movie file is open (*bool*).
MovieFileWriter.framesOut	Total number of frames written to the movie file (*int*).
MovieFileWriter.bytesOut	Total number of bytes (*int*) saved to the movie file.
MovieFileWriter.framesWaiting	The number of frames waiting to be written to disk (*int*).
MovieFileWriter.totalFrames	The total number of frames that will be written to the movie file (*int*).
MovieFileWriter.frameInterval	The time interval between frames (*float*).
MovieFileWriter.duration	The duration of the movie in seconds (*float*).
MovieFileWriter.open()	Open the movie file for writing.
MovieFileWriter.flush()	Flush waiting frames to the movie file.
MovieFileWriter.close()	Close the movie file.
MovieFileWriter.addFrame(image[, pts])	Add a frame to the movie.
closeAllMovieWriters()	Signal all movie writers to close.
addAudioToMovie(outputFile, videoFile, audioFile)	Add an audio track to a video file.

Details

class psychopy.tools.movietools.**MovieFileWriter**(*filename*, *size*, *fps*, *codec=None*, *pixelFormat='rgb24'*, *encoderLib='ffpyplayer'*, *encoderOpts=None*)

Create movies from a sequence of images.

This class allows for the creation of movies from a sequence of images using FFMPEG (via the *ffpyplayer* or *cv2* libraries). Writing movies to disk is a slow process, so this class uses a separate thread to write the movie in the background. This means that you can continue to add images to the movie while frames are still being written to disk. Movie writers are closed automatically when the main thread exits. Any remaining frames are flushed to the file before the file is finalized.

Writing audio tracks is not supported. If you need to add audio to your movie, create the file with the video content first, then add the audio track to the file. The `addAudioToMovie()` function can be used to do this after the video and audio files have been saved to disk.

Parameters

- **filename** (*str*) – The name (or path) of the file to write the movie to. The file extension determines the movie format if *codec* is *None* for some backends. Otherwise it must be explicitly specified.
- **size** (*tuple or str*) – The size of the movie in pixels (width, height). If a string is passed, it should be one of the keys in the *VIDEO_RESOLUTIONS* dictionary.
- **fps** (*float*) – The number of frames per second.

- **codec** (*str or None*) – The codec to use for encoding the movie. This may be a codec identifier (e.g., 'libx264') or a FourCC code. The value depends of the *encoderLib* in use. If *None*, the writer will select the codec based on the file extension of *filename* (if supported by the backend).
- **pixelFormat** (*str*) – Pixel format for frames being added to the movie. This should be either 'rgb24' or 'rgba32'. The default is 'rgb24'. When passing frames to *addFrame()* as a numpy array, the array should be in the format specified here.
- **encoderLib** (*str*) – The library to use to handle encoding and writing the movie to disk. The default is 'ffpyplayer'.
- **encoderOpts** (*dict or None*) – A dictionary of options to pass to the encoder. These option can be used to control the quality of the movie, for example. The options depend on the *encoderLib* in use. If *None*, the writer will use the default options for the backend.

Examples

Create a movie from a sequence of generated noise images:

```
import psychopy.tools.movietools as movietools
import numpy as np

# create a movie writer
writer = movietools.MovieFileWriter(
    filename='myMovie.mp4',
    size=(640, 480),
    fps=30)

# open the movie for writing
writer.open()

# add some frames to the movie
for i in range(5 * writer.fps):  # 5 seconds of video
    # create a frame, just some random noise
    frame = np.random.randint(0, 255, (640, 480, 3), dtype=np.uint8)
    # add the frame to the movie
    writer.addFrame(frame)

# close the movie, this completes the writing process
writer.close()
```

Setting additional options for the movie encoder requires passing a dictionary of options to the *encoderOpts* parameter. The options depend on the encoder library in use. For example, to set the quality of the movie when using the *ffpyplayer* library, you can do the following:

```
ffmpegOpts = {'preset': 'medium', 'crf': 16}  # medium quality, crf=16
writer = movietools.MovieFileWriter(
    filename='myMovie.mp4',
    size='720p',
    fps=30,
    encoderLib='ffpyplayer',
    encoderOpts=ffmpegOpts)
```

The OpenCV backend specifies options differently. To set the quality of the movie when using the OpenCV library with a codec that support variable quality, you can do the following:

```
cvOpts = {'quality': 80}  # set the quality to 80 (0-100)
writer = movietools.MovieFileWriter(
    filename='myMovie.mp4',
    size='720p',
    fps=30,
    encoderLib='opencv',
    encoderOpts=cvOpts)
```

PIXEL_FORMAT_RGB24 = 'rgb24'

PIXEL_FORMAT_RGBA32 = 'rgb32'

_convertImage(*image*)

Convert an image to a pixel format appropriate for the encoder.

This is used internally to convert an image (i.e. frame) to the native frame format which the encoder library can work with. At the very least, this function should accept a *numpy.array* as a valid type for *image* no matter what encoder library is being used.

Parameters
image (*Any*) – The image to convert.

Returns
The converted image. Resulting object type depends on the encoder library being used.

Return type
Any

_openFFPyPlayer()

Open a movie writer using FFPyPlayer.

This is called by *open()* if *encoderLib* is 'ffpyplayer'. It will create a background thread to write the movie file. This method is not intended to be called directly.

_openOpenCV()

Open a movie writer using OpenCV.

This is called by *open()* if *encoderLib* is 'opencv'. It will create a background thread to write the movie file. This method is not intended to be called directly.

addFrame(*image*, *pts=None*)

Add a frame to the movie.

This adds a frame to the movie. The frame will be added to a queue and written to disk by a background thread. This method will block until the frame is added to the queue.

Any color space conversion or resizing will be performed in the caller's thread. This may be threaded too in the future.

Parameters

- **image** (*numpy.ndarray or ffpyplayer.pic.Image*) – The image to add to the movie. The image must be in RGB format and have the same size as the movie. If the image is an *Image* instance, it must have the same size as the movie.
- **pts** (*float or None*) – The presentation timestamp for the frame. This is the time at which the frame should be displayed. The presentation timestamp is in seconds and should be monotonically increasing. If *None*, the presentation timestamp will be automatically generated based on the chosen frame rate for the output video. Not all encoder libraries support presentation timestamps, so this parameter may be ignored.

Returns
Presentation timestamp assigned to the frame. Should match the value passed in as *pts* if provided, otherwise it will be the computed presentation timestamp.

Return type
float

`property bytesOut`

Total number of bytes (*int*) saved to the movie file.

Use this to monitor how much disk space is occupied by the frames that have been written so far. This value is updated asynchronously, so it may not be accurate if you are adding frames to the movie file very quickly.

This value is retained after the movie file is closed. It is cleared when a new movie file is opened.

`close()`

Close the movie file.

This shuts down the background thread and finalizes the movie file. Any frames still waiting in the queue will be written to disk before the movie file is closed. This will block the program until all frames are written, therefore, it is recommended for *close()* to be called outside any time-critical code.

`property codec`

The codec to use for encoding the movie (*str*).

This may be a codec identifier (e.g., 'libx264'), or a FourCC code (e.g. 'MPV4'). The value depends of the *encoderLib* in use. If *None*, the a codec determined by the file extension will be used.

`property duration`

The duration of the movie in seconds (*float*).

This is the total duration of the movie in seconds based on the number of frames that have been added to the movie and the frame rate. This does not represent the actual duration of the movie file on disk, which may be longer if frames are still being written to disk.

`property encoderLib`

The library to use for writing the movie (*str*).

Can only be set before the movie file is opened. The default is 'ffpyplayer'.

`property encoderOpts`

Encoder options (*dict*).

These are passed directly to the encoder library. The default is an empty dictionary.

`property filename`

The name (path) of the movie file (*str*).

This cannot be changed after the writer has been opened.

`flush()`

Flush waiting frames to the movie file.

This will cause all frames waiting in the queue to be written to disk before continuing the program i.e. the thread that called this method. This is useful for ensuring that all frames are written to disk before the program exits.

`property fps`

Output frames per second (*float*).

This is the number of frames per second that will be written to the movie file. The default is 30.

`property frameInterval`

The time interval between frames (*float*).

This is the time interval between frames in seconds. This is the reciprocal of the frame rate.

`property frameRate`

Output frames per second (*float*).

This is an alias for *fps* to synchronize naming with other video classes around PsychoPy.

`property frameSize`

The size *(w, h)* of the movie in pixels (*tuple*).

This is an alias for *size* to synchronize naming with other video classes around PsychoPy.

`property framesOut`

Total number of frames written to the movie file (*int*).

Use this to monitor the progress of the movie file writing. This value is updated asynchronously, so it may not be accurate if you are adding frames to the movie file very quickly.

This value is retained after the movie file is closed. It is cleared when a new movie file is opened.

`property framesWaiting`

The number of frames waiting to be written to disk (*int*).

This value increases when you call *addFrame()* and decreases when the frame is written to disk. This number can be reduced to zero by calling *flush()*.

`property isOpen`

Whether the movie file is open (*bool*).

If *True*, the movie file is open and frames can be added to it. If *False*, the movie file is closed and no more frames can be added to it.

`property lastVideoFile`

The name of the last video file written to disk (*str* or *None*).

This is *None* if no video file has been written to disk yet. Only valid after the movie file has been closed (i.e. after calling *close()*.)

`open()`

Open the movie file for writing.

This creates a new thread that will write the movie file to disk in the background.

After calling this method, you can add frames to the movie using *addFrame()*. When you are done adding frames, call *close()* to finalize the movie file.

`property pixelFormat`

Pixel format for frames being added to the movie (*str*).

This should be either 'rgb24' or 'rgba32'. The default is 'rgb24'. When passing frames to *addFrame()* as a numpy array, the array should be in the format specified here.

`property size`

The size *(w, h)* of the movie in pixels (*tuple* or *str*).

If a string is passed, it should be one of the keys in the *VIDEO_RESOLUTIONS* dictionary.

This can not be changed after the writer has been opened.

property totalFrames

The total number of frames that will be written to the movie file (*int*).

This incudes frames that have already been written to disk and frames that are waiting to be written to disk.

psychopy.tools.movietools.**closeAllMovieWriters**()

Signal all movie writers to close.

This function should only be called once at the end of the program. This can be registered *atexit* to ensure that all movie writers are closed when the program exits. If there are open file writers with frames still queued, this function will block until all frames remaining are written to disk.

Use caution when calling this function when file writers are being used in a multi-threaded environment. Threads that are writing movie frames must be stopped prior to calling this function. If not, the thread may continue to write frames to the queue during the flush operation and never exit.

psychopy.tools.movietools.**addAudioToMovie**(*outputFile*, *videoFile*, *audioFile*, *useThreads=True*, *removeFiles=False*, *writerOpts=None*)

Add an audio track to a video file.

This function will add an audio track to a video file. If the video file already has an audio track, it will be replaced with the audio file provided. If no audio file is provided, the audio track will be removed from the video file.

The audio track should be exactly the same length as the video track.

Parameters

- **outputFile** (*str*) – Path to the output video file where audio and video will be merged.
- **videoFile** (*str*) – Path to the input video file.
- **audioFile** (*str or None*) – Path to the audio file to add to the video file.
- **codec** (*str*) – The name of the audio codec to use. This should be a valid codec name for the encoder library being used. If *None*, the default codec for the encoder library will be used.
- **useThreads** (*bool*) – If *True*, the audio will be added in a separate thread. This allows the audio to be added in the background while the program continues to run. If *False*, the audio will be added in the main thread and the program will block until the audio is added. Defaults to *True*.
- **removeFiles** (*bool*) – If *True*, the input video (*videoFile*) and audio (*audioFile*) files will be removed (i.e. deleted from disk) after the audio has been added to the video. Defaults to *False*.
- **writerOpts** (*dict or None*) – Options to pass to the movie writer. This should be a dictionary of keyword arguments to pass to the movie writer. If *None*, the default options for the movie writer will be used. Defaults to *None*. See documentation for *moviepy.video.io.VideoFileClip.write_videofile* for possible values.

Examples

Combine a video file and an audio file into a single video file:

```
from psychopy.tools.movietools import addAudioToMovie
addAudioToMovie('output.mp4', 'video.mp4', 'audio.mp3')
```

11.8.9 psychopy.tools.pkgtools

Tools for working with packages within the Python environment.

getUserPackagesPath()	Get the path to the user's PsychoPy package directory.
getDistributions()	Get a list of active distributions in the current environment.
addDistribution(distPath)	Add a distribution to the current environment.
installPackage(package[, target, upgrade, ...])	Install a package using the default package management system.
uninstallPackage(package)	Uninstall a package from the current distribution.
getInstalledPackages()	Get a list of installed packages and their versions.
getPackageMetadata(packageName)	Get the metadata for a specified package.
getPypiInfo(packageName[, silence])	

Details

psychopy.tools.pkgtools.**getUserPackagesPath**()

Get the path to the user's PsychoPy package directory.

This is the directory that plugin and extension packages are installed to which is added to *sys.path* when *psychopy* is imported.

Returns
Path to user's package directory.

Return type
str

psychopy.tools.pkgtools.**getDistributions**()

Get a list of active distributions in the current environment.

Returns
List of paths where active distributions are located. These paths refer to locations where packages containing importable modules and plugins can be found.

Return type
list

psychopy.tools.pkgtools.**addDistribution**(*distPath*)

Add a distribution to the current environment.

This function can be used to add a distribution to the present environment which contains Python packages that have importable modules or plugins.

Parameters
distPath (*str*) – Path to distribution. May be either a path for a directory or archive file (e.g. ZIP).

psychopy.tools.pkgtools.**installPackage**(*package*, *target=None*, *upgrade=False*, *forceReinstall=False*, *noDeps=False*, *awaited=True*, *outputCallback=None*, *terminateCallback=None*, *extra=None*)

Install a package using the default package management system.

This is intended to be used only by PsychoPy itself for installing plugins and packages through the builtin package manager.

Parameters

- **package** (*str*) – Package name (e.g., *'psychopy-connect'*, *'scipy'*, etc.) with version if needed. You may also specify URLs to Git repositories and such.
- **target** (*str or None*) – Location to install packages to directly to. If *None*, the user's package directory is set at the prefix and the package is installed there. If a *target* is specified, the package top-level directory must be added to *sys.path* manually.
- **upgrade** (*bool*) – Upgrade the specified package to the newest available version.
- **forceReinstall** (*bool*) – If *True*, the package and all it's dependencies will be reinstalled if they are present in the current distribution.
- **noDeps** (*bool*) – Don't install dependencies if *True*.
- **awaited** (*bool*) – If False, then use an asynchronous install process - this function will return right away and the plugin install will happen in a different thread.
- **outputCallback** (*function*) – Function to be called when any output text is received from the process performing the install. Not used if awaited=True.
- **terminateCallback** (*function*) – Function to be called when installation is finished. Not used if awaited=True.
- **extra** (*dict*) – Extra information to be supplied to the install thread when installing asynchronously. Not used if awaited=True.

Returns

If *awaited=True*:
True if the package installed without errors. If *False*, check 'stderr' for more information. The package may still have installed correctly, but it doesn't work. Second value contains standard output and error from the subprocess.

If *awaited=False*:
Returns the job (thread) which is running the install.

Return type
tuple or psychopy.app.jobs.Job

psychopy.tools.pkgtools.**uninstallPackage**(*package*)

Uninstall a package from the current distribution.

Parameters
package (*str*) – Package name (e.g., *'psychopy-connect'*, *'scipy'*, etc.) with version if needed. You may also specify URLs to Git repositories and such.

Returns
True if the package removed without errors. If *False*, check 'stderr' for more information. The package may still have uninstalled correctly, but some other issues may have arose during the process.

Return type
tuple

Notes

- The *–yes* flag is appended to the pip command. No confirmation will be requested if the package already exists.

psychopy.tools.pkgtools.**getInstalledPackages**()

Get a list of installed packages and their versions.

Returns
List of installed packages and their versions i.e. *('PsychoPy', '2021.3.1')*.

Return type
list

`psychopy.tools.pkgtools.`**`getPackageMetadata`**`(`*packageName*`)`

Get the metadata for a specified package.

Parameters
`packageName` (*`str`*) – Project name of package to get metadata from.

Returns
Dictionary of metadata fields. If *None* is returned, the package isn't present in the current distribution.

Return type
dict or None

`psychopy.tools.pkgtools.`**`getPypiInfo`**`(`*packageName*, *silence=False*`)`

11.8.10 `psychopy.tools.plottools`

Functions and classes related to plotting

`psychopy.tools.plottools.`**`plotFrameIntervals`**`(`*intervals*`)`

Plot a histogram of the frame intervals.

Where *intervals* is either a filename to a file, saved by Window.saveFrameIntervals, or simply a list (or array) of frame intervals

11.8.11 `psychopy.tools.rifttools`

Various tools for working with the *`Rift`* class. The documentation for classes in on this page originate from PsychXR and may make references to functions and objects not included with .

Overview

Classes

These classes are included with PsychXR to use with the LibOVR interface. They can be accessed from this module to avoid needing to explicitly import PsychXR. If PsychXR is not available on the system, these classes will have values *None*.

`LibOVRPose`
`LibOVRPoseState`
`LibOVRHapticsBuffer`
`LibOVRBounds`

Functions

These functions can be called without first starting a VR session (initializing a `Rift` instance) to check if the drivers/services are running on this computer or if an HMD is connected.

`isHmdConnected`([timeout])	Check if an HMD is connected.
`isOculusServiceRunning`([timeout])	Check if the Oculus(tm) service is currently running.

Details

`psychopy.tools.rifttools.`**`LibOVRPose`**

alias of `None`

`psychopy.tools.rifttools.`**`LibOVRPoseState`**

alias of `None`

`psychopy.tools.rifttools.`**`LibOVRBounds`**

alias of `None`

`psychopy.tools.rifttools.`**`LibOVRHapticsBuffer`**

alias of `None`

`psychopy.tools.rifttools.`**`isHmdConnected`**(*timeout=0*)

Check if an HMD is connected.

Parameters
`timeout` (*`int`*) – Timeout in milliseconds.

Returns
True if an HMD is connected.

Return type
bool

`psychopy.tools.rifttools.`**`isOculusServiceRunning`**(*timeout=0*)

Check if the Oculus(tm) service is currently running.

Parameters
`timeout` (*`int`*) – Timeout in milliseconds.

Returns
True if the service is loaded and running.

Return type
bool

11.8.12 `psychopy.tools.systemtools`

Tools for interacting with the system PsychoPy is installed on. This involves things such as getting information about installed devices and software.

Overview

getAudioDevices()	Get all audio devices.
getAudioCaptureDevices()	Get audio capture devices (i.e. microphones) installed on the system.
getAudioPlaybackDevices()	Get audio playback devices (i.e. speakers) installed on the system.
getCameras()	Get information about installed cameras and their formats on this system.
getKeyboards()	Get information about attached keyboards.
getSerialPorts()	Get serial ports attached to this system.
systemProfilerMacOS([dataTypes, ...])	Call the MacOS system profiler and return data in a JSON format.

Details

psychopy.tools.systemtools.**getAudioDevices**()

Get all audio devices.

This function gets all audio devices attached to the system, either playback or capture. Uses the *psychtoolbox* library to obtain the relevant information.

This command is supported on Windows, MacOSX and Linux. On Windows, WASAPI devices are preferred to achieve precise timing and will be returned by default. To get all audio devices (including non-WASAPI ones), set the preference *audioForceWASAPI* to *False*.

Returns

Dictionary where the keys are devices names and values are mappings whose fields contain information about the device.

Return type

dict

Examples

Get audio devices installed on this system:

```
allDevs = getAudioDevices()
```

The following dictionary is returned by the above command when called on an Apple MacBook Pro (2022):

```
{
    'MacBook Pro Microphone': {   # audio capture device
        'index': 0,
        'name': 'MacBook Pro Microphone',
        'hostAPI': 'Core Audio',
        'outputChannels': 0,
        'outputLatency': (0.01, 0.1),
        'inputChannels': 1,
        'inputLatency': (0.0326984126984127, 0.04285714285714286),
        'defaultSampleRate': 44100.0,
        'audioLib': 'ptb'
    },
    'MacBook Pro Speakers': {    # audio playback device
        'index': 1,
```

(continues on next page)

(continued from previous page)

```
        'name': 'MacBook Pro Speakers',
        'hostAPI': 'Core Audio',
        'outputChannels': 2,
        'outputLatency': (0.008480725623582767, 0.018639455782312925),
        'inputChannels': 0,
        'inputLatency': (0.01, 0.1),
        'defaultSampleRate': 44100.0,
        'audioLib': 'ptb'
    }
}
```

To determine whether something is a playback or capture device, check the number of output and input channels, respectively:

```
# determine if a device is for audio capture
isCapture = allDevs['MacBook Pro Microphone']['inputChannels'] > 0

# determine if a device is for audio playback
isPlayback = allDevs['MacBook Pro Microphone']['outputChannels'] > 0
```

You may also call `getAudioCaptureDevices()` and `getAudioPlaybackDevices()` to get just audio capture and playback devices.

`psychopy.tools.systemtools.`**`getAudioCaptureDevices`**`()`

Get audio capture devices (i.e. microphones) installed on the system.

This command is supported on Windows, MacOSX and Linux. On Windows, WASAPI devices are preferred to achieve precise timing and will be returned by default. To get all audio capture devices (including non-WASAPI ones), set the preference *audioForceWASAPI* to *False*.

Uses the *psychtoolbox* library to obtain the relevant information.

Returns

Dictionary where the keys are devices names and values are mappings whose fields contain information about the capture device. See `getAudioDevices()` examples to see the format of the output.

Return type

dict

`psychopy.tools.systemtools.`**`getAudioPlaybackDevices`**`()`

Get audio playback devices (i.e. speakers) installed on the system.

This command is supported on Windows, MacOSX and Linux. On Windows, WASAPI devices are preferred to achieve precise timing and will be returned by default. To get all audio playback devices (including non-WASAPI ones), set the preference *audioForceWASAPI* to *False*.

Uses the *psychtoolbox* library to obtain the relevant information.

Returns

Dictionary where the keys are devices names and values are mappings whose fields contain information about the playback device. See `getAudioDevices()` examples to see the format of the output.

Return type

dict

psychopy.tools.systemtools.**getCameras**()

Get information about installed cameras and their formats on this system.

The command presently only works on Window and MacOSX. Linux support for cameras is not available yet.

Returns

Mapping where camera names (*str*) are keys and values are and array of *CameraInfo* objects.

Return type

dict

psychopy.tools.systemtools.**getKeyboards**()

Get information about attached keyboards.

This command works on Windows, MacOSX and Linux.

Returns

Dictionary where the keys are device names and values are mappings whose fields contain information about that device. See the *Examples* section for field names.

Return type

dict

Notes

- Keyboard names are generated (taking the form of "Generic Keyboard n") if the OS does not report the name.

Examples

Get keyboards attached to this system:

```
installedKeyboards = getKeyboards()
```

Running the previous command on an Apple MacBook Pro (2022) returns the following dictionary:

```
{
    'TouchBarUserDevice': {
        'usagePageValue': 1,
        'usageValue': 6,
        'usageName': 'Keyboard',
        'index': 4,
        'transport': '',
        'vendorID': 1452,
        'productID': 34304,
        'version': 0.0,
        'manufacturer': '',
        'product': 'TouchBarUserDevice',
        'serialNumber': '',
        'locationID': 0,
        'interfaceID': -1,
        'totalElements': 1046,
        'features': 0,
        'inputs': 1046,
        'outputs': 0,
        'collections': 1,
        'axes': 0,
```

(continues on next page)

(continued from previous page)

```
        'buttons': 0,
        'hats': 0,
        'sliders': 0,
        'dials': 0,
        'wheels': 0,
        'touchDeviceType': -1,
        'maxTouchpoints': -1},
    'Generic Keyboard 0': {
        'usagePageValue': 1,
        'usageValue': 6,
        'usageName': 'Keyboard',
        'index': 13,
        # snip ...
        'dials': 0,
        'wheels': 0,
        'touchDeviceType': -1,
        'maxTouchpoints': -1
    }
}
```

psychopy.tools.systemtools.**getSerialPorts**()

Get serial ports attached to this system.

Serial ports are used for inter-device communication using the RS-232/432 protocol. This function gets a list of available ports and their default configurations as specified by the OS. Ports that are in use by another process are not returned.

This command is supported on Windows, MacOSX and Linux. On Windows, all available ports are returned regardless if anything is connected to them, so long as they aren't in use. On Unix(-likes) such as MacOSX and Linux, port are only returned if there is a device attached and is not being accessed by some other process. MacOSX and Linux also have no guarantee port names are persistent, where a physical port may not always be assigned the same name or enum index when a device is connected or after a system reboot.

Returns

Mapping (*dict*) where keys are serial port names (*str*) and values are mappings of the default settings of the port (*dict*). See *Examples* below for the format of the returned data.

Return type

dict

Examples

Getting available serial ports:

```
allPorts = getSerialPorts()
```

On a MacBook Pro (2022) with an Arduino Mega (2560) connected to the USB-C port, the following dictionary is returned:

```
{
    '/dev/cu.Bluetooth-Incoming-Port': {
        'index': 0,
        'port': '/dev/cu.Bluetooth-Incoming-Port',
        'baudrate': 9600,
```

(continues on next page)

(continued from previous page)

```
        'bytesize': 8,
        'parity': 'N',
        'stopbits': 1,
        'xonxoff': False,
        'rtscts': False,
        'dsrdtr': False
    },
    '/dev/cu.usbmodem11101': {
        'index': 1,
        # ... snip ...
        'dsrdtr': False
    },
    '/dev/tty.Bluetooth-Incoming-Port': {
        'index': 2,
        # ... snip ...
    },
    '/dev/tty.usbmodem11101': {
        'index': 3,
        # ... snip ...
    }
}
```

`psychopy.tools.systemtools.`**`systemProfilerMacOS`**(*dataTypes=None*, *detailLevel='basic'*, *timeout=180*)

Call the MacOS system profiler and return data in a JSON format.

Parameters

- **dataTypes** (*str, list or None*) – Identifier(s) for the data to retrieve. All data types available will be returned if *None*. See output of shell command *system_profiler -listDataTypes* for all possible values. Specifying data types also speeds up the time it takes for this function to return as superfluous information is not queried.
- **detailLevel** (*int or str*) – Level of detail for the report. Possible values are *'mini'*, *'basic'*, or *'full'*. Note that increasing the level of detail will expose personally identifying information in the resulting report. Best practice is to use the lowest level of detail needed to obtain the desired information, or use *dataTypes* to limit what information is returned.
- **timeout** (*float or int*) – Amount of time to spend gathering data in seconds. Default is 180 seconds, while specifying 0 means no timeout.

Returns

Result of the *system_profiler* call as a JSON formatted string. You can pass the string to a JSON library to parse out what information is desired.

Return type

str

Examples

Get details about cameras attached to this system:

```
dataTypes = "SPCameraDataType"  # data to query
systemReportJSON = systemProfilerMacOS(dataTypes, detailLevel='basic')
# >>> print(systemReportJSON)
# {
```

(continues on next page)

(continued from previous page)

```
#   "SPCameraDataType" : [
#     ...
#   ]
# }
```

Parse the result using a JSON library:

```
import json
systemReportJSON = systemProfilerMacOS(
    "SPCameraDataType", detailLevel='mini')
cameraInfo = json.loads(systemReportJSON)
# >>> print(cameraInfo)
# {'SPCameraDataType': [{'_name': 'Live! Cam Sync 1080p',
# 'spcamera_model-id': 'UVC Camera VendorID_1054 ProductID_16541',
# 'spcamera_unique-id': '0x2200000041e409d'}]
```

11.8.13 psychopy.tools.typetools

Functions and classes related to variable type conversion

psychopy.tools.typetools.**float_uint8**(*inarray*)

Converts arrays, lists, tuples and floats ranging -1:1 into an array of Uint8s ranging 0:255

```
>>> float_uint8(-1)
0
>>> float_uint8(0)
128
```

psychopy.tools.typetools.**uint8_float**(*inarray*)

Converts arrays, lists, tuples and UINTs ranging 0:255 into an array of floats ranging -1:1

```
>>> uint8_float(0)
-1.0
>>> uint8_float(128)
0.0
```

psychopy.tools.typetools.**float_uint16**(*inarray*)

Converts arrays, lists, tuples and floats ranging -1:1 into an array of Uint16s ranging 0:2^16

```
>>> float_uint16(-1)
0
>>> float_uint16(0)
32768
```

11.8.14 psychopy.tools.unittools

Functions and classes related to unit conversion

psychopy.tools.unittools.**radians**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*])

Convert angles from degrees to radians.

Parameters

- **x** (*array_like*) – Input array in degrees.
- **out** (*ndarray, None, or tuple of ndarray and None, optional*) – A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- **where** (*array_like, optional*) – This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.
- ****kwargs** – For other keyword-only arguments, see the ufunc docs.

Returns
y – The corresponding radian values. This is a scalar if *x* is a scalar.

Return type
ndarray

See also

deg2rad
equivalent function

Examples

Convert a degree array to radians

```
>>> deg = np.arange(12.) * 30.
>>> np.radians(deg)
array([ 0.        ,  0.52359878,  1.04719755,  1.57079633,  2.0943951 ,
        2.61799388,  3.14159265,  3.66519143,  4.1887902 ,  4.71238898,
        5.23598776,  5.75958653])
```

```
>>> out = np.zeros((deg.shape))
>>> ret = np.radians(deg, out)
>>> ret is out
True
```

psychopy.tools.unittools.**degrees**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*])

Convert angles from radians to degrees.

Parameters

- **x** (*array_like*) – Input array in radians.
- **out** (*ndarray, None, or tuple of ndarray and None, optional*) – A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- **where** (*array_like, optional*) – This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

- **`**kwargs`** – For other keyword-only arguments, see the ufunc docs.

Returns

y – The corresponding degree values; if *out* was supplied this is a reference to it. This is a scalar if *x* is a scalar.

Return type

ndarray of floats

See also

`rad2deg`
equivalent function

Examples

Convert a radian array to degrees

```
>>> rad = np.arange(12.)*np.pi/6
>>> np.degrees(rad)
array([   0.,   30.,   60.,   90.,  120.,  150.,  180.,  210.,  240.,
        270.,  300.,  330.])
```

```
>>> out = np.zeros((rad.shape))
>>> r = np.degrees(rad, out)
>>> np.all(r == out)
True
```

11.8.15 `psychopy.tools.viewtools`

Math functions for working with view transformations and performing visibility testing (see also *`mathtools`*). Tools for working with view projections for 2- and 3-D rendering.

Overview

visualAngle(size, distance[, degrees, out, ...])	Get the visual angle for an object of *size* at *distance*.
computeFrustum(scrWidth, scrAspect, scrDist)	Calculate frustum parameters.
computeFrustumFOV(scrFOV, scrAspect, scrDist)	Compute a frustum for a given field-of-view (FOV).
projectFrustum(frustum, dist[, dtype])	Project a frustum on a fronto-parallel plane and get the width and height of the required drawing area.
projectFrustumToPlane(frustum, planeOrig[, ...])	Project a frustum on a fronto-parallel plane and get the coordinates of the corners in physical space.
generalizedPerspectiveProjection(...[, ...])	Generalized derivation of projection and view matrices based on the physical configuration of the display system.
orthoProjectionMatrix(left, right, bottom, top)	Compute an orthographic projection matrix with provided frustum parameters.
perspectiveProjectionMatrix(left, right, ...)	Compute an perspective projection matrix with provided frustum parameters.
lookAt(eyePos, centerPos[, upVec, out, dtype])	Create a transformation matrix to orient a view towards some point.
pointToNdc(wcsPos, viewMatrix, projectionMatrix)	Map the position of a point in world space to normalized device coordinates/space.
cursorToRay(cursorX, cursorY, winSize, ...)	Convert a 2D mouse coordinate to a 3D ray.
visible(points, mvp[, mode, dtype])	Test if points are visible.
visibleBBox(extents, mvp[, dtype])	Check if a bounding box is visible.

Details

psychopy.tools.viewtools.**visualAngle**(*size*, *distance*, *degrees=True*, *out=None*, *dtype=None*)

Get the visual angle for an object of *size* at *distance*. Object is assumed to be fronto-parallel with the viewer.

This function supports vector inputs. Values for *size* and *distance* can be arrays or single values. If both inputs are arrays, they must have the same size.

Parameters

- **size** (*float or array_like*) – Size of the object in meters.
- **distance** (*float or array_like*) – Distance to the object in meters.
- **degrees** (*bool*) – Return result in degrees, if *False* result will be in radians.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for arrays, can either be 'float32' or 'float64'. If *None* is specified, the data type is inferred by *out*. If *out* is not provided, the default is 'float64'.

Returns

Visual angle.

Return type

float

Examples

Calculating the visual angle (vertical FOV) of a monitor screen:

```
monDist = 0.5  # monitor distance, 50cm
monHeight = 0.45  # monitor height, 45cm

vertFOV = visualAngle(monHeight, monDist)
```

Compute visual angle at multiple distances for objects with the same size:

```
va = visualAngle(0.20, [1.0, 2.0, 3.0])  # returns
# [11.42118627  5.72481045  3.81830487]
```

psychopy.tools.viewtools.**computeFrustum**(*scrWidth*, *scrAspect*, *scrDist*, *convergeOffset=0.0*, *eyeOffset=0.0*, *nearClip=0.01*, *farClip=100.0*, *dtype=None*)

Calculate frustum parameters. If an eye offset is provided, an asymmetric frustum is returned which can be used for stereoscopic rendering.

Parameters

- **scrWidth** (*float*) – The display's width in meters.
- **scrAspect** (*float*) – Aspect ratio of the display (width / height).
- **scrDist** (*float*) – Distance to the screen from the view in meters. Measured from the center of their eyes.
- **convergeOffset** (*float*) – Offset of the convergence plane from the screen. Objects falling on this plane will have zero disparity. For best results, the convergence plane should be set to the same distance as the screen (0.0 by default).
- **eyeOffset** (*float*) – Half the inter-ocular separation (i.e. the horizontal distance between the nose and center of the pupil) in meters. If eyeOffset is 0.0, a symmetric frustum is returned.
- **nearClip** (*float*) – Distance to the near clipping plane in meters from the viewer. Should be at least less than *scrDist*.
- **farClip** (*float*) – Distance to the far clipping plane from the viewer in meters. Must be >nearClip.
- **dtype** (*dtype or str, optional*) – Data type for arrays, can either be 'float32' or 'float64'. If *None* is specified, the data type is inferred by *out*. If *out* is not provided, the default is 'float64'.

Returns

Array of frustum parameters. Can be directly passed to glFrustum (e.g. glFrustum(*f)).

Return type

ndarray

Notes

- The view point must be transformed for objects to appear correctly. Offsets in the X-direction must be applied +/- eyeOffset to account for inter-ocular separation. A transformation in the Z-direction must be applied to account for screen distance. These offsets MUST be applied to the GL_MODELVIEW matrix, not the GL_PROJECTION matrix! Doing so may break lighting calculations.

Examples

Creating a frustum and setting a window's projection matrix:

```
scrWidth = 0.5  # screen width in meters
scrAspect = win.size[0] / win.size[1]
scrDist = win.scrDistCM * 100.0  # monitor setting, can be anything
frustum = viewtools.computeFrustum(scrWidth, scrAspect, scrDist)
```

Accessing frustum parameters:

```
left, right, bottom, top, nearVal, farVal = frustum
# ... or ...
left = frustum.left
```

Off-axis frustums for stereo rendering:

```
# compute view matrix for each eye, these value usually don't change
eyeOffset = (-0.035, 0.035)  # +/- IOD / 2.0
scrDist = 0.50  # 50cm
scrWidth = 0.53  # 53cm
scrAspect = 1.778
leftFrustum = viewtools.computeFrustum(
    scrWidth, scrAspect, scrDist, eyeOffset[0])
rightFrustum = viewtools.computeFrustum(
    scrWidth, scrAspect, scrDist, eyeOffset[1])
# make sure your view matrix accounts for the screen distance and eye
# offsets!
```

Using computed view frustums with a window:

```
win.projectionMatrix = viewtools.perspectiveProjectionMatrix(*frustum)
# generate a view matrix looking ahead with correct viewing distance,
# origin is at the center of the screen. Assumes eye is centered with
# the screen.
eyePos = [0.0, 0.0, scrDist]
screenPos = [0.0, 0.0, 0.0]  # look at screen center
eyeUp = [0.0, 1.0, 0.0]
win.viewMatrix = viewtools.lookAt(eyePos, screenPos, eyeUp)
win.applyViewTransform()  # call before drawing
```

`psychopy.tools.viewtools.`**`computeFrustumFOV`**(*scrFOV*, *scrAspect*, *scrDist*, *convergeOffset=0.0*, *eyeOffset=0.0*, *nearClip=0.01*, *farClip=100.0*, *dtype=None*)

Compute a frustum for a given field-of-view (FOV).

Similar to *computeFrustum*, but computes a frustum based on FOV rather than screen dimensions.

Parameters

- **`scrFOV`** (*`float`*) – Vertical FOV in degrees (fovY).
- **`scrAspect`** (*`float`*) – Aspect between the horizontal and vertical FOV (ie. fovX / fovY).
- **`scrDist`** (*`float`*) – Distance to the screen from the view in meters. Measured from the center of the viewer's eye(s).
- **`convergeOffset`** (*`float`*) – Offset of the convergence plane from the screen. Objects falling on this plane will have zero disparity. For best results, the convergence plane should

be set to the same distance as the screen (0.0 by default).

- **eyeOffset** (*float*) – Half the inter-ocular separation (i.e. the horizontal distance between the nose and center of the pupil) in meters. If eyeOffset is 0.0, a symmetric frustum is returned.
- **nearClip** (*float*) – Distance to the near clipping plane in meters from the viewer. Should be at least less than *scrDist*. Never should be 0.
- **farClip** (*float*) – Distance to the far clipping plane from the viewer in meters. Must be >nearClip.
- **dtype** (*dtype or str, optional*) – Data type for arrays, can either be 'float32' or 'float64'. If *None* is specified, the data type is inferred by *out*. If *out* is not provided, the default is 'float64'.

Examples

Equivalent to *gluPerspective*:

```
frustum =  computeFrustumFOV(45.0, 1.0, 0.5)
projectionMatrix = perspectiveProjectionMatrix(*frustum)
```

psychopy.tools.viewtools.**projectFrustum**(*frustum*, *dist*, *dtype=None*)

Project a frustum on a fronto-parallel plane and get the width and height of the required drawing area.

This function can be used to determine the size of the drawing area required for a given frustum on a screen. This is useful for cases where the observer is viewing the screen through a physical aperture that limits the FOV to a sub-region of the display. You must convert the size in meters to units of your screen and apply any offsets.

Parameters

- **frustum** (*array_like*) – Frustum parameters (left, right, bottom, top, near, far), you can exclude *far* since it is not used in this calculation. However, the function will still succeed if given.
- **dist** (*float*) – Distance to project points to in meters.
- **dtype** (*dtype or str, optional*) – Data type for arrays, can either be 'float32' or 'float64'. If *None* is specified, the data type is inferred by *out*. If *out* is not provided, the default is 'float64'.

Returns

Width and height (w, h) of the area intersected by the given frustum at *dist*.

Return type

ndarray

Examples

Compute the viewport required to draw in the area where the frustum intersects the screen:

```
# needed information
scrWidthM = 0.52
scrDistM = 0.72
scrWidthPIX = 1920
scrHeightPIX = 1080
scrAspect = scrWidthPIX / float(scrHeightPIX)
pixPerMeter = scrWidthPIX / scrWidthM
```

(continues on next page)

(continued from previous page)

```
# Compute a frustum for 20 degree vertical FOV at distance of the
# screen.
frustum = computeFrustumFOV(20., scrAspect, scrDistM)

# get the dimensions of the frustum
w, h = projectFrustum(frustum, scrDistM) * pixPerMeter

# get the origin of the viewport, relative to center of screen.
x = (scrWidthPIX - w) / 2.
y = (scrHeightPIX - h) / 2.

# if there is an eye offset ...
# x = (scrWidthPIX - w + eyeOffsetM * pixPerMeter) / 2.

# viewport rectangle
rect = np.asarray((x, y, w, h), dtype=int)
```

You can then set the viewport/scissor rectangle of the buffer to restrict drawing to *rect*.

`psychopy.tools.viewtools.`**`projectFrustumToPlane`**(*frustum*, *planeOrig*, *dtype=None*)

Project a frustum on a fronto-parallel plane and get the coordinates of the corners in physical space.

Parameters

- **frustum** (*array_like*) – Frustum parameters (left, right, bottom, top, near, far), you can exclude *far* since it is not used in this calculation. However, the function will still succeed if given.
- **planeOrig** (*float*) – Distance of plane to project points on in meters.
- **dtype** (*dtype or str, optional*) – Data type for arrays, can either be 'float32' or 'float64'. If *None* is specified, the data type is inferred by *out*. If *out* is not provided, the default is 'float64'.

Returns

4x3 array of coordinates in the physical reference frame with origin at the eye.

Return type

ndarray

`psychopy.tools.viewtools.`**`generalizedPerspectiveProjection`**(*posBottomLeft*, *posBottomRight*, *posTopLeft*, *eyePos*, *nearClip=0.01*, *farClip=100.0*, *dtype=None*)

Generalized derivation of projection and view matrices based on the physical configuration of the display system.

This implementation is based on Robert Kooima's 'Generalized Perspective Projection' method[1].

Parameters

- **posBottomLeft** (*list of float or ndarray*) – Bottom-left 3D coordinate of the screen in meters.
- **posBottomRight** (*list of float or ndarray*) – Bottom-right 3D coordinate of the screen in meters.

[1] Kooima, R. (2009). Generalized perspective projection. J. Sch. Electron. Eng. Comput. Sci.

- **posTopLeft** (*list of float or ndarray*) – Top-left 3D coordinate of the screen in meters.
- **eyePos** (*list of float or ndarray*) – Coordinate of the eye in meters.
- **nearClip** (*float*) – Near clipping plane distance from viewer in meters.
- **farClip** (*float*) – Far clipping plane distance from viewer in meters.
- **dtype** (*dtype or str, optional*) – Data type for arrays, can either be 'float32' or 'float64'. If *None* is specified, the data type is inferred by *out*. If *out* is not provided, the default is 'float64'.

Returns

The 4x4 projection and view matrix.

Return type

tuple

See also

computeFrustum

Compute frustum parameters.

Notes

- The resulting projection frustums are off-axis relative to the center of the display.
- The returned matrices are row-major. Values are floats with 32-bits of precision stored as a contiguous (C-order) array.

References

Examples

Computing a projection and view matrices for a window:

```
projMatrix, viewMatrix = viewtools.generalizedPerspectiveProjection(
    posBottomLeft, posBottomRight, posTopLeft, eyePos)
# set the window matrices
win.projectionMatrix = projMatrix
win.viewMatrix = viewMatrix
# before rendering
win.applyEyeTransform()
```

Stereo-pair rendering example from Kooima (2009):

```
# configuration of screen and eyes
posBottomLeft = [-1.5, -0.75, -18.0]
posBottomRight = [1.5, -0.75, -18.0]
posTopLeft = [-1.5, 0.75, -18.0]
posLeftEye = [-1.25, 0.0, 0.0]
posRightEye = [1.25, 0.0, 0.0]
# create projection and view matrices
leftProjMatrix, leftViewMatrix = generalizedPerspectiveProjection(
    posBottomLeft, posBottomRight, posTopLeft, posLeftEye)
```

(continues on next page)

(continued from previous page)

```
rightProjMatrix, rightViewMatrix = generalizedPerspectiveProjection(
    posBottomLeft, posBottomRight, posTopLeft, posRightEye)
```

psychopy.tools.viewtools.**orthoProjectionMatrix**(*left*, *right*, *bottom*, *top*, *nearClip=0.01*, *farClip=100.0*, *out=None*, *dtype=None*)

Compute an orthographic projection matrix with provided frustum parameters.

Parameters

- **left** (*float*) – Left clipping plane coordinate.
- **right** (*float*) – Right clipping plane coordinate.
- **bottom** (*float*) – Bottom clipping plane coordinate.
- **top** (*float*) – Top clipping plane coordinate.
- **nearClip** (*float*) – Near clipping plane distance from viewer.
- **farClip** (*float*) – Far clipping plane distance from viewer.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for arrays, can either be 'float32' or 'float64'. If *None* is specified, the data type is inferred by *out*. If *out* is not provided, the default is 'float64'.

Returns
4x4 projection matrix

Return type
ndarray

See also

perspectiveProjectionMatrix
Compute a perspective projection matrix.

Notes

- The returned matrix is row-major. Values are floats with 32-bits of precision stored as a contiguous (C-order) array.

psychopy.tools.viewtools.**perspectiveProjectionMatrix**(*left*, *right*, *bottom*, *top*, *nearClip=0.01*, *farClip=100.0*, *out=None*, *dtype=None*)

Compute an perspective projection matrix with provided frustum parameters. The frustum can be asymmetric.

Parameters

- **left** (*float*) – Left clipping plane coordinate.
- **right** (*float*) – Right clipping plane coordinate.
- **bottom** (*float*) – Bottom clipping plane coordinate.
- **top** (*float*) – Top clipping plane coordinate.
- **nearClip** (*float*) – Near clipping plane distance from viewer.

- **farClip** (*float*) – Far clipping plane distance from viewer.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for arrays, can either be 'float32' or 'float64'. If *None* is specified, the data type is inferred by *out*. If *out* is not provided, the default is 'float64'.

Returns
4x4 projection matrix

Return type
ndarray

See also

orthoProjectionMatrix
Compute a orthographic projection matrix.

Notes

- The returned matrix is row-major. Values are floats with 32-bits of precision stored as a contiguous (C-order) array.

`psychopy.tools.viewtools.`**`pointToNdc`**(*wcsPos*, *viewMatrix*, *projectionMatrix*, *out=None*, *dtype=None*)

Map the position of a point in world space to normalized device coordinates/space.

Parameters

- **wcsPos** (*tuple, list or ndarray*) – Nx3 position vector(s) (xyz) in world space coordinates.
- **viewMatrix** (*ndarray*) – 4x4 view matrix.
- **projectionMatrix** (*ndarray*) – 4x4 projection matrix.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for arrays, can either be 'float32' or 'float64'. If *None* is specified, the data type is inferred by *out*. If *out* is not provided, the default is 'float64'.

Returns
3x1 vector of normalized device coordinates with type 'float32'

Return type
ndarray

Notes

- The point is not visible, falling outside of the viewing frustum, if the returned coordinates fall outside of -1 and 1 along any dimension.
- In the rare instance the point falls directly on the eye in world space where the frustum converges to a point (singularity), the divisor will be zero during perspective division. To avoid this, the divisor is 'bumped' to 1e-5.

- This function assumes the display area is rectilinear. Any distortion or warping applied in normalized device or viewport space is not considered.

Examples

Determine if a point is visible:

```
point = (0.0, 0.0, 10.0)  # behind the observer
ndc = pointToNdc(point, win.viewMatrix, win.projectionMatrix)
isVisible = not np.any((ndc > 1.0) | (ndc < -1.0))
```

Convert NDC to viewport (or pixel) coordinates:

```
scrRes = (1920, 1200)
point = (0.0, 0.0, -5.0)  # forward -5.0 from eye
x, y, z = pointToNdc(point, win.viewMatrix, win.projectionMatrix)
pixelX = ((x + 1.0) / 2.0) * scrRes[0])
pixelY = ((y + 1.0) / 2.0) * scrRes[1])
# object at point will appear at (pixelX, pixelY)
```

psychopy.tools.viewtools.**cursorToRay**(*cursorX*, *cursorY*, *winSize*, *viewport*, *projectionMatrix*, *normalize=True*, *out=None*, *dtype=None*)

Convert a 2D mouse coordinate to a 3D ray.

Takes a 2D window/mouse coordinate and transforms it to a 3D direction vector from the viewpoint in eye space (vector origin is [0, 0, 0]). The center of the screen projects to vector [0, 0, -1].

Parameters

- **cursorX** (*float or int*) – Window coordinates. These need to be scaled if you are using a framebuffer that does not have 1:1 pixel mapping (i.e. retina display).
- **cursorY** (*float or int*) – Window coordinates. These need to be scaled if you are using a framebuffer that does not have 1:1 pixel mapping (i.e. retina display).
- **winSize** (*array_like*) – Size of the window client area [w, h].
- **viewport** (*array_like*) – Viewport rectangle [x, y, w, h] being used.
- **projectionMatrix** (*ndarray*) – 4x4 projection matrix being used.
- **normalize** (*bool*) – Normalize the resulting vector.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for arrays, can either be 'float32' or 'float64'. If *None* is specified, the data type is inferred by *out*. If *out* is not provided, the default is 'float64'.

Returns

Direction vector (x, y, z).

Return type

ndarray

Examples

Place a 3D stim at the mouse location 5.0 scene units (meters) away:

```
# define camera
camera = RigidBodyPose((-3.0, 5.0, 3.5))
camera.alignTo((0, 0, 0))

# in the render loop

dist = 5.0
mouseRay = vt.cursorToRay(x, y, win.size, win.viewport, win.projectionMatrix)
mouseRay *= dist  # scale the vector

# set the sphere position by transforming vector to world space
sphere.thePose.pos = camera.transform(mouseRay)
```

psychopy.tools.viewtools.**visible**(*points*, *mvp*, *mode='discrete'*, *dtype=None*)

Test if points are visible.

This function is useful for visibility culling, where objects are only drawn if a portion of them are visible. This test can avoid costly drawing calls and OpenGL state changes if the object is not visible.

Parameters

- **points** (*array_like*) – Point(s) or bounding box to test. Input array must be Nx3 or Nx4, where each row is a point. It is recommended that the input be Nx4 since the *w* component will be appended if the input is Nx3 which adds overhead.
- **mvp** (*array_like*) – 4x4 MVP matrix.
- **mode** (*str*) – Test mode. If *'discrete'*, rows of *points* are treated as individual points. This function will return an array of boolean values with length equal to the number of rows in *points*, where the value at each index corresponds to the visibility test results for points at the matching row index of *points*. If *'group'* a single boolean value is returned, which is *False* if all points fall to one side of the frustum.
- **dtype** (*dtype or str, optional*) – Data type for arrays, can either be 'float32' or 'float64'. If *None* is specified, the data type is inferred by *out*. If *out* is not provided, the default is 'float64'.

Returns

Test results. The type returned depends on *mode*.

Return type

bool or ndarray

Examples

Visibility culling, only a draw line connecting two points if visible:

```
linePoints = [[-1.0, -1.0, -1.0, 1.0],
              [ 1.0,  1.0,  1.0, 1.0]]

mvp = np.matmul(win.projectionMatrix, win.viewMatrix)
if visible(linePoints, mvp, mode='group'):
    # drawing commands here ...
```

psychopy.tools.viewtools.**visibleBBox**(*extents*, *mvp*, *dtype=None*)

Check if a bounding box is visible.

This function checks if a bonding box intersects a frustum defined by the current projection matrix, after being transformed by the model-view matrix.

Parameters

- **extents** (*array_like*) – Bounding box minimum and maximum extents as a 2x3 array. The first row if the minimum extents along each axis, and the second row the maximum extents (eg. [[minX, minY, minZ], [maxX, maxY, maxZ]]).
- **mvp** (*array_like*) – 4x4 MVP matrix.
- **dtype** (*dtype or str, optional*) – Data type for arrays, can either be 'float32' or 'float64'. If *None* is specified, the data type is inferred by *out*. If *out* is not provided, the default is 'float64'.

Returns

Visibility test results.

Return type

ndarray or bool

11.9 psychopy.app - the application suite

This module contains everything needed to run and manage the wxPython GUI application suite (e.g., Coder, Builder and Runner).

The functions presented here provide a simple interface to the application instance and any frames associated with it. This interface is intended for unit testing the GUI and for developers wishing to extend it.

11.10 Overview

startApp([showSplash, testMode, safeMode, ...])	Start the PsychoPy GUI.
quitApp()	Quit the running PsychoPy application instance.
isAppStarted()	Check if the GUI portion of PsychoPy is running.
getAppInstance()	Get a reference to the *PsychoPyApp* object.
getAppFrame(frameName)	Get the reference to one of PsychoPy's application frames.

11.11 Details

psychopy.app.**startApp**(*showSplash=True*, *testMode=False*, *safeMode=False*, *startView=None*, *startFiles=None*, *firstRun=False*, *profiling=False*)

Start the PsychoPy GUI.

This function is idempotent, where additional calls after the app starts will have no effect unless *quitApp()* was previously called. After this function returns, you can get the handle to the created *PsychoPyApp* instance by calling *getAppInstance()* (returns *None* otherwise).

Errors raised during initialization due to unhandled exceptions with respect to the GUI application are usually fatal. You can examine 'last_app_load.log' inside the 'psychopy3' user directory (specified by preference 'userPrefsDir') to see the traceback. After startup, unhandled exceptions will appear in a special dialog box that shows the error traceback and provides some means to recover their work. Regular logging messages will appear in the log file or GUI. We use a separate error dialog here is delineate errors occurring in the user's experiment scripts and those of the application itself.

Parameters

- **showSplash** (*bool*) – Show the splash screen on start.
- **testMode** (*bool*) – Must be *True* if creating an instance for unit testing.
- **safeMode** (*bool*) – Start PsychoPy in safe-mode. If *True*, the GUI application will launch with without loading plugins.
- **startView** (*str, None*) – Name of the view to start the app with. Valid values are 'coder', 'builder' or 'runner'. If *None*, the app will start with the default view or the view specifed with the *PSYCHOPYSTARTVIEW* environment variable.

psychopy.app.**quitApp**()

Quit the running PsychoPy application instance.

Will have no effect if *startApp()* has not been called previously.

psychopy.app.**isAppStarted**()

Check if the GUI portion of PsychoPy is running.

Returns

True if the GUI is started else *False*.

Return type

bool

psychopy.app.**getAppInstance**()

Get a reference to the *PsychoPyApp* object.

This function will return *None* if PsychoPy has been imported as a library or the app has not been fully realized.

Returns

Handle to the application instance. Returns *None* if the app has not been started yet or the PsychoPy is being used without a GUI.

Return type

PsychoPyApp or None

Examples

Get the coder frame (if any):

```
import psychopy.app as app
coder = app.getAppInstance().coder
```

psychopy.app.**getAppFrame**(*frameName*)

Get the reference to one of PsychoPy's application frames. Returns *None* if the specified frame has not been fully realized yet or PsychoPy is not in GUI mode.

Parameters

frameName (*str*) – Identifier for the frame to get a reference to. Valid names are 'coder', 'builder' or 'runner'.

Returns

Reference to the frame instance (i.e. *CoderFrame*, *BuilderFrame* or *RunnerFrame*). *None* is returned if the frame has not been created or the app is not running. May return a list if more than one window is opened.

Return type
object or None

11.12 psychopy.colors - For working with colors

Classes and functions for working with colors.

11.12.1 Overview

Color([color, space, contrast, conematrix])	A class to store color details, knows what colour space it's in and can supply colours in any space.
isValidColor(color[, space])	Depreciated as of 2021.0
hex2rgb255(hexColor)	Depreciated as of 2021.0

11.12.2 Details

class psychopy.colors.**Color**(*color=None*, *space=None*, *contrast=None*, *conematrix=None*)

A class to store color details, knows what colour space it's in and can supply colours in any space.

Parameters

- **color** (*ArrayLike or None*) – Color values (coordinates). Value must be in a format applicable to the specified *space*.
- **space** (*str or None*) – Colorspace to interpret the value of *color* as being within.
- **contrast** (*int or float*) – Factor to modulate the contrast of the color.
- **conematrix** (*ArrayLike or None*) – Cone matrix for colorspaces which require it. Must be a 3x3 array.

property alpha

How opaque (1) or transparent (0) this color is. Synonymous with *opacity*.

property contrast

copy()

Return a duplicate of this colour

property dkl

Color value expressed as a DKL triplet.

property dklCart

Color value expressed as a cartesian DKL triplet.

property dkla

Color value expressed as a DKL triplet, with alpha value (0 to 1).

property dklaCart

Color value expressed as a cartesian DKL triplet, with alpha value (0 to 1).

getReadable(*contrast=0.21428571428571427*)

Get a color which will stand out and be easily readable against this one. Useful for choosing text colors based on background color.

Parameters

`contrast` (*float*) – Desired perceived contrast between the two colors, between 0 (the same color) and 1 (as opposite as possible). Default is the w3c recommended minimum of 4.5/21 (dividing by 21 to adjust for sRGB units).

Returns

A contrasting color to this color.

Return type

colors.Color

`property hex`

Color value expressed as a hex string. Can be a '#' followed by 6 values from 0 to F (e.g. #F2545B).

`property hsv`

Color value expressed as an HSV triplet.

`property hsva`

Color value expressed as an HSV triplet, with alpha value (0 to 1).

`property lms`

Color value expressed as an LMS triplet.

`property lmsa`

Color value expressed as an LMS triplet, with alpha value (0 to 1).

`property named`

The name of this color, if it has one (*str*).

`property opacity`

How opaque (1) or transparent (0) this color is (*float*). Synonymous with *alpha*.

`render`(*space='rgb'*)

Apply contrast to the base color value and return the adjusted color value.

`property rgb`

Color value expressed as an RGB triplet from -1 to 1.

`property rgb1`

Color value expressed as an RGB triplet from 0 to 1.

`property rgb255`

Color value expressed as an RGB triplet from 0 to 255.

`property rgba`

Color value expressed as an RGB triplet from -1 to 1, with alpha values (0 to 1).

`property rgba1`

Color value expressed as an RGB triplet from 0 to 1, with alpha value (0 to 1).

`property rgba255`

Color value expressed as an RGB triplet from 0 to 255, with alpha value (0 to 1).

`set`(*color=None*, *space=None*)

Set the colour of this object - essentially the same as what happens on creation, but without having to initialise a new object.

`property srgb`

Color value expressed as an sRGB triplet

validate(*color*, *space=None*)

Check that a color value is valid in the given space, or all spaces if space==None.

psychopy.colors.**isValidColor**(*color*, *space='rgb'*)

Depreciated as of 2021.0

psychopy.colors.**hex2rgb255**(*hexColor*)

Depreciated as of 2021.0

Converts a hex color string (e.g. "#05ff66") into an rgb triplet ranging from 0:255

11.13 psychopy.data - functions for storing/saving/analysing data

Contents:

- *ExperimentHandler* - to combine multiple loops in one study
- *TrialHandler* - basic predefined trial matrix
- *TrialHandler2* - similar to TrialHandler but with ability to update mid-run
- *TrialHandlerExt* - similar to TrialHandler but with ability to run oddball designs
- *StairHandler* - for basic up-down (fixed step) staircases
- *QuestHandler* - for traditional QUEST algorithm
- *QuestPlusHandler* - for the updated QUEST+ algorithm (Watson, 2017)
- *PsiHandler* - the Psi staircase of Kontsevich & Tyler (1999)
- *MultiStairHandler* - a wrapper to combine interleaved staircases of any sort

Utility functions:

- *importConditions()* - to load a list of dicts from a csv/excel file
- *functionFromStaircase()*- to convert a staircase into its psychopmetric function
- *bootStraps()* - generate a set of bootstrap resamples from a dataset
- getDateStr() - provide a date string (in format suitable for filenames)

Curve Fitting:

- *FitWeibull*
- *FitLogistic*
- *FitNakaRushton*
- *FitCumNormal*

11.13.1 ExperimentHandler

class psychopy.data.**ExperimentHandler**(*name=''*, *version=''*, *extraInfo=None*, *runtimeInfo=None*, *originPath=None*, *savePickle=True*, *saveWideText=True*, *sortColumns=False*, *dataFileName=''*, *autoLog=True*, *appendFiles=False*)

A container class for keeping track of multiple loops/handlers

Useful for generating a single data file from an experiment with many different loops (e.g. interleaved staircases or loops within loops

Usage

exp = data.ExperimentHandler(name="Face Preference",version='0.1.0')

Parameters

name

[a string or unicode] As a useful identifier later

version

[usually a string (e.g. '1.1.0')] To keep track of which version of the experiment was run

extraInfo

[a dictionary] Containing useful information about this run (e.g. {'participant':'jwp','gender':'m','orientation':90})

runtimeInfo

[*psychopy.info.RunTimeInfo*] Containing information about the system as detected at runtime

originPath

[string or unicode] The path and filename of the originating script/experiment If not provided this will be determined as the path of the calling script.

dataFileName

[string] This is defined in advance and the file will be saved at any point that the handler is removed or discarded (unless .abort() had been called in advance). The handler will attempt to populate the file even in the event of a (not too serious) crash!

savePickle : True (default) or False

saveWideText : True (default) or False

sortColumns

[str or bool] How (if at all) to sort columns in the data file, if none is given to saveAsWideText. Can be: - "alphabetical", "alpha", "a" or True: Sort alphabetically by header name - "priority", "pr" or "p": Sort according to priority - other: Do not sort, columns remain in order they were added

autoLog : True (default) or False

_getAllParamNames()

Returns the attribute names of loop parameters (trialN etc) that the current set of loops contain, ready to build a wide-format data file.

_getExtraInfo()

Get the names and vals from the extraInfo dict (if it exists)

_getLoopInfo(*loop*)

Returns the attribute names and values for the current trial of a particular loop. Does not return data inputs from the subject, only info relating to the trial execution.

_guessPriority(*name*)

Get a best guess at the priority of a column based on its name

Parameters

name (*str*) – Name of the column

Returns

One of the following: - HIGH (19): Important columns which are near the front of the data file - MEDIUM (9): Possibly important columns which are around the middle of the data file - LOW (-1): Columns unlikely to be important which are at the end of the data file

NOTE: Values returned from this function are 1 less than values in *constants.priority*, columns whose priority was guessed are behind equivalently prioritised columns whose priority was specified.

Return type
int

abort()

Inform the ExperimentHandler that the run was aborted.

Experiment handler will attempt automatically to save data (even in the event of a crash if possible). So if you quit your script early you may want to tell the Handler not to save out the data files for this run. This is the method that allows you to do that.

addAnnotation(*value*)

Add an annotation at the current point in the experiment

Parameters
value (*str*) – Value of the annotation

addData(*name*, *value*, *row=None*, *priority=None*)

Add the data with a given name to the current experiment.

Typically the user does not need to use this function; if you added your data to the loop and had already added the loop to the experiment then the loop will automatically inform the experiment that it has received data.

Multiple data name/value pairs can be added to any given entry of the data file and is considered part of the same entry until the nextEntry() call is made.

e.g.:

```
# add some data for this trial
exp.addData('resp.rt', 0.8)
exp.addData('resp.key', 'k')
# end of trial - move to next line in data output
exp.nextEntry()
```

Parameters

- **name** (*str*) – Name of the column to add data as.
- **value** (*any*) – Value to add
- **row** (*int or None*) – Row in which to add this data. Leave as None to add to the current entry.
- **priority** (*int*) – Priority value to set the column to - higher priority columns appear nearer to the start of the data file. Use values from *constants.priority* as landmark values: - CRITICAL: Always at the start of the data file, generally reserved for Routine start times - HIGH: Important columns which are near the front of the data file - MEDIUM: Possibly important columns which are around the middle of the data file - LOW: Columns unlikely to be important which are at the end of the data file - EXCLUDE: Always at the end of the data file, actively marked as unimportant

addLoop(*loopHandler*)

Add a loop such as a *TrialHandler* or *StairHandler* Data from this loop will be included in the resulting data files.

`close()`

`connectSaveMethod(`*fcn*, **args*, ***kwargs*`)`

Tell this experiment handler to call the given function with the given arguments and keyword arguments whenever it saves its own data.

Parameters

- **fcn** (*function*) – Function to call
- ***args** – Positional arguments to be given to the function when it's called
- ****kwargs** – Keyword arguments to be given to the function when it's called

`property currentLoop`

Return the loop which we are currently in, this will either be a handle to a loop, such as a *TrialHandler* or *StairHandler*, or the handle of the *ExperimentHandler* itself if we are not in a loop.

`getAllEntries()`

Fetches a copy of all the entries including a final (orphan) entry if that exists. This allows entries to be saved even if nextEntry() is not yet called.

Returns
copy (not pointer) to entries

`getAllTrials()`

Returns all trials (elapsed, current and upcoming) with an index indicating which trial is the current trial.

Returns

- *list[Trial]* – List of trials, in order (oldest to newest)
- *int* – Index of the current trial in this list

`getCurrentTrial()`

Returns the current trial (*.thisTrial*)

Returns
The current trial

Return type
Trial

`getFutureTrial(`*n=1*`)`

Returns the condition for n trials into the future, without advancing the trials. Returns 'None' if attempting to go beyond the last trial in the current loop, or if there is no current loop.

`getFutureTrials(`*n=1*, *start=0*`)`

Returns Trial objects for a given range in the future. Will start looking at *start* trials in the future and will return n trials from then, so e.g. to get all trials from 2 in the future to 5 in the future you would use *start=2* and *n=3*.

Parameters

- **n** (*int, optional*) – How many trials into the future to look, by default 1
- **start** (*int, optional*) – How many trials into the future to start looking at, by default 0

Returns
List of Trial objects n long. Any trials beyond the last trial are None.

Return type
list[Trial or None]

`getJSON`(*priorityThreshold=-9*)

Get the experiment data as a JSON string.

Parameters
priorityThreshold (*int*) – Output will only include columns whose priority is greater than or equal to this value. Use values in psychopy.constants.priority as a guideline for priority levels. Default is -9 (constants.priority.EXCLUDE + 1)

Returns
JSON string with the following fields: - 'type': Indicates that this is data from an ExperimentHandler (will always be "trials_data") - 'trials': *list* of *dict`s representing requested trials data - 'priority': `dict* of column names

Return type
str

`getPriority`(*name*)

Get the priority value for a given column. If no priority value is stored, returns best guess based on column name.

Parameters
name (*str*) – Column name

Returns
The priority value stored/guessed for this column, most likely a value from *constants.priority*, one of: - CRITICAL (30): Always at the start of the data file, generally reserved for Routine start times - HIGH (20): Important columns which are near the front of the data file - MEDIUM (10): Possibly important columns which are around the middle of the data file - LOW (0): Columns unlikely to be important which are at the end of the data file - EXCLUDE (-10): Always at the end of the data file, actively marked as unimportant

Return type
int

`loopEnded`(*loopHandler*)

Informs the experiment handler that the loop is finished and not to include its values in further entries of the experiment.

This method is called by the loop itself if it ends its iterations, so is not typically needed by the user.

`nextEntry`()

Calling nextEntry indicates to the ExperimentHandler that the current trial has ended and so further addData() calls correspond to the next trial.

`pause`()

Set status to be PAUSED.

`queueNextCollision`(*fileCollisionMethod*, *fileName=None*)

Tell this ExperimentHandler than, next time the named file is saved, it should handle collisions a certain way. This is useful if you want to save multiple times within an experiment.

Parameters

- **fileCollisionMethod** (*str*) – File collision method to use, see *saveAsWideText* or *saveAsPickle* for details.
- **fileName** (*str*) – Filename to queue collision on, if None (default) will use this ExperimentHandler's *dataFileName*

resume()

Set status to be STARTED.

rewindTrials(*n=1*)

Skip ahead n trials - the trials inbetween will be marked as "skipped". If you try to skip past the last trial, will log a warning and skip *to* the last trial.

Parameters
n (*int*) – Number of trials to skip ahead

save()

Work out from own settings how to save, then use the appropriate method (saveAsWideText, saveAsPickle, etc.)

saveAsPickle(*fileName*, *fileCollisionMethod=None*)

Basically just saves a copy of self (with data) to a pickle file.

This can be reloaded if necessary and further analyses carried out.

Parameters
fileCollisionMethod (*str*) – Collision method passed to `handleFileCollision()`

Returns
Final filename (including _1, _2, etc. and file extension) which data was saved as

Return type
str

saveAsWideText(*fileName*, *delim='auto'*, *matrixOnly=False*, *appendFile=None*, *encoding='utf-8-sig'*, *fileCollisionMethod=None*, *sortColumns=None*)

Saves a long, wide-format text file, with one line representing the attributes and data for a single trial. Suitable for analysis in R and SPSS.

If *appendFile=True* then the data will be added to the bottom of an existing file. Otherwise, if the file exists already it will be kept and a new file will be created with a slightly different name. If you want to overwrite the old file, pass 'overwrite' to `fileCollisionMethod`.

If *matrixOnly=True* then the file will not contain a header row, which can be handy if you want to append data to an existing file of the same format.

Parameters

- **fileName** – if extension is not specified, '.csv' will be appended if the delimiter is ',', else '.tsv' will be appended. Can include path info.
- **delim** – allows the user to use a delimiter other than the default tab ("," is popular with file extension ".csv")
- **matrixOnly** – outputs the data with no header row.
- **appendFile** – will add this output to the end of the specified file if it already exists.
- **encoding** – The encoding to use when saving a the file. Defaults to *utf-8-sig*.
- **fileCollisionMethod** – Collision method passed to `handleFileCollision()`
- **sortColumns** (*str or bool*) – How (if at all) to sort columns in the data file. Can be: - "alphabetical", "alpha", "a" or True: Sort alphabetically by header name - "priority", "pr" or "p": Sort according to priority - other: Do not sort, columns remain in order they were added

Returns
Final filename (including _1, _2, etc. and file extension) which data was saved as

Return type
str

`setPriority`(*name*, *value=20*)

Set the priority of a column in the data file.

Parameters

- **name** (*str*) – Name of the column, e.g. *text.started*
- **value** (*int*) – Priority value to set the column to - higher priority columns appear nearer to the start of the data file. Use values from *constants.priority* as landmark values: - CRITICAL (30): Always at the start of the data file, generally reserved for Routine start times - HIGH (20): Important columns which are near the front of the data file - MEDIUM (10): Possibly important columns which are around the middle of the data file - LOW (0): Columns unlikely to be important which are at the end of the data file - EXCLUDE (-10): Always at the end of the data file, actively marked as unimportant

`skipTrials`(*n=1*)

Skip ahead n trials - the trials inbetween will be marked as "skipped". If you try to skip past the last trial, will log a warning and skip *to* the last trial.

Parameters
n (*int*) – Number of trials to skip ahead

`property status`

`stop`()

Set status to be FINISHED.

`timestampOnFlip`(*win*, *name*, *format=<class 'float'>*)

Add a timestamp (in the future) to the current row

Parameters

- **win** (`psychopy.visual.Window`) – The window object that we'll base the timestamp flip on
- **name** (*str*) – The name of the column in the datafile being written, such as 'myStim.stopped'
- **format** (*str, class or None*) – Format in which to return time, see clock.Timestamp.resolve() for more info. Defaults to *float*.

`updateEntryFromLoop`(*thisLoop*)

Add all values from the given loop to the current entry.

Parameters
thisLoop (*BaseLoopHandler*) – Loop to get fields from

11.13.2 TrialHandler

`class psychopy.data.TrialHandler`(*trialList*, *nReps*, *method='random'*, *dataTypes=None*, *extraInfo=None*, *seed=None*, *originPath=None*, *name=''*, *autoLog=True*)

Class to handle trial sequencing and data storage.

Calls to .next() will fetch the next trial object given to this handler, according to the method specified (random, sequential, fullRandom). Calls will raise a StopIteration error if trials have finished.

See demo_trialHandler.py

The psydat file format is literally just a pickled copy of the TrialHandler object that saved it. You can open it with:

```python
from psychopy.tools.filetools import fromFile
dat = fromFile(path)
```

Then you'll find that *dat* has the following attributes that

Parameters

trialList: a simple list (or flat array) of dictionaries
specifying conditions. This can be imported from an excel/csv file using *importConditions()*

nReps: number of repeats for all conditions

method: *'random'*, 'sequential', or 'fullRandom'
'sequential' obviously presents the conditions in the order they appear in the list. 'random' will result in a shuffle of the conditions on each repeat, but all conditions occur once before the second repeat etc. 'fullRandom' fully randomises the trials across repeats as well, which means you could potentially run all trials of one condition before any trial of another.

dataTypes: (optional) list of names for data storage.
e.g. ['corr','rt','resp']. If not provided then these will be created as needed during calls to *addData()*

extraInfo: A dictionary
This will be stored alongside the data and usually describes the experiment and subject ID, date etc.

seed: an integer
If provided then this fixes the random number generator to use the same pattern of trials, by seeding its startpoint

originPath: a string describing the location of the
script / experiment file path. The psydat file format will store a copy of the experiment if possible. If *originPath==None* is provided here then the TrialHandler will still store a copy of the script where it was created. If *OriginPath==-1* then nothing will be stored.

Attributes (after creation)

.data - a dictionary (or more strictly, a *DataHandler* sub-
class of a dictionary) of numpy arrays, one for each data type stored

.trialList - the original list of dicts, specifying the conditions

.thisIndex - the index of the current trial in the original
conditions list

.nTotal - the total number of trials that will be run

.nRemaining - the total number of trials remaining

.thisN - total trials completed so far

.thisRepN - which repeat you are currently on

.thisTrialN - which trial number *within* that repeat

.thisTrial - a dictionary giving the parameters of the current
trial

.finished - True/False for have we finished yet

.extraInfo - the dictionary of extra info as given at beginning

.origin - the contents of the script or builder experiment that
created the handler

`_createOutputArray`(*stimOut*, *dataOut*, *delim=None*, *matrixOnly=False*)

Does the leg-work for saveAsText and saveAsExcel. Combines stimOut with ._parseDataOutput()

`_createOutputArrayData`(*dataOut*)

This just creates the dataOut part of the output matrix. It is called by _createOutputArray() which creates the header line and adds the stimOut columns

`_createSequence`()

Pre-generates the sequence of trial presentations (for non-adaptive methods). This is called automatically when the TrialHandler is initialised so doesn't need an explicit call from the user.

The returned sequence has form indices[stimN][repN] Example: sequential with 6 trialtypes (rows), 5 reps (cols), returns:

```
[[0 0 0 0 0]
[1 1 1 1 1]
[2 2 2 2 2]
[3 3 3 3 3]
[4 4 4 4 4]
[5 5 5 5 5]]
```

These 30 trials will be returned by .next() in the order:
0, 1, 2, 3, 4, 5, 0, 1, 2, … … 3, 4, 5

To add a new type of sequence (as of v1.65.02): - add the sequence generation code here - adjust "if self.method in […]:" in both __init__ and .next() - adjust allowedVals in experiment.py -> shows up in DlgLoopProperties Note that users can make any sequence whatsoever outside of PsychoPy, and specify sequential order; any order is possible this way.

`_makeIndices`(*inputArray*)

Creates an array of tuples the same shape as the input array where each tuple contains the indices to itself in the array.

Useful for shuffling and then using as a reference.

`_terminate`()

Remove references to ourself in experiments and terminate the loop

`addData`(*thisType*, *value*, *position=None*)

Add data for the current trial

`getCurrentTrial`()

Returns the condition for the current trial, without advancing the trials.

`getEarlierTrial`(*n=-1*)

Returns the condition information from n trials previously. Useful for comparisons in n-back tasks. Returns 'None' if trying to access a trial prior to the first.

`getExp`()

Return the ExperimentHandler that this handler is attached to, if any. Returns None if not attached

getFutureTrial(*n=1*)

Returns the condition for n trials into the future, without advancing the trials. A negative n returns a previous (past) trial. Returns 'None' if attempting to go beyond the last trial.

getOriginPathAndFile(*originPath=None*)

Attempts to determine the path of the script that created this data file and returns both the path to that script and its contents. Useful to store the entire experiment with the data.

If originPath is provided (e.g. from Builder) then this is used otherwise the calling script is the originPath (fine from a standard python script).

next()

Advances to next trial and returns it. Updates attributes; thisTrial, thisTrialN and thisIndex If the trials have ended this method will raise a StopIteration error. This can be handled with code such as:

```
trials = data.TrialHandler(.......)
for eachTrial in trials:  # automatically stops when done
    # do stuff
```

or:

```
trials = data.TrialHandler(.......)
while True:  # ie forever
    try:
        thisTrial = trials.next()
    except StopIteration:  # we got a StopIteration error
        break #break out of the forever loop
    # do stuff here for the trial
```

printAsText(*stimOut=None*, *dataOut=('all_mean', 'all_std', 'all_raw')*, *delim='\t'*, *matrixOnly=False*)

Exactly like saveAsText() except that the output goes to the screen instead of a file

saveAsExcel(*fileName*, *sheetName='rawData'*, *stimOut=None*, *dataOut=('n', 'all_mean', 'all_std', 'all_raw')*, *matrixOnly=False*, *appendFile=True*, *fileCollisionMethod='rename'*)

Save a summary data file in Excel OpenXML format workbook (*xlsx*) for processing in most spreadsheet packages. This format is compatible with versions of Excel (2007 or greater) and and with OpenOffice (>=3.0).

It has the advantage over the simpler text files (see `TrialHandler.saveAsText()`) that data can be stored in multiple named sheets within the file. So you could have a single file named after your experiment and then have one worksheet for each participant. Or you could have one file for each participant and then multiple sheets for repeated sessions etc.

The file extension *.xlsx* will be added if not given already.

Parameters

fileName: string
the name of the file to create or append. Can include relative or absolute path

sheetName: string
the name of the worksheet within the file

stimOut: list of strings
the attributes of the trial characteristics to be output. To use this you need to have provided a list of dictionaries specifying to trialList parameter of the TrialHandler and give here the names of strings specifying entries in that dictionary

dataOut: list of strings

specifying the dataType and the analysis to be performed, in the form *dataType_analysis*. The data can be any of the types that you added using trialHandler.data.add() and the analysis can be either 'raw' or most things in the numpy library, including 'mean','std','median','max','min'. e.g. *rt_max* will give a column of max reaction times across the trials assuming that *rt* values have been stored. The default values will output the raw, mean and std of all datatypes found.

appendFile: True or False

If False any existing file with this name will be kept and a new file will be created with a slightly different name. If you want to overwrite the old file, pass 'overwrite' to `fileCollisionMethod`. If True then a new worksheet will be appended. If a worksheet already exists with that name a number will be added to make it unique.

fileCollisionMethod: string

Collision method (`rename`,``overwrite``, `fail`) passed to `handleFileCollision()` This is ignored if `append` is `True`.

`saveAsJson`(*fileName=None*, *encoding='utf-8'*, *fileCollisionMethod='rename'*)

Serialize the object to the JSON format.

Parameters

- **fileName** (*string, or None*) – the name of the file to create or append. Can include a relative or absolute path. If *None*, will not write to a file, but return an in-memory JSON object.
- **encoding** (*string, optional*) – The encoding to use when writing the file.
- **fileCollisionMethod** (*string*) – Collision method passed to `handleFileCollision()`. Can be either of *'rename'*, *'overwrite'*, or *'fail'*.

Notes

Currently, a copy of the object is created, and the copy's .origin attribute is set to an empty string before serializing because loading the created JSON file would sometimes fail otherwise.

`saveAsPickle`(*fileName*, *fileCollisionMethod='rename'*)

Basically just saves a copy of the handler (with data) to a pickle file.

This can be reloaded if necessary and further analyses carried out.

Parameters

fileCollisionMethod: Collision method passed to `handleFileCollision()`

`saveAsText`(*fileName*, *stimOut=None*, *dataOut=('n', 'all_mean', 'all_std', 'all_raw')*, *delim=None*, *matrixOnly=False*, *appendFile=True*, *summarised=True*, *fileCollisionMethod='rename'*, *encoding='utf-8-sig'*)

Write a text file with the data and various chosen stimulus attributes

Parameters

fileName:

will have .tsv appended and can include path info.

stimOut:

the stimulus attributes to be output. To use this you need to use a list of dictionaries and give here the names of dictionary keys that you want as strings

dataOut:
a list of strings specifying the dataType and the analysis to be performed,in the form *dataType_analysis*. The data can be any of the types that you added using trialHandler.data.add() and the analysis can be either 'raw' or most things in the numpy library, including; 'mean','std','median','max','min'… The default values will output the raw, mean and std of all datatypes found

delim:
allows the user to use a delimiter other than tab ("," is popular with file extension ".csv")

matrixOnly:
outputs the data with no header row or extraInfo attached

appendFile:
will add this output to the end of the specified file if it already exists

fileCollisionMethod:
Collision method passed to `handleFileCollision()`

encoding:
The encoding to use when saving a the file. Defaults to *utf-8-sig*.

saveAsWideText(*fileName*, *delim=None*, *matrixOnly=False*, *appendFile=True*, *encoding='utf-8-sig'*, *fileCollisionMethod='rename'*)

Write a text file with the session, stimulus, and data values from each trial in chronological order. Also, return a pandas DataFrame containing same information as the file.

That is, unlike 'saveAsText' and 'saveAsExcel':

- each row comprises information from only a single trial.
- no summarizing is done (such as collapsing to produce mean and standard deviation values across trials).

This 'wide' format, as expected by R for creating dataframes, and various other analysis programs, means that some information must be repeated on every row.

In particular, if the trialHandler's 'extraInfo' exists, then each entry in there occurs in every row. In builder, this will include any entries in the 'Experiment info' field of the 'Experiment settings' dialog. In Coder, this information can be set using something like:

```
myTrialHandler.extraInfo = {'SubjID': 'Joan Smith',
                            'Group': 'Control'}
```

Parameters

fileName:
if extension is not specified, '.csv' will be appended if the delimiter is ',', else '.tsv' will be appended. Can include path info.

delim:
allows the user to use a delimiter other than the default tab ("," is popular with file extension ".csv")

matrixOnly:
outputs the data with no header row.

appendFile:
will add this output to the end of the specified file if it already exists.

fileCollisionMethod:
Collision method passed to `handleFileCollision()`

encoding:
The encoding to use when saving a the file. Defaults to *utf-8-sig*.

setExp(*exp*)

Sets the ExperimentHandler that this handler is attached to

Do NOT attempt to set the experiment using:

```
trials._exp = myExperiment
```

because it needs to be performed using the *weakref* module.

11.13.3 TrialHandler2

class psychopy.data.**TrialHandler2**(*trialList*, *nReps*, *method='random'*, *dataTypes=None*, *extraInfo=None*, *seed=None*, *originPath=None*, *name=''*, *autoLog=True*)

Class to handle trial sequencing and data storage.

Calls to .next() will fetch the next trial object given to this handler, according to the method specified (random, sequential, fullRandom). Calls will raise a StopIteration error if trials have finished.

See demo_trialHandler.py

The psydat file format is literally just a pickled copy of the TrialHandler object that saved it. You can open it with:

```python
from psychopy.tools.filetools import fromFile
dat = fromFile(path)
```

Then you'll find that *dat* has the following attributes that

Parameters

trialList: filename or a simple list (or flat array) of
dictionaries specifying conditions

nReps: number of repeats for all conditions

method: *'random'*, 'sequential', or 'fullRandom'
'sequential' obviously presents the conditions in the order they appear in the list. 'random' will result in a shuffle of the conditions on each repeat, but all conditions occur once before the second repeat etc. 'fullRandom' fully randomises the trials across repeats as well, which means you could potentially run all trials of one condition before any trial of another.

dataTypes: (optional) list of names for data storage.
e.g. ['corr','rt','resp']. If not provided then these will be created as needed during calls to `addData()`

extraInfo: A dictionary
This will be stored alongside the data and usually describes the experiment and subject ID, date etc.

seed: an integer
If provided then this fixes the random number generator to use the same pattern of trials, by seeding its startpoint.

originPath: a string describing the location of the script /
experiment file path. The psydat file format will store a copy of the experiment if possible. If *originPath==None* is provided here then the TrialHandler will still store a copy of the script where it was created. If *OriginPath==-1* then nothing will be stored.

Attributes (after creation)

.data - a dictionary of numpy arrays, one for each data type
stored

.trialList - the original list of dicts, specifying the conditions

.thisIndex - the index of the current trial in the original
conditions list

.nTotal - the total number of trials that will be run

.nRemaining - the total number of trials remaining

.thisN - total trials completed so far

.thisRepN - which repeat you are currently on

.thisTrialN - which trial number *within* that repeat

.thisTrial - a dictionary giving the parameters of the current
trial

.finished - True/False for have we finished yet

.extraInfo - the dictionary of extra info as given at beginning

.origin - the contents of the script or builder experiment that
created the handler

`_createOutputArray`(*stimOut*, *dataOut*, *delim=None*, *matrixOnly=False*)

Does the leg-work for saveAsText and saveAsExcel. Combines stimOut with ._parseDataOutput()

`_createOutputArrayData`(*dataOut*)

This just creates the dataOut part of the output matrix. It is called by _createOutputArray() which creates the header line and adds the stimOut columns

`_terminate`()

Remove references to ourself in experiments and terminate the loop

`abortCurrentTrial`(*action='random'*)

Abort the current trial.

Calling this during an experiment replace this trial. The condition related to the aborted trial will be replaced elsewhere in the session depending on the *method* in use for sampling conditions.

Parameters
action (*str*) – Action to take with the aborted trial. Can be either of *'random'*, or *'append'*. The default action is *'random'*.

Notes

- When using *action='random'*, the RNG state for the trial handler is not used.

`addData`(*thisType*, *value*)

Add a piece of data to the current trial

`calculateUpcoming`(*fromIndex=-1*)

Rebuild the sequence of trial/state info as if running the trials

Parameters
fromIndex (*int, optional*) – the point in the sequnce from where to rebuild. Defaults to -1.

`property data`

Returns a pandas DataFrame of the trial data so far Read only attribute - you can't directly modify TrialHandler.data

Note that data are stored internally as a list of dictionaries, one per trial. These are converted to a DataFrame on access.

`property finished`

Whether this loop has finished or not. Will be True if there are no upcoming trials and False if there are any. Set *.finished = True* to skip all remaining trials (equivalent to calling *.skipTrials()* with a value larger than the number of trials remaining)

Returns
True if there are no upcoming trials, False otherwise.

Return type
bool

`getAllTrials()`

Returns all trials (elapsed, current and upcoming) with an index indicating which trial is the current trial.

Returns

- *list[Trial]* – List of trials, in order (oldest to newest)
- *int* – Index of the current trial in this list

`getCurrentTrial()`

Returns the current trial (*.thisTrial*)

Returns
The current trial

Return type
Trial

`getEarlierTrial(`*n=-1*`)`

Returns the condition information from n trials previously. Useful for comparisons in n-back tasks. Returns 'None' if trying to access a trial prior to the first.

`getExp()`

Return the ExperimentHandler that this handler is attached to, if any. Returns None if not attached

`getFutureTrial(`*n=1*`)`

Returns the condition for n trials into the future, without advancing the trials. Returns 'None' if attempting to go beyond the last trial.

Returns
Trial object for n trials into the future.

Return type
Trial or None

`getFutureTrials(`*n=None*, *start=0*`)`

Returns Trial objects for a given range in the future. Will start looking at *start* trials in the future and will return n trials from then, so e.g. to get all trials from 2 in the future to 5 in the future you would use *start=2* and *n=3*.

Parameters

- **n** (`int, optional`) – How many trials into the future to look, by default None. Leave as None to show all future trials

- **start** (*int, optional*) – How many trials into the future to start looking at, by default 0

Returns
List of Trial objects n long. Any trials beyond the last trial are None.

Return type
list[Trial or None]

getOriginPathAndFile(*originPath=None*)

Attempts to determine the path of the script that created this data file and returns both the path to that script and its contents. Useful to store the entire experiment with the data.

If originPath is provided (e.g. from Builder) then this is used otherwise the calling script is the originPath (fine from a standard python script).

next()

Advances to next trial and returns it. Updates attributes; thisTrial, thisTrialN and thisIndex If the trials have ended this method will raise a StopIteration error. This can be handled with code such as:

```
trials = data.TrialHandler(.......)
for eachTrial in trials:  # automatically stops when done
    # do stuff
```

or:

```
trials = data.TrialHandler(.......)
while True:  # ie forever
    try:
        thisTrial = trials.next()
    except StopIteration:  # we got a StopIteration error
        break  # break out of the forever loop
    # do stuff here for the trial
```

printAsText(*stimOut=None*, *dataOut=('all_mean', 'all_std', 'all_raw')*, *delim='\t'*, *matrixOnly=False*)

Exactly like saveAsText() except that the output goes to the screen instead of a file

rewindTrials(*n=1*)

Rewind back n trials - previously elapsed trials will return to being upcoming. If you try to rewind before the first trial, will log a warning and rewind *to* the first trial.

Parameters
n (*int*) – Number of trials to rewind back

saveAsExcel(*fileName*, *sheetName='rawData'*, *stimOut=None*, *dataOut=('n', 'all_mean', 'all_std', 'all_raw')*, *matrixOnly=False*, *appendFile=True*, *fileCollisionMethod='rename'*)

Save a summary data file in Excel OpenXML format workbook (*xlsx*) for processing in most spreadsheet packages. This format is compatible with versions of Excel (2007 or greater) and and with OpenOffice (>=3.0).

It has the advantage over the simpler text files (see *TrialHandler.saveAsText()*) that data can be stored in multiple named sheets within the file. So you could have a single file named after your experiment and then have one worksheet for each participant. Or you could have one file for each participant and then multiple sheets for repeated sessions etc.

The file extension *.xlsx* will be added if not given already.

Parameters

fileName: string
the name of the file to create or append. Can include relative or absolute path

sheetName: string
the name of the worksheet within the file

stimOut: list of strings
the attributes of the trial characteristics to be output. To use this you need to have provided a list of dictionaries specifying to trialList parameter of the TrialHandler and give here the names of strings specifying entries in that dictionary

dataOut: list of strings
specifying the dataType and the analysis to be performed, in the form *dataType_analysis*. The data can be any of the types that you added using trialHandler.data.add() and the analysis can be either 'raw' or most things in the numpy library, including 'mean','std','median','max','min'. e.g. *rt_max* will give a column of max reaction times across the trials assuming that *rt* values have been stored. The default values will output the raw, mean and std of all datatypes found.

appendFile: True or False
If False any existing file with this name will be kept and a new file will be created with a slightly different name. If you want to overwrite the old file, pass 'overwrite' to `fileCollisionMethod`. If True then a new worksheet will be appended. If a worksheet already exists with that name a number will be added to make it unique.

fileCollisionMethod: string
Collision method (`rename`,``overwrite``, `fail`) passed to `handleFileCollision()` This is ignored if `append` is `True`.

`saveAsJson`(*fileName=None*, *encoding='utf-8'*, *fileCollisionMethod='rename'*)

Serialize the object to the JSON format.

Parameters

- **fileName** (*string, or None*) – the name of the file to create or append. Can include a relative or absolute path. If *None*, will not write to a file, but return an in-memory JSON object.
- **encoding** (*string, optional*) – The encoding to use when writing the file.
- **fileCollisionMethod** (*string*) – Collision method passed to `handleFileCollision()`. Can be either of *'rename'*, *'overwrite'*, or *'fail'*.

Notes

Currently, a copy of the object is created, and the copy's .origin attribute is set to an empty string before serializing because loading the created JSON file would sometimes fail otherwise.

The RNG self._rng cannot be serialized as-is, so we store its state in self._rng_state so we can restore it when loading.

`saveAsPickle`(*fileName*, *fileCollisionMethod='rename'*)

Basically just saves a copy of the handler (with data) to a pickle file.

This can be reloaded if necessary and further analyses carried out.

Parameters
fileCollisionMethod: Collision method passed to `handleFileCollision()`

`saveAsText`(*fileName*, *stimOut=None*, *dataOut=('n', 'all_mean', 'all_std', 'all_raw')*, *delim=None*, *matrixOnly=False*, *appendFile=True*, *summarised=True*, *fileCollisionMethod='rename'*, *encoding='utf-8-sig'*)

Write a text file with the data and various chosen stimulus attributes

Parameters

fileName:
will have .tsv appended and can include path info.

stimOut:
the stimulus attributes to be output. To use this you need to use a list of dictionaries and give here the names of dictionary keys that you want as strings

dataOut:
a list of strings specifying the dataType and the analysis to be performed,in the form *dataType_analysis*. The data can be any of the types that you added using trialHandler.data.add() and the analysis can be either 'raw' or most things in the numpy library, including; 'mean','std','median','max','min'… The default values will output the raw, mean and std of all datatypes found

delim:
allows the user to use a delimiter other than tab ("," is popular with file extension ".csv")

matrixOnly:
outputs the data with no header row or extraInfo attached

appendFile:
will add this output to the end of the specified file if it already exists

fileCollisionMethod:
Collision method passed to `handleFileCollision()`

encoding:
The encoding to use when saving a the file. Defaults to *utf-8-sig*.

`saveAsWideText`(*fileName*, *delim=None*, *matrixOnly=False*, *appendFile=True*, *encoding='utf-8-sig'*, *fileCollisionMethod='rename'*)

Write a text file with the session, stimulus, and data values from each trial in chronological order. Also, return a pandas DataFrame containing same information as the file.

That is, unlike 'saveAsText' and 'saveAsExcel':

- each row comprises information from only a single trial.
- no summarising is done (such as collapsing to produce mean and standard deviation values across trials).

This 'wide' format, as expected by R for creating dataframes, and various other analysis programs, means that some information must be repeated on every row.

In particular, if the trialHandler's 'extraInfo' exists, then each entry in there occurs in every row. In builder, this will include any entries in the 'Experiment info' field of the 'Experiment settings' dialog. In Coder, this information can be set using something like:

```
myTrialHandler.extraInfo = {'SubjID': 'Joan Smith',
                            'Group': 'Control'}
```

Parameters

fileName:
if extension is not specified, '.csv' will be appended if the delimiter is ',', else '.tsv' will be appended. Can include path info.

delim:
allows the user to use a delimiter other than the default tab ("," is popular with file extension ".csv")

matrixOnly:
outputs the data with no header row.

appendFile:
will add this output to the end of the specified file if it already exists.

fileCollisionMethod:
Collision method passed to `handleFileCollision()`

encoding:
The encoding to use when saving a the file. Defaults to *utf-8-sig*.

setExp(*exp*)

Sets the ExperimentHandler that this handler is attached to

Do NOT attempt to set the experiment using:

```
trials._exp = myExperiment
```

because it needs to be performed using the *weakref* module.

skipTrials(*n=1*)

Skip ahead n trials - the trials inbetween will be marked as "skipped". If you try to skip past the last trial, will log a warning and skip *to* the last trial.

Parameters
n (*int*) – Number of trials to skip ahead

property thisIndex

property thisN

property thisRepN

property thisTrialN

11.13.4 TrialHandlerExt

class psychopy.data.**TrialHandlerExt**(*trialList*, *nReps*, *method='random'*, *dataTypes=None*, *extraInfo=None*, *seed=None*, *originPath=None*, *name=''*, *autoLog=True*)

A class for handling trial sequences in a *non-counterbalanced design* (i.e. *oddball paradigms*). Its functions are a superset of the class TrialHandler, and as such, can also be used for normal trial handling.

TrialHandlerExt has the same function names for data storage facilities.

To use non-counterbalanced designs, all TrialType dict entries in the trial list must have a key called "weight". For example, if you want trial types A, B, C, and D to have 10, 5, 3, and 2 repetitions per block, then the trialList can look like:

[{Name:'A', . . . , weight:10},
{Name:'B', . . . , weight:5}, {Name:'C', . . . , weight:3}, {Name:'D', . . . , weight:2}]

For experimenters using an excel or csv file for trial list, a column called weight is appropriate for this purpose.

Calls to .next() will fetch the next trial object given to this handler, according to the method specified (random, sequential, fullRandom). Calls will raise a StopIteration error when all trials are exhausted.

Authored by Suddha Sourav at BPN, Uni Hamburg - heavily borrowing from the TrialHandler class

Parameters

trialList: a simple list (or flat array) of dictionaries
specifying conditions. This can be imported from an excel / csv file using `importConditions()` For non-counterbalanced designs, each dict entry in trialList must have a key called weight!

nReps: number of repeats for all conditions. When using a
non-counterbalanced design, nReps is analogous to the number of blocks.

method: *'random',* 'sequential', or 'fullRandom'
When the weights are not specified: 'sequential' presents the conditions in the order they appear in the list. 'random' will result in a shuffle of the conditions on each repeat, but all conditions occur once before the second repeat etc. 'fullRandom' fully randomises the trials across repeats as well, which means you could potentially run all trials of one condition before any trial of another.

In the presence of weights: 'sequential' presents each trial type the number of times specified by its weight, before moving on to the next type. 'random' randomizes the presentation order within block. 'fulLRandom' shuffles trial order across weights an nRep, that is, a full shuffling.

dataTypes: (optional) list of names for data storage. e.g.
['corr','rt','resp']. If not provided then these will be created as needed during calls to `addData()`

extraInfo: A dictionary
This will be stored alongside the data and usually describes the experiment and subject ID, date etc.

seed: an integer
If provided then this fixes the random number generator to use the same pattern of trials, by seeding its startpoint

originPath: a string describing the location of the script /
experiment file path. The psydat file format will store a copy of the experiment if possible. If *originPath==None* is provided here then the TrialHandler will still store a copy of the script where it was created. If *OriginPath==-1* then nothing will be stored.

Attributes (after creation)

.data - a dictionary of numpy arrays, one for each data type
stored

.trialList - the original list of dicts, specifying the conditions

.thisIndex - the index of the current trial in the original
conditions list

.nTotal - the total number of trials that will be run

.nRemaining - the total number of trials remaining

.thisN - total trials completed so far

.thisRepN - which repeat you are currently on

.thisTrialN - which trial number *within* that repeat

.thisTrial - a dictionary giving the parameters of the current
trial

.finished - True/False for have we finished yet

.extraInfo - the dictionary of extra info as given at beginning

.origin - the contents of the script or builder experiment that
created the handler

.trialWeights - None if all weights are not specified. If all
weights are specified, then a list containing the weights of the trial types.

`_createOutputArray`(*stimOut*, *dataOut*, *delim=None*, *matrixOnly=False*)

Does the leg-work for saveAsText and saveAsExcel. Combines stimOut with ._parseDataOutput()

`_createOutputArrayData`(*dataOut*)

This just creates the dataOut part of the output matrix. It is called by _createOutputArray() which creates the header line and adds the stimOut columns

`_createSequence`()

Pre-generates the sequence of trial presentations (for non-adaptive methods). This is called automatically when the TrialHandler is initialised so doesn't need an explicit call from the user.

The returned sequence has form indices[stimN][repN] Example: sequential with 6 trialtypes (rows), 5 reps (cols), returns:

```
[[0 0 0 0 0]
[1 1 1 1 1]
[2 2 2 2 2]
[3 3 3 3 3]
[4 4 4 4 4]
[5 5 5 5 5]]
```

These 30 trials will be returned by .next() in the order:
0, 1, 2, 3, 4, 5, 0, 1, 2, … … 3, 4, 5

Example: random, with 3 trialtypes, where the weights of conditions 0,1, and 2 are 3,2, and 1 respectively, and a rep value of 5, might return:

```
[[0 1 2 0 1]
[1 0 1 1 1]
[0 2 0 0 0]
[0 0 0 1 0]
[2 0 1 0 2]
[1 1 0 2 0]]
```

These 30 trials will be returned by .next() in the order:
0, 1, 0, 0, 2, 1, 1, 0, 2, 0, 0, 1, … … 0, 2, 0 *stopIteration*

To add a new type of sequence (as of v1.65.02): - add the sequence generation code here - adjust "if self.method in […]:" in both __init__ and .next() - adjust allowedVals in experiment.py -> shows up in DlgLoopProperties Note that users can make any sequence whatsoever outside of PsychoPy, and specify sequential order; any order is possible this way.

`_makeIndices`(*inputArray*)

Creates an array of tuples the same shape as the input array where each tuple contains the indices to itself in the array.

Useful for shuffling and then using as a reference.

_terminate()

Remove references to ourself in experiments and terminate the loop

addData(*thisType*, *value*, *position=None*)

Add data for the current trial

getCurrentTrial()

Returns the condition for the current trial, without advancing the trials.

getCurrentTrialPosInDataHandler()

getEarlierTrial(*n=-1*)

Returns the condition information from n trials previously. Useful for comparisons in n-back tasks. Returns 'None' if trying to access a trial prior to the first.

getExp()

Return the ExperimentHandler that this handler is attached to, if any. Returns None if not attached

getFutureTrial(*n=1*)

Returns the condition for n trials into the future, without advancing the trials. A negative n returns a previous (past) trial. Returns 'None' if attempting to go beyond the last trial.

getNextTrialPosInDataHandler()

getOriginPathAndFile(*originPath=None*)

Attempts to determine the path of the script that created this data file and returns both the path to that script and its contents. Useful to store the entire experiment with the data.

If originPath is provided (e.g. from Builder) then this is used otherwise the calling script is the originPath (fine from a standard python script).

next()

Advances to next trial and returns it. Updates attributes; thisTrial, thisTrialN and thisIndex If the trials have ended this method will raise a StopIteration error. This can be handled with code such as:

```python
trials = data.TrialHandler(.......)
for eachTrial in trials:  # automatically stops when done
    # do stuff
```

or:

```python
trials = data.TrialHandler(.......)
while True:  # ie forever
    try:
        thisTrial = trials.next()
    except StopIteration:  # we got a StopIteration error
        break  # break out of the forever loop
    # do stuff here for the trial
```

printAsText(*stimOut=None*, *dataOut=('all_mean', 'all_std', 'all_raw')*, *delim='\t'*, *matrixOnly=False*)

Exactly like saveAsText() except that the output goes to the screen instead of a file

saveAsExcel(*fileName*, *sheetName='rawData'*, *stimOut=None*, *dataOut=('n', 'all_mean', 'all_std', 'all_raw')*, *matrixOnly=False*, *appendFile=True*, *fileCollisionMethod='rename'*)

Save a summary data file in Excel OpenXML format workbook (*xlsx*) for processing in most spreadsheet packages. This format is compatible with versions of Excel (2007 or greater) and and with OpenOffice (>=3.0).

It has the advantage over the simpler text files (see `TrialHandler.saveAsText()`) that data can be stored in multiple named sheets within the file. So you could have a single file named after your experiment and then have one worksheet for each participant. Or you could have one file for each participant and then multiple sheets for repeated sessions etc.

The file extension *.xlsx* will be added if not given already.

Parameters

fileName: string
the name of the file to create or append. Can include relative or absolute path

sheetName: string
the name of the worksheet within the file

stimOut: list of strings
the attributes of the trial characteristics to be output. To use this you need to have provided a list of dictionaries specifying to trialList parameter of the TrialHandler and give here the names of strings specifying entries in that dictionary

dataOut: list of strings
specifying the dataType and the analysis to be performed, in the form *dataType_analysis*. The data can be any of the types that you added using trialHandler.data.add() and the analysis can be either 'raw' or most things in the numpy library, including 'mean','std','median','max','min'. e.g. *rt_max* will give a column of max reaction times across the trials assuming that *rt* values have been stored. The default values will output the raw, mean and std of all datatypes found.

appendFile: True or False
If False any existing file with this name will be kept and a new file will be created with a slightly different name. If you want to overwrite the old file, pass 'overwrite' to `fileCollisionMethod`. If True then a new worksheet will be appended. If a worksheet already exists with that name a number will be added to make it unique.

fileCollisionMethod: string
Collision method (`rename`,``overwrite``, `fail`) passed to `handleFileCollision()` This is ignored if `append` is `True`.

saveAsJson(*fileName=None*, *encoding='utf-8'*, *fileCollisionMethod='rename'*)

Serialize the object to the JSON format.

Parameters

- **fileName** (*string, or None*) – the name of the file to create or append. Can include a relative or absolute path. If *None*, will not write to a file, but return an in-memory JSON object.
- **encoding** (*string, optional*) – The encoding to use when writing the file.
- **fileCollisionMethod** (*string*) – Collision method passed to `handleFileCollision()`. Can be either of *'rename'*, *'overwrite'*, or *'fail'*.

Notes

Currently, a copy of the object is created, and the copy's .origin attribute is set to an empty string before serializing because loading the created JSON file would sometimes fail otherwise.

saveAsPickle(*fileName*, *fileCollisionMethod='rename'*)

Basically just saves a copy of the handler (with data) to a pickle file.

This can be reloaded if necessary and further analyses carried out.

Parameters
fileCollisionMethod: Collision method passed to `handleFileCollision()`

`saveAsText`(*fileName*, *stimOut=None*, *dataOut=('n', 'all_mean', 'all_std', 'all_raw')*, *delim=None*, *matrixOnly=False*, *appendFile=True*, *summarised=True*, *fileCollisionMethod='rename'*, *encoding='utf-8-sig'*)

Write a text file with the data and various chosen stimulus attributes

Parameters

fileName:
will have .tsv appended and can include path info.

stimOut:
the stimulus attributes to be output. To use this you need to use a list of dictionaries and give here the names of dictionary keys that you want as strings

dataOut:
a list of strings specifying the dataType and the analysis to be performed,in the form *dataType_analysis*. The data can be any of the types that you added using trialHandler.data.add() and the analysis can be either 'raw' or most things in the numpy library, including; 'mean','std','median','max','min'… The default values will output the raw, mean and std of all datatypes found

delim:
allows the user to use a delimiter other than tab ("," is popular with file extension ".csv")

matrixOnly:
outputs the data with no header row or extraInfo attached

appendFile:
will add this output to the end of the specified file if it already exists

fileCollisionMethod:
Collision method passed to `handleFileCollision()`

encoding:
The encoding to use when saving a the file. Defaults to *utf-8-sig*.

`saveAsWideText`(*fileName*, *delim='\t'*, *matrixOnly=False*, *appendFile=True*, *encoding='utf-8-sig'*, *fileCollisionMethod='rename'*)

Write a text file with the session, stimulus, and data values from each trial in chronological order.

That is, unlike 'saveAsText' and 'saveAsExcel':

- each row comprises information from only a single trial.
- no summarizing is done (such as collapsing to produce mean and standard deviation values across trials).

This 'wide' format, as expected by R for creating dataframes, and various other analysis programs, means that some information must be repeated on every row.

In particular, if the trialHandler's 'extraInfo' exists, then each entry in there occurs in every row. In builder, this will include any entries in the 'Experiment info' field of the 'Experiment settings' dialog. In Coder, this information can be set using something like:

```
myTrialHandler.extraInfo = {'SubjID':'Joan Smith',
                            'Group':'Control'}
```

Parameters

fileName:
if extension is not specified, '.csv' will be appended if the delimiter is ',', else '.txt' will be appended. Can include path info.

delim:
allows the user to use a delimiter other than the default tab ("," is popular with file extension ".csv")

matrixOnly:
outputs the data with no header row.

appendFile:
will add this output to the end of the specified file if it already exists.

fileCollisionMethod:
Collision method passed to `handleFileCollision()`

encoding:
The encoding to use when saving a the file. Defaults to *utf-8-sig*.

setExp(*exp*)

Sets the ExperimentHandler that this handler is attached to

Do NOT attempt to set the experiment using:

```
trials._exp = myExperiment
```

because it needs to be performed using the *weakref* module.

11.13.5 StairHandler

class psychopy.data.**StairHandler**(*startVal*, *nReversals=None*, *stepSizes=4*, *nTrials=0*, *nUp=1*, *nDown=3*, *applyInitialRule=True*, *extraInfo=None*, *method='2AFC'*, *stepType='db'*, *minVal=None*, *maxVal=None*, *originPath=None*, *name=''*, *autoLog=True*, ***kwargs*)

Class to handle smoothly the selection of the next trial and report current values etc. Calls to next() will fetch the next object given to this handler, according to the method specified.

See `Demos >> ExperimentalControl >> JND_staircase_exp.py`

The staircase will terminate when *nTrials* AND *nReversals* have been exceeded. If *stepSizes* was an array and has been exceeded before nTrials is exceeded then the staircase will continue to reverse.

nUp and *nDown* are always considered as 1 until the first reversal is reached. The values entered as arguments are then used.

Parameters

startVal:
The initial value for the staircase.

nReversals:
The minimum number of reversals permitted. If *stepSizes* is a list, but the minimum number of reversals to perform, *nReversals*, is less than the length of this list, PsychoPy will automatically increase the minimum number of reversals and emit a warning. This minimum number of reversals is always set to be greater than 0.

stepSizes:
The size of steps as a single value or a list (or array). For a single value the step size is fixed. For an array or list the step size will progress to the next entry at each reversal.

nTrials:
The minimum number of trials to be conducted. If the staircase has not reached the required number of reversals then it will continue.

nUp:
The number of 'incorrect' (or 0) responses before the staircase level increases.

nDown:
The number of 'correct' (or 1) responses before the staircase level decreases.

applyInitialRule
[bool] Whether to apply a 1-up/1-down rule until the first reversal point (if *True*), before switching to the specified up/down rule.

extraInfo:
A dictionary (typically) that will be stored along with collected data using `saveAsPickle()` or `saveAsText()` methods.

method:
Not used and may be deprecated in future releases.

stepType: *'db'*, 'lin', 'log'
The type of steps that should be taken each time. 'lin' will simply add or subtract that amount each step, 'db' and 'log' will step by a certain number of decibels or log units (note that this will prevent your value ever reaching zero or less)

minVal: *None*, or a number
The smallest legal value for the staircase, which can be used to prevent it reaching impossible contrast values, for instance.

maxVal: *None*, or a number
The largest legal value for the staircase, which can be used to prevent it reaching impossible contrast values, for instance.

Additional keyword arguments will be ignored.

Notes

The additional keyword arguments ***kwargs* might for example be passed by the *MultiStairHandler*, which expects a *label* keyword for each staircase. These parameters are to be ignored by the StairHandler.

`_intensityDec`()

decrement the current intensity and reset counter

`_intensityInc`()

increment the current intensity and reset counter

`_terminate`()

Remove references to ourself in experiments and terminate the loop

`addData`(*result*, *intensity=None*)

Deprecated since 1.79.00: This function name was ambiguous. Please use one of these instead:

- .addResponse(result, intensity)
- .addOtherData('dataName', value')

`addOtherData`(*dataName*, *value*)

Add additional data to the handler, to be tracked alongside the result data but not affecting the value of the staircase `addResponse`(*result*, *intensity=None*)

Add a 1 or 0 to signify a correct / detected or incorrect / missed trial.

This is essential to advance the staircase to a new intensity level!

Supplying an *intensity* value here indicates that you did not use the recommended intensity in your last trial and the staircase will replace its recorded value with the one you supplied here.

`calculateNextIntensity`()

Based on current intensity, counter of correct responses, and current direction.

`getExp`()

Return the ExperimentHandler that this handler is attached to, if any. Returns None if not attached

`getOriginPathAndFile`(*originPath=None*)

Attempts to determine the path of the script that created this data file and returns both the path to that script and its contents. Useful to store the entire experiment with the data.

If originPath is provided (e.g. from Builder) then this is used otherwise the calling script is the originPath (fine from a standard python script).

`property intensity`

The intensity (level) of the current staircase

`next`()

Advances to next trial and returns it. Updates attributes; *thisTrial*, *thisTrialN* and *thisIndex*.

If the trials have ended, calling this method will raise a StopIteration error. This can be handled with code such as:

```
staircase = data.StairHandler(.......)
for eachTrial in staircase:  # automatically stops when done
    # do stuff
```

or:

```
staircase = data.StairHandler(.......)
while True:  # ie forever
    try:
        thisTrial = staircase.next()
    except StopIteration:  # we got a StopIteration error
        break  # break out of the forever loop
    # do stuff here for the trial
```

`printAsText`(*stimOut=None*, *dataOut=('all_mean', 'all_std', 'all_raw')*, *delim='\t'*, *matrixOnly=False*)

Exactly like saveAsText() except that the output goes to the screen instead of a file

`saveAsExcel`(*fileName*, *sheetName='data'*, *matrixOnly=False*, *appendFile=True*, *fileCollisionMethod='rename'*)

Save a summary data file in Excel OpenXML format workbook (*xlsx*) for processing in most spreadsheet packages. This format is compatible with versions of Excel (2007 or greater) and and with OpenOffice (>=3.0).

It has the advantage over the simpler text files (see `TrialHandler.saveAsText()`) that data can be stored in multiple named sheets within the file. So you could have a single file named after your experiment and then have one worksheet for each participant. Or you could have one file for each participant and then multiple sheets for repeated sessions etc.

The file extension *.xlsx* will be added if not given already.

The file will contain a set of values specifying the staircase level ('intensity') at each reversal, a list of reversal indices (trial numbers), the raw staircase / intensity level on *every* trial and the corresponding responses of the participant on every trial.

Parameters

fileName: string
the name of the file to create or append. Can include relative or absolute path.

sheetName: string
the name of the worksheet within the file

matrixOnly: True or False
If set to True then only the data itself will be output (no additional info)

appendFile: True or False
If False any existing file with this name will be overwritten. If True then a new worksheet will be appended. If a worksheet already exists with that name a number will be added to make it unique.

fileCollisionMethod: string
Collision method passed to `handleFileCollision()` This is ignored if `appendFile` is `True`.

`saveAsJson`(*fileName=None*, *encoding='utf-8-sig'*, *fileCollisionMethod='rename'*)

Serialize the object to the JSON format.

Parameters

- **fileName** (*string, or None*) – the name of the file to create or append. Can include a relative or absolute path. If *None*, will not write to a file, but return an in-memory JSON object.
- **encoding** (*string, optional*) – The encoding to use when writing the file.
- **fileCollisionMethod** (*string*) – Collision method passed to `handleFileCollision()`. Can be either of *'rename'*, *'overwrite'*, or *'fail'*.

Notes

Currently, a copy of the object is created, and the copy's .origin attribute is set to an empty string before serializing because loading the created JSON file would sometimes fail otherwise.

`saveAsPickle`(*fileName*, *fileCollisionMethod='rename'*)

Basically just saves a copy of self (with data) to a pickle file.

This can be reloaded if necessary and further analyses carried out.

Parameters
fileCollisionMethod: Collision method passed to `handleFileCollision()`

`saveAsText`(*fileName*, *delim=None*, *matrixOnly=False*, *fileCollisionMethod='rename'*, *encoding='utf-8-sig'*)

Write a text file with the data

Parameters

fileName: a string
The name of the file, including path if needed. The extension *.tsv* will be added if not included.

delim: a string
the delimitter to be used (e.g. ‘ ‘ for tab-delimitted, ‘,’ for csv files)

matrixOnly: True/False
If True, prevents the output of the *extraInfo* provided at initialisation.

fileCollisionMethod:
Collision method passed to `handleFileCollision()`

encoding:
The encoding to use when saving a the file. Defaults to *utf-8-sig*.

`setExp`(*exp*)

Sets the ExperimentHandler that this handler is attached to

Do NOT attempt to set the experiment using:

```
trials._exp = myExperiment
```

because it needs to be performed using the *weakref* module.

11.13.6 PsiHandler

class `psychopy.data.`**`PsiHandler`**(*nTrials*, *intensRange*, *alphaRange*, *betaRange*, *intensPrecision*, *alphaPrecision*, *betaPrecision*, *delta*, *stepType='lin'*, *expectedMin=0.5*, *prior=None*, *fromFile=False*, *extraInfo=None*, *name=''*)

Handler to implement the “Psi” adaptive psychophysical method (Kontsevich & Tyler, 1999).

This implementation assumes the form of the psychometric function to be a cumulative Gaussian. Psi estimates the two free parameters of the psychometric function, the location (alpha) and slope (beta), using Bayes’ rule and grid approximation of the posterior distribution. It chooses stimuli to present by minimizing the entropy of this grid. Because this grid is represented internally as a 4-D array, one must choose the intensity, alpha, and beta ranges carefully so as to avoid a Memory Error. Maximum likelihood is used to estimate Lambda, the most likely location/slope pair. Because Psi estimates the entire psychometric function, any threshold defined on the function may be estimated once Lambda is determined.

It is advised that Lambda estimates are examined after completion of the Psi procedure. If the estimated alpha or beta values equal your specified search bounds, then the search range most likely did not contain the true value. In this situation the procedure should be repeated with appropriately adjusted bounds.

Because Psi is a Bayesian method, it can be initialized with a prior from existing research. A function to save the posterior over Lambda as a Numpy binary file is included.

Kontsevich & Tyler (1999) specify their psychometric function in terms of d’. PsiHandler avoids this and treats all parameters with respect to stimulus intensity. Specifically, the forms of the psychometric function assumed for Yes/No and Two Alternative Forced Choice (2AFC) are, respectively:

_normCdf = norm.cdf(x, mean=alpha, sd=beta) Y(x) = .5 * delta + (1 - delta) * _normCdf

Y(x) = .5 * delta + (1 - delta) * (.5 + .5 * _normCdf)

Initializes the handler and creates an internal Psi Object for grid approximation.

Parameters

nTrials (int)
The number of trials to run.

intensRange (list)
Two element list containing the (inclusive) endpoints of the stimuli intensity range.

alphaRange (list)
Two element list containing the (inclusive) endpoints of the alpha (location parameter) range.

betaRange (list)
Two element list containing the (inclusive) endpoints of the beta (slope parameter) range.

intensPrecision (float or int)
If stepType == 'lin', this specifies the step size of the stimuli intensity range. If stepType == 'log', this specifies the number of steps in the stimuli intensity range.

alphaPrecision (float)
The step size of the alpha (location parameter) range.

betaPrecision (float)
The step size of the beta (slope parameter) range.

delta (float)
The guess rate.

stepType (str)
The type of steps to be used when constructing the stimuli intensity range. If 'lin' then evenly spaced steps are used. If 'log' then logarithmically spaced steps are used. Defaults to 'lin'.

expectedMin (float)
The expected lower asymptote of the psychometric function (PMF).

For a Yes/No task, the PMF usually extends across the interval [0, 1]; here, *expectedMin* should be set to *0*.

For a 2-AFC task, the PMF spreads out across [0.5, 1.0]. Therefore, *expectedMin* should be set to *0.5* in this case, and the 2-AFC psychometric function described above going to be is used.

Currently, only Yes/No and 2-AFC designs are supported.

Defaults to 0.5, or a 2-AFC task.

prior (numpy ndarray or str)
Optional prior distribution with which to initialize the Psi Object. This can either be a numpy ndarray object or the path to a numpy binary file (.npy) containing the ndarray.

fromFile (str)
Flag specifying whether prior is a file pathname or not.

extraInfo (dict)
Optional dictionary object used in PsychoPy's built-in logging system.

name (str)
Optional name for the PsiHandler used in PsychoPy's built-in logging system.

Raises

NotImplementedError
If the supplied *minVal* parameter implies an experimental design other than Yes/No or 2-AFC.

`_checkFinished()`

checks if we are finished. Updates attribute: *finished*

`_intensityDec()`

decrement the current intensity and reset counter `_intensityInc()`

increment the current intensity and reset counter

`_terminate()`

Remove references to ourself in experiments and terminate the loop

`addData`(*result*, *intensity=None*)

Deprecated since 1.79.00: This function name was ambiguous. Please use one of these instead:

- .addResponse(result, intensity)
- .addOtherData('dataName', value')

`addOtherData`(*dataName*, *value*)

Add additional data to the handler, to be tracked alongside the result data but not affecting the value of the staircase

`addResponse`(*result*, *intensity=None*)

Add a 1 or 0 to signify a correct / detected or incorrect / missed trial. Supplying an *intensity* value here indicates that you did not use the recommended intensity in your last trial and the staircase will replace its recorded value with the one you supplied here.

`calculateNextIntensity()`

Based on current intensity, counter of correct responses, and current direction.

`estimateLambda()`

Returns a tuple of (location, slope)

`estimateThreshold`(*thresh*, *lamb=None*)

Returns an intensity estimate for the provided probability.

The optional argument 'lamb' allows thresholds to be estimated without having to recompute the maximum likelihood lambda.

`getExp()`

Return the ExperimentHandler that this handler is attached to, if any. Returns None if not attached

`getOriginPathAndFile`(*originPath=None*)

Attempts to determine the path of the script that created this data file and returns both the path to that script and its contents. Useful to store the entire experiment with the data.

If originPath is provided (e.g. from Builder) then this is used otherwise the calling script is the originPath (fine from a standard python script).

`property intensity`

The intensity (level) of the current staircase

`next()`

Advances to next trial and returns it.

`printAsText`(*stimOut=None*, *dataOut=('all_mean', 'all_std', 'all_raw')*, *delim='\t'*, *matrixOnly=False*)

Exactly like saveAsText() except that the output goes to the screen instead of a file

`saveAsExcel`(*fileName*, *sheetName='data'*, *matrixOnly=False*, *appendFile=True*, *fileCollisionMethod='rename'*)

Save a summary data file in Excel OpenXML format workbook (*xlsx*) for processing in most spreadsheet packages. This format is compatible with versions of Excel (2007 or greater) and and with OpenOffice (>=3.0).

It has the advantage over the simpler text files (see `TrialHandler.saveAsText()`) that data can be stored in multiple named sheets within the file. So you could have a single file named after your experiment and then have one worksheet for each participant. Or you could have one file for each participant and then multiple sheets for repeated sessions etc.

The file extension *.xlsx* will be added if not given already.

The file will contain a set of values specifying the staircase level ('intensity') at each reversal, a list of reversal indices (trial numbers), the raw staircase / intensity level on *every* trial and the corresponding responses of the participant on every trial.

Parameters

fileName: string
the name of the file to create or append. Can include relative or absolute path.

sheetName: string
the name of the worksheet within the file

matrixOnly: True or False
If set to True then only the data itself will be output (no additional info)

appendFile: True or False
If False any existing file with this name will be overwritten. If True then a new worksheet will be appended. If a worksheet already exists with that name a number will be added to make it unique.

fileCollisionMethod: string
Collision method passed to `handleFileCollision()` This is ignored if `appendFile` is `True`.

`saveAsJson`(*fileName=None*, *encoding='utf-8-sig'*, *fileCollisionMethod='rename'*)

Serialize the object to the JSON format.

Parameters

- **fileName** (*string, or None*) – the name of the file to create or append. Can include a relative or absolute path. If *None*, will not write to a file, but return an in-memory JSON object.
- **encoding** (*string, optional*) – The encoding to use when writing the file.
- **fileCollisionMethod** (*string*) – Collision method passed to `handleFileCollision()`. Can be either of *'rename'*, *'overwrite'*, or *'fail'*.

Notes

Currently, a copy of the object is created, and the copy's .origin attribute is set to an empty string before serializing because loading the created JSON file would sometimes fail otherwise.

`saveAsPickle`(*fileName*, *fileCollisionMethod='rename'*)

Basically just saves a copy of self (with data) to a pickle file.

This can be reloaded if necessary and further analyses carried out.

Parameters
fileCollisionMethod: Collision method passed to `handleFileCollision()`

`saveAsText`(*fileName*, *delim=None*, *matrixOnly=False*, *fileCollisionMethod='rename'*, *encoding='utf-8-sig'*)

Write a text file with the data

Parameters

fileName: a string
The name of the file, including path if needed. The extension *.tsv* will be added if not included.

delim: a string
the delimitter to be used (e.g. ‘ ‘ for tab-delimitted, ‘,’ for csv files)

matrixOnly: True/False
If True, prevents the output of the *extraInfo* provided at initialisation.

fileCollisionMethod:
Collision method passed to `handleFileCollision()`

encoding:
The encoding to use when saving a the file. Defaults to *utf-8-sig*.

`savePosterior`(*fileName*, *fileCollisionMethod='rename'*)

Saves the posterior array over probLambda as a pickle file with the specified name.

Parameters
`fileCollisionMethod` (*`string`*) – Collision method passed to `handleFileCollision()`

`setExp`(*exp*)

Sets the ExperimentHandler that this handler is attached to

Do NOT attempt to set the experiment using:

```
trials._exp = myExperiment
```

because it needs to be performed using the *weakref* module.

11.13.7 `QuestHandler`

`class psychopy.data.QuestHandler`(*startVal*, *startValSd*, *pThreshold=0.82*, *nTrials=None*, *stopInterval=None*, *method='quantile'*, *beta=3.5*, *delta=0.01*, *gamma=0.5*, *grain=0.01*, *range=None*, *extraInfo=None*, *minVal=None*, *maxVal=None*, *staircase=None*, *originPath=None*, *name=''*, *autoLog=True*, ***kwargs*)

Class that implements the Quest algorithm for quick measurement of psychophysical thresholds.

Uses Andrew Straw’s QUEST, which is a Python port of Denis Pelli’s Matlab code.

Measures threshold using a Weibull psychometric function. Currently, it is not possible to use a different psychometric function.

The Weibull psychometric function is given by the formula

$\Psi(x) = \delta\gamma + (1-\delta)[1-(1-\gamma)\exp(-10^{beta(x-T+\epsilon)})]$

Here, x is an intensity or a contrast (in log10 units), and T is estimated threshold.

Quest internally shifts the psychometric function such that intensity at the user-specified threshold performance level `pThreshold` (e.g., 50% in a yes-no or 75% in a 2-AFC task) is euqal to 0. The parameter ϵ is responsible for this shift, and is determined automatically based on the specified `pThreshold` value. It is the parameter Watson & Pelli (1983) introduced to perform measurements at the “optimal sweat factor”. Assuming your `QuestHandler` instance is called `q`, you can retrieve this value via `q.epsilon`.

Example:

```
# setup display/window
...
# create stimulus
stimulus = visual.RadialStim(win=win, tex='sinXsin', size=1,
                             pos=[0,0], units='deg')
...
# create staircase object
# trying to find out the contrast where subject gets 63% correct
# if wanted to do a 2AFC then the defaults for pThreshold and gamma
# are good. As start value, we'll use 50% contrast, with SD = 20%
staircase = data.QuestHandler(0.5, 0.2,
    pThreshold=0.63, gamma=0.01,
    nTrials=20, minVal=0, maxVal=1)
...
while thisContrast in staircase:
    # setup stimulus
    stimulus.setContrast(thisContrast)
    stimulus.draw()
    win.flip()
    core.wait(0.5)
    # get response
    ...
    # inform QUEST of the response, needed to calculate next level
    staircase.addResponse(thisResp)
...
# can now access 1 of 3 suggested threshold levels
staircase.mean()
staircase.mode()
staircase.quantile(0.5)  # gets the median
```

Typical values for pThreshold are:

- 0.82 which is equivalent to a 3 up 1 down standard staircase
- **0.63 which is equivalent to a 1 up 1 down standard staircase**
  (and might want gamma=0.01)

The variable(s) nTrials and/or stopSd must be specified.

beta, *delta*, and *gamma* are the parameters of the Weibull psychometric function.

Parameters

startVal:
Prior threshold estimate or your initial guess threshold.

startValSd:
Standard deviation of your starting guess threshold. Be generous with the sd as QUEST will have trouble finding the true threshold if it's more than one sd from your initial guess.

pThreshold
Your threshold criterion expressed as probability of response==1. An intensity offset is introduced into the psychometric function so that the threshold (i.e., the midpoint of the table) yields pThreshold.

nTrials: *None* or a number
The maximum number of trials to be conducted.

stopInterval: ***None*** **or a number**
The minimum 5-95% confidence interval required in the threshold estimate before stopping. If both this and nTrials is specified, whichever happens first will determine when Quest will stop.

method: ***'quantile'*****, 'mean', 'mode'**
The method used to determine the next threshold to test. If you want to get a specific threshold level at the end of your staircasing, please use the quantile, mean, and mode methods directly.

beta: ***3.5*** **or a number**
Controls the steepness of the psychometric function.

delta: ***0.01*** **or a number**
The fraction of trials on which the observer presses blindly.

gamma: ***0.5*** **or a number**
The fraction of trials that will generate response 1 when intensity=-Inf.

grain: ***0.01*** **or a number**
The quantization of the internal table.

range: ***None*****, or a number**
The intensity difference between the largest and smallest intensity that the internal table can store. This interval will be centered on the initial guess tGuess. QUEST assumes that intensities outside of this range have zero prior probability (i.e., they are impossible).

extraInfo:
A dictionary (typically) that will be stored along with collected data using *saveAsPickle()* or *saveAsText()* methods.

minVal: ***None*****, or a number**
The smallest legal value for the staircase, which can be used to prevent it reaching impossible contrast values, for instance.

maxVal: ***None*****, or a number**
The largest legal value for the staircase, which can be used to prevent it reaching impossible contrast values, for instance.

staircase: ***None*** **or StairHandler**
Can supply a staircase object with intensities and results. Might be useful to give the quest algorithm more information if you have it. You can also call the importData function directly.

Additional keyword arguments will be ignored.

Notes

The additional keyword arguments ***kwargs* might for example be passed by the *MultiStairHandler*, which expects a *label* keyword for each staircase. These parameters are to be ignored by the StairHandler.

_checkFinished()

checks if we are finished Updates attribute: *finished*

_intensity()

assigns the next intensity level

_intensityDec()

decrement the current intensity and reset counter

_intensityInc()

increment the current intensity and reset counter `_terminate()`

Remove references to ourself in experiments and terminate the loop

`addData(`*result*, *intensity=None*`)`

Deprecated since 1.79.00: This function name was ambiguous. Please use one of these instead:

- .addResponse(result, intensity)
- .addOtherData('dataName', value')

`addOtherData(`*dataName*, *value*`)`

Add additional data to the handler, to be tracked alongside the result data but not affecting the value of the staircase

`addResponse(`*result*, *intensity=None*`)`

Add a 1 or 0 to signify a correct / detected or incorrect / missed trial

Supplying an *intensity* value here indicates that you did not use the recommended intensity in your last trial and the staircase will replace its recorded value with the one you supplied here.

`property beta`

`calculateNextIntensity()`

based on current intensity and counter of correct responses

`confInterval(`*getDifference=False*`)`

Return estimate for the 5%–95% confidence interval (CI).

Parameters

getDifference (bool)
If `True`, return the width of the confidence interval (95% - 5% percentiles). If `False`, return an NumPy array with estimates for the 5% and 95% boundaries.

Returns
scalar or array of length 2.

`property delta`

`property epsilon`

`property gamma`

`getExp()`

Return the ExperimentHandler that this handler is attached to, if any. Returns None if not attached

`getOriginPathAndFile(`*originPath=None*`)`

Attempts to determine the path of the script that created this data file and returns both the path to that script and its contents. Useful to store the entire experiment with the data.

If originPath is provided (e.g. from Builder) then this is used otherwise the calling script is the originPath (fine from a standard python script).

`property grain`

`importData(`*intensities*, *results*`)`

import some data which wasn't previously given to the quest algorithm

`incTrials(`*nNewTrials*`)`

increase maximum number of trials Updates attribute: *nTrials*

property intensity

The intensity (level) of the current staircase

mean()

mean of Quest posterior pdf

mode()

mode of Quest posterior pdf

next()

Advances to next trial and returns it. Updates attributes; *thisTrial*, *thisTrialN*, *thisIndex*, *finished*, *intensities*

If the trials have ended, calling this method will raise a StopIteration error. This can be handled with code such as:

```
staircase = data.QuestHandler(.......)
for eachTrial in staircase:  # automatically stops when done
    # do stuff
```

or:

```
staircase = data.QuestHandler(.......)
while True:  # i.e. forever
    try:
        thisTrial = staircase.next()
    except StopIteration:  # we got a StopIteration error
        break  # break out of the forever loop
    # do stuff here for the trial
```

printAsText(*stimOut=None*, *dataOut=('all_mean', 'all_std', 'all_raw')*, *delim='\t'*, *matrixOnly=False*)

Exactly like saveAsText() except that the output goes to the screen instead of a file

quantile(*p=None*)

quantile of Quest posterior pdf

property range

saveAsExcel(*fileName*, *sheetName='data'*, *matrixOnly=False*, *appendFile=True*, *fileCollisionMethod='rename'*)

Save a summary data file in Excel OpenXML format workbook (*xlsx*) for processing in most spreadsheet packages. This format is compatible with versions of Excel (2007 or greater) and and with OpenOffice (>=3.0).

It has the advantage over the simpler text files (see `TrialHandler.saveAsText()`) that data can be stored in multiple named sheets within the file. So you could have a single file named after your experiment and then have one worksheet for each participant. Or you could have one file for each participant and then multiple sheets for repeated sessions etc.

The file extension *.xlsx* will be added if not given already.

The file will contain a set of values specifying the staircase level ('intensity') at each reversal, a list of reversal indices (trial numbers), the raw staircase / intensity level on *every* trial and the corresponding responses of the participant on every trial.

Parameters

fileName: string
the name of the file to create or append. Can include relative or absolute path.

sheetName: string
the name of the worksheet within the file

matrixOnly: True or False
If set to True then only the data itself will be output (no additional info)

appendFile: True or False
If False any existing file with this name will be overwritten. If True then a new worksheet will be appended. If a worksheet already exists with that name a number will be added to make it unique.

fileCollisionMethod: string
Collision method passed to `handleFileCollision()` This is ignored if `appendFile` is `True`.

`saveAsJson`(*fileName=None*, *encoding='utf-8-sig'*, *fileCollisionMethod='rename'*)

Serialize the object to the JSON format.

Parameters

- `fileName` (*string, or None*) – the name of the file to create or append. Can include a relative or absolute path. If *None*, will not write to a file, but return an in-memory JSON object.
- `encoding` (*string, optional*) – The encoding to use when writing the file.
- `fileCollisionMethod` (*string*) – Collision method passed to `handleFileCollision()`. Can be either of *'rename'*, *'overwrite'*, or *'fail'*.

Notes

Currently, a copy of the object is created, and the copy's .origin attribute is set to an empty string before serializing because loading the created JSON file would sometimes fail otherwise.

`saveAsPickle`(*fileName*, *fileCollisionMethod='rename'*)

Basically just saves a copy of self (with data) to a pickle file.

This can be reloaded if necessary and further analyses carried out.

Parameters
fileCollisionMethod: Collision method passed to `handleFileCollision()`

`saveAsText`(*fileName*, *delim=None*, *matrixOnly=False*, *fileCollisionMethod='rename'*, *encoding='utf-8-sig'*)

Write a text file with the data

Parameters

fileName: a string
The name of the file, including path if needed. The extension *.tsv* will be added if not included.

delim: a string
the delimitter to be used (e.g. ' ' for tab-delimitted, ',' for csv files)

matrixOnly: True/False
If True, prevents the output of the *extraInfo* provided at initialisation.

fileCollisionMethod:
Collision method passed to `handleFileCollision()`

encoding:
The encoding to use when saving a the file. Defaults to *utf-8-sig*.

sd()

standard deviation of Quest posterior pdf

setExp(*exp*)

Sets the ExperimentHandler that this handler is attached to

Do NOT attempt to set the experiment using:

```
trials._exp = myExperiment
```

because it needs to be performed using the *weakref* module.

simulate(*tActual*)

returns a simulated user response to the next intensity level presented by Quest, need to supply the actual threshold level

11.13.8 QuestPlusHandler

class psychopy.data.**QuestPlusHandler**(*nTrials*, *intensityVals*, *thresholdVals*, *slopeVals*, *lowerAsymptoteVals*, *lapseRateVals*, *responseVals=('Yes', 'No')*, *prior=None*, *startIntensity=None*, *psychometricFunc='weibull'*, *stimScale='log10'*, *stimSelectionMethod='minEntropy'*, *stimSelectionOptions=None*, *paramEstimationMethod='mean'*, *extraInfo=None*, *name=''*, *label=''*, ***kwargs*)

QUEST+ implementation. Currently only supports parameter estimation of a Weibull-shaped psychometric function.

The parameter estimates can be retrieved via the *.paramEstimate* attribute, which returns a dictionary whose keys correspond to the names of the estimated parameters (i.e., *QuestPlusHandler.paramEstimate['threshold']* will provide the threshold estimate). Retrieval of the marginal posterior distributions works similarly: they can be accessed via the *.posterior* dictionary.

Parameters

- **nTrials** (*int*) – Number of trials to run.
- **intensityVals** (*collection of floats*) – The complete set of possible stimulus levels. Note that the stimulus levels are not necessarily limited to intensities (as the name of this parameter implies), but they could also be contrasts, durations, weights, etc.
- **thresholdVals** (*float or collection of floats*) – The complete set of possible threshold values.
- **slopeVals** (*float or collection of floats*) – The complete set of possible slope values.
- **lowerAsymptoteVals** (*float or collection of floats*) – The complete set of possible values of the lower asymptote. This corresponds to false-alarm rates in yes-no tasks, and to the guessing rate in n-AFC tasks. Therefore, when performing an n-AFC experiment, the collection should consists of a single value only (e.g., *[0.5]* for 2-AFC, *[0.33]* for 3-AFC, *[0.25]* for 4-AFC, etc.).
- **lapseRateVals** (*float or collection of floats*) – The complete set of possible lapse rate values. The lapse rate defines the upper asymptote of the psychometric function, which will be at *1 - lapse rate*.
- **responseVals** (*collection*) – The complete set of possible response outcomes. Currently, only two outcomes are supported: the first element must correspond to a successful response / stimulus detection, and the second one to an unsuccessful or incorrect response.

For example, in a yes-no task, one would use *['Yes', 'No']*, and in an n-AFC task, *['Correct', 'Incorrect']*; or, alternatively, the less verbose *[1, 0]* in both cases.

- **prior** (*dict of floats*) – The prior probabilities to assign to the parameter values. The dictionary keys correspond to the respective parameters: `threshold`, `slope`, `lowerAsymptote`, `lapseRate`.
- **startIntensity** (*float*) – The very first intensity (or stimulus level) to present.
- **psychometricFunc** (*{'weibull'}*) – The psychometric function to fit. Currently, only the Weibull function is supported.
- **stimScale** (*{'log10', 'dB', 'linear'}*) – The scale on which the stimulus intensities (or stimulus levels) are provided. Currently supported are the decadic logarithm, *log10*; decibels, *dB*; and a linear scale, *linear*.
- **stimSelectionMethod** (*{'minEntropy', 'minNEntropy'}*) – How to select the next stimulus. *minEntropy* will select the stimulus that will minimize the expected entropy. *minNEntropy* will randomly pick pick a stimulus from the set of stimuli that will produce the smallest, 2nd-smallest, . . . , N-smallest entropy. This can be used to ensure some variation in the stimulus selection (and subsequent presentation) procedure. The number *N* will then have to be specified via the *stimSelectionOption* parameter.
- **stimSelectionOptions** (*dict*) – This parameter further controls how to select the next stimulus in case *stimSelectionMethod=minNEntropy*. The dictionary supports two keys: *N* and *maxConsecutiveReps*. *N* defines the number of "best" stimuli (i.e., those which produce the smallest *N* expected entropies) from which to randomly select a stimulus for presentation in the next trial. *maxConsecutiveReps* defines how many times the exact same stimulus can be presented on consecutive trials. For example, to randomly pick a stimulus from those which will produce the 4 smallest expected entropies, and to allow the same stimulus to be presented on two consecutive trials max, use *stimSelectionOptions=dict(N=4, maxConsecutiveReps=2)*. To achieve reproducible results, you may pass a seed to the random number generator via the *randomSeed* key.
- **paramEstimationMethod** (*{'mean', 'mode'}*) – How to calculate the final parameter estimate. *mean* returns the mean of each parameter, weighted by their respective posterior probabilities. *mode* returns the parameters at the peak of the posterior distribution.
- **extraInfo** (*dict*) – Additional information to store along the actual QUEST+ staircase data.
- **name** (*str*) – The name of the QUEST+ staircase object. This will appear in the PsychoPy logs.
- **label** (*str*) – Only used by *MultiStairHandler*, and otherwise ignored.
- **kwargs** (*dict*) – Additional keyword arguments. These might be passed, for example, through a *MultiStairHandler*, and will be ignored. A warning will be emitted whenever additional keyword arguments have been passed.

Warns

RuntimeWarning – If an unknown keyword argument was passed.

Notes

The QUEST+ algorithm was first described by[1].

[1] Andrew B. Watson (2017). QUEST+: A general multidimensional Bayesian adaptive psychometric method. Journal of Vision, 17(3):10. doi: 10.1167/17.3.10.

_intensityDec()

decrement the current intensity and reset counter

_intensityInc()

increment the current intensity and reset counter

_terminate()

Remove references to ourself in experiments and terminate the loop

addData(*result*, *intensity=None*)

Deprecated since 1.79.00: This function name was ambiguous. Please use one of these instead:

- .addResponse(result, intensity)
- .addOtherData('dataName', value')

addOtherData(*dataName*, *value*)

Add additional data to the handler, to be tracked alongside the result data but not affecting the value of the staircase

addResponse(*response*, *intensity=None*)

Add a 1 or 0 to signify a correct / detected or incorrect / missed trial.

This is essential to advance the staircase to a new intensity level!

Supplying an *intensity* value here indicates that you did not use the recommended intensity in your last trial and the staircase will replace its recorded value with the one you supplied here.

calculateNextIntensity()

Based on current intensity, counter of correct responses, and current direction.

getExp()

Return the ExperimentHandler that this handler is attached to, if any. Returns None if not attached

getOriginPathAndFile(*originPath=None*)

Attempts to determine the path of the script that created this data file and returns both the path to that script and its contents. Useful to store the entire experiment with the data.

If originPath is provided (e.g. from Builder) then this is used otherwise the calling script is the originPath (fine from a standard python script).

property intensity

The intensity (level) of the current staircase

next()

Advances to next trial and returns it. Updates attributes; *thisTrial*, *thisTrialN* and *thisIndex*.

If the trials have ended, calling this method will raise a StopIteration error. This can be handled with code such as:

```
staircase = data.StairHandler(.......)
for eachTrial in staircase:  # automatically stops when done
    # do stuff
```

or:

```
staircase = data.StairHandler(.......)
while True:  # ie forever
    try:
```

(continues on next page)

(continued from previous page)

```
        thisTrial = staircase.next()
    except StopIteration:  # we got a StopIteration error
        break  # break out of the forever loop
    # do stuff here for the trial
```

property paramEstimate

The estimated parameters of the psychometric function.

Returns

A dictionary whose keys correspond to the names of the estimated parameters.

Return type

dict of floats

property posterior

The marginal posterior distributions.

Returns

A dictionary whose keys correspond to the names of the estimated parameters.

Return type

dict of np.ndarrays

printAsText(*stimOut=None*, *dataOut=('all_mean', 'all_std', 'all_raw')*, *delim='\t'*, *matrixOnly=False*)

Exactly like saveAsText() except that the output goes to the screen instead of a file

property prior

The marginal prior distributions.

Returns

A dictionary whose keys correspond to the names of the parameters.

Return type

dict of np.ndarrays

saveAsExcel(*fileName*, *sheetName='data'*, *matrixOnly=False*, *appendFile=True*, *fileCollisionMethod='rename'*)

Save a summary data file in Excel OpenXML format workbook (*xlsx*) for processing in most spreadsheet packages. This format is compatible with versions of Excel (2007 or greater) and and with OpenOffice (>=3.0).

It has the advantage over the simpler text files (see `TrialHandler.saveAsText()`) that data can be stored in multiple named sheets within the file. So you could have a single file named after your experiment and then have one worksheet for each participant. Or you could have one file for each participant and then multiple sheets for repeated sessions etc.

The file extension *.xlsx* will be added if not given already.

The file will contain a set of values specifying the staircase level ('intensity') at each reversal, a list of reversal indices (trial numbers), the raw staircase / intensity level on *every* trial and the corresponding responses of the participant on every trial.

Parameters

fileName: string
the name of the file to create or append. Can include relative or absolute path.

sheetName: string
the name of the worksheet within the file

matrixOnly: True or False
If set to True then only the data itself will be output (no additional info)

appendFile: True or False
If False any existing file with this name will be overwritten. If True then a new worksheet will be appended. If a worksheet already exists with that name a number will be added to make it unique.

fileCollisionMethod: string
Collision method passed to `handleFileCollision()` This is ignored if `appendFile` is `True`.

`saveAsJson`(*fileName=None*, *encoding='utf-8-sig'*, *fileCollisionMethod='rename'*)

Serialize the object to the JSON format.

Parameters

- **fileName** (*string, or None*) – the name of the file to create or append. Can include a relative or absolute path. If *None*, will not write to a file, but return an in-memory JSON object.
- **encoding** (*string, optional*) – The encoding to use when writing the file.
- **fileCollisionMethod** (*string*) – Collision method passed to `handleFileCollision()`. Can be either of *'rename'*, *'overwrite'*, or *'fail'*.

Notes

Currently, a copy of the object is created, and the copy's .origin attribute is set to an empty string before serializing because loading the created JSON file would sometimes fail otherwise.

`saveAsPickle`(*fileName*, *fileCollisionMethod='rename'*)

Basically just saves a copy of self (with data) to a pickle file.

This can be reloaded if necessary and further analyses carried out.

Parameters
fileCollisionMethod: Collision method passed to `handleFileCollision()`

`saveAsText`(*fileName*, *delim=None*, *matrixOnly=False*, *fileCollisionMethod='rename'*, *encoding='utf-8-sig'*)

Write a text file with the data

Parameters

fileName: a string
The name of the file, including path if needed. The extension *.tsv* will be added if not included.

delim: a string
the delimitter to be used (e.g. ' ' for tab-delimitted, ',' for csv files)

matrixOnly: True/False
If True, prevents the output of the *extraInfo* provided at initialisation.

fileCollisionMethod:
Collision method passed to `handleFileCollision()`

encoding:
The encoding to use when saving a the file. Defaults to *utf-8-sig*.

setExp(*exp*)

Sets the ExperimentHandler that this handler is attached to

Do NOT attempt to set the experiment using:

```
trials._exp = myExperiment
```

because it needs to be performed using the *weakref* module.

property startIntensity

11.13.9 MultiStairHandler

class psychopy.data.**MultiStairHandler**(*stairType='simple'*, *method='random'*, *conditions=None*, *nTrials=50*, *randomSeed=None*, *originPath=None*, *name=''*, *autoLog=True*)

A Handler to allow easy interleaved staircase procedures (simple or QUEST).

Parameters for the staircases, as used by the relevant `StairHandler` or `QuestHandler` (e.g. the *startVal*, *minVal*, *maxVal*. . .) should be specified in the *conditions* list and may vary between each staircase. In particular, the conditions **must** include a *startVal* (because this is a required argument to the above handlers), a *label* to tag the staircase and a *startValSd* (only for QUEST staircases). Any parameters not specified in the conditions file will revert to the default for that individual handler.

If you need to customize the behaviour further you may want to look at the recipe on *Coder - interleave staircases*.

Params

stairType: 'simple', 'quest', or 'questplus'

Use a `StairHandler`, a `QuestHandler`, or a `QuestPlusHandler`.

method: 'random', 'fullRandom', or 'sequential'
If *random*, stairs are shuffled in each repeat but not randomized more than that (so you can't have 3 repeats of the same staircase in a row unless it's the only one still running). If *fullRandom*, the staircase order is "fully" randomized, meaning that, theoretically, a large number of subsequent trials could invoke the same staircase repeatedly. If *sequential*, don't perform any randomization.

conditions: a list of dictionaries specifying conditions
Can be used to control parameters for the different staircases. Can be imported from an Excel file using *psychopy.data.importConditions* MUST include keys providing, 'startVal', 'label' and 'startValSd' (QUEST only). The 'label' will be used in data file saving so should be unique. See Example Usage below.

nTrials=50
Minimum trials to run (but may take more if the staircase hasn't also met its minimal reversals. See `StairHandler`

randomSeed
[int or None] The seed with which to initialize the random number generator (RNG). If *None* (default), do not initialize the RNG with a specific value.

Example usage:

```
conditions=[
    {'label':'low', 'startVal': 0.1, 'ori':45},
    {'label':'high','startVal': 0.8, 'ori':45},
```

(continues on next page)

(continued from previous page)

```
    {'label':'low', 'startVal': 0.1, 'ori':90},
    {'label':'high','startVal': 0.8, 'ori':90},
    ]
stairs = data.MultiStairHandler(conditions=conditions, nTrials=50)

for thisIntensity, thisCondition in stairs:
    thisOri = thisCondition['ori']

    # do something with thisIntensity and thisOri

    stairs.addResponse(correctIncorrect)  # this is ESSENTIAL

# save data as multiple formats
stairs.saveDataAsExcel(fileName)  # easy to browse
stairs.saveAsPickle(fileName)  # contains more info
```

Raises

`ValueError` – If an unknown randomization option was passed via the *method* keyword argument.

`_startNewPass()`

Create a new iteration of the running staircases for this pass.

This is not normally needed by the user - it gets called at __init__ and every time that next() runs out of trials for this pass.

`_terminate()`

Remove references to ourself in experiments and terminate the loop

`abortCurrentTrial`(*action='random'*)

Abort the current trial (staircase).

Calling this during an experiment abort the current staircase used this trial. The current staircase will be reshuffled into available staircases depending on the *action* parameter.

Parameters

action (*str*) – Action to take with the aborted trial. Can be either of *'random'*, or *'append'*. The default action is *'random'*.

Notes

- When using *action='random'*, the RNG state for the trial handler is not used.

`addData`(*result*, *intensity=None*)

Deprecated 1.79.00: It was ambiguous whether you were adding the response (0 or 1) or some other data concerning the trial so there is now a pair of explicit methods:

- **addResponse(corr,intensity) #some data that alters the next**
  trial value
- **addOtherData('RT', reactionTime) #some other data that won't**
  control staircase

`addOtherData`(*name*, *value*)

Add some data about the current trial that will not be used to control the staircase(s) such as reaction time data

addResponse(*result*, *intensity=None*)

Add a 1 or 0 to signify a correct / detected or incorrect / missed trial

This is essential to advance the staircase to a new intensity level!

getExp()

Return the ExperimentHandler that this handler is attached to, if any. Returns None if not attached

getOriginPathAndFile(*originPath=None*)

Attempts to determine the path of the script that created this data file and returns both the path to that script and its contents. Useful to store the entire experiment with the data.

If originPath is provided (e.g. from Builder) then this is used otherwise the calling script is the originPath (fine from a standard python script).

property intensity

The intensity (level) of the current staircase

next()

Advances to next trial and returns it.

This can be handled with code such as:

```
staircase = data.MultiStairHandler(.......)
for eachTrial in staircase:  # automatically stops when done
    # do stuff here for the trial
```

or:

```
staircase = data.MultiStairHandler(.......)
while True:  # ie forever
    try:
        thisTrial = staircase.next()
    except StopIteration:  # we got a StopIteration error
        break  # break out of the forever loop
    # do stuff here for the trial
```

printAsText(*delim='\t'*, *matrixOnly=False*)

Write the data to the standard output stream

Parameters

delim: a string
the delimitter to be used (e.g. ‘ ‘ for tab-delimitted, ‘,’ for csv files)

matrixOnly: True/False
If True, prevents the output of the *extraInfo* provided at initialisation.

saveAsExcel(*fileName*, *matrixOnly=False*, *appendFile=False*, *fileCollisionMethod='rename'*)

Save a summary data file in Excel OpenXML format workbook (*xlsx*) for processing in most spreadsheet packages. This format is compatible with versions of Excel (2007 or greater) and with OpenOffice (>=3.0).

It has the advantage over the simpler text files (see `TrialHandler.saveAsText()`) that the data from each staircase will be save in the same file, with the sheet name coming from the ‘label’ given in the dictionary of conditions during initialisation of the Handler.

The file extension *.xlsx* will be added if not given already.

The file will contain a set of values specifying the staircase level ('intensity') at each reversal, a list of reversal indices (trial numbers), the raw staircase/intensity level on *every* trial and the corresponding responses of the participant on every trial.

Parameters

fileName: string
the name of the file to create or append. Can include relative or absolute path

matrixOnly: True or False
If set to True then only the data itself will be output (no additional info)

appendFile: True or False
If False any existing file with this name will be overwritten. If True then a new worksheet will be appended. If a worksheet already exists with that name a number will be added to make it unique.

fileCollisionMethod: string
Collision method passed to `handleFileCollision()` This is ignored if `append` is `True`.

`saveAsJson`(*fileName=None*, *encoding='utf-8-sig'*, *fileCollisionMethod='rename'*)

Serialize the object to the JSON format.

Parameters

- **fileName** (*string, or None*) – the name of the file to create or append. Can include a relative or absolute path. If *None*, will not write to a file, but return an in-memory JSON object.
- **encoding** (*string, optional*) – The encoding to use when writing the file.
- **fileCollisionMethod** (*string*) – Collision method passed to `handleFileCollision()`. Can be either of *'rename'*, *'overwrite'*, or *'fail'*.

Notes

Currently, a copy of the object is created, and the copy's .origin attribute is set to an empty string before serializing because loading the created JSON file would sometimes fail otherwise.

`saveAsPickle`(*fileName*, *fileCollisionMethod='rename'*)

Saves a copy of self (with data) to a pickle file.

This can be reloaded later and further analyses carried out.

Parameters
fileCollisionMethod: Collision method passed to `handleFileCollision()`

`saveAsText`(*fileName*, *delim=None*, *matrixOnly=False*, *fileCollisionMethod='rename'*, *encoding='utf-8-sig'*)

Write out text files with the data.

For MultiStairHandler this will output one file for each staircase that was run, with _label added to the fileName that you specify above (label comes from the condition dictionary you specified when you created the Handler).

Parameters

fileName: a string
The name of the file, including path if needed. The extension *.tsv* will be added if not included.

delim: a string
the delimiter to be used (e.g. ' ' for tab-delimited, ',' for csv files)

matrixOnly: True/False
If True, prevents the output of the *extraInfo* provided at initialisation.

fileCollisionMethod:
Collision method passed to `handleFileCollision()`

encoding:
The encoding to use when saving a the file. Defaults to *utf-8-sig*.

setExp(*exp*)

Sets the ExperimentHandler that this handler is attached to

Do NOT attempt to set the experiment using:

```
trials._exp = myExperiment
```

because it needs to be performed using the *weakref* module.

11.13.10 FitWeibull

class psychopy.data.**FitWeibull**(*xx*, *yy*, *sems=1.0*, *guess=None*, *display=1*, *expectedMin=0.5*, *optimize_kws=None*)

Fit a Weibull function (either 2AFC or YN) of the form:

```
y = chance + (1.0-chance)*(1-exp( -(xx/alpha)**(beta) ))
```

and with inverse:

```
x = alpha * (-log((1.0-y)/(1-chance)))**(1.0/beta)
```

After fitting the function you can evaluate an array of x-values with `fit.eval(x)`, retrieve the inverse of the function with `fit.inverse(y)` or retrieve the parameters from `fit.params` (a list with `[alpha, beta]`)

_doFit()

The Fit class that derives this needs to specify its _evalFunction

eval(*xx*, *params=None*)

Evaluate xx for the current parameters of the model, or for arbitrary params if these are given.

inverse(*yy*, *params=None*)

Evaluate yy for the current parameters of the model, or for arbitrary params if these are given.

11.13.11 FitLogistic

class psychopy.data.**FitLogistic**(*xx*, *yy*, *sems=1.0*, *guess=None*, *display=1*, *expectedMin=0.5*, *optimize_kws=None*)

Fit a Logistic function (either 2AFC or YN) of the form:

```
y = chance + (1-chance)/(1+exp((PSE-xx)*JND))
```

and with inverse:

```
x = PSE - log((1-chance)/(yy-chance) - 1)/JND
```

After fitting the function you can evaluate an array of x-values with `fit.eval(x)`, retrieve the inverse of the function with `fit.inverse(y)` or retrieve the parameters from `fit.params` (a list with `[PSE, JND]`)

_doFit()

The Fit class that derives this needs to specify its _evalFunction

eval(*xx*, *params=None*)

Evaluate xx for the current parameters of the model, or for arbitrary params if these are given.

inverse(*yy*, *params=None*)

Evaluate yy for the current parameters of the model, or for arbitrary params if these are given.

11.13.12 FitNakaRushton

class psychopy.data.**FitNakaRushton**(*xx*, *yy*, *sems=1.0*, *guess=None*, *display=1*, *expectedMin=0.5*, *optimize_kws=None*)

Fit a Naka-Rushton function of the form:

```
yy = rMin + (rMax-rMin) * xx**n/(xx**n+c50**n)
```

After fitting the function you can evaluate an array of x-values with `fit.eval(x)`, retrieve the inverse of the function with `fit.inverse(y)` or retrieve the parameters from `fit.params` (a list with `[rMin, rMax, c50, n]`)

Note that this differs from most of the other functions in not using a value for the expected minimum. Rather, it fits this as one of the parameters of the model.

_doFit()

The Fit class that derives this needs to specify its _evalFunction

eval(*xx*, *params=None*)

Evaluate xx for the current parameters of the model, or for arbitrary params if these are given.

inverse(*yy*, *params=None*)

Evaluate yy for the current parameters of the model, or for arbitrary params if these are given.

11.13.13 FitCumNormal

class psychopy.data.**FitCumNormal**(*xx*, *yy*, *sems=1.0*, *guess=None*, *display=1*, *expectedMin=0.5*, *optimize_kws=None*)

Fit a Cumulative Normal function (aka error function or erf) of the form:

```
y = chance + (1-chance)*((special.erf((xx-xShift)/(sqrt(2)*sd))+1)*0.5)
```

and with inverse:

```
x = xShift+sqrt(2)*sd*(erfinv(((yy-chance)/(1-chance)-.5)*2))
```

After fitting the function you can evaluate an array of x-values with fit.eval(x), retrieve the inverse of the function with fit.inverse(y) or retrieve the parameters from fit.params (a list with [centre, sd] for the Gaussian distribution forming the cumulative)

NB: Prior to version 1.74 the parameters had different meaning, relating to xShift and slope of the function (similar to 1/sd). Although that is more in with the parameters for the Weibull fit, for instance, it is less in keeping with standard expectations of normal (Gaussian distributions) so in version 1.74.00 the parameters became the [centre,sd] of the normal distribution.

_doFit()

The Fit class that derives this needs to specify its _evalFunction

eval(*xx*, *params=None*)

Evaluate xx for the current parameters of the model, or for arbitrary params if these are given.

inverse(*yy*, *params=None*)

Evaluate yy for the current parameters of the model, or for arbitrary params if these are given.

11.13.14 importConditions()

psychopy.data.**importConditions**(*fileName*, *returnFieldNames=False*, *selection=''*)

Imports a list of conditions from an .xlsx, .csv, or .pkl file

The output is suitable as an input to `TrialHandler` *trialList* or to `MultiStairHandler` as a *conditions* list.

If *fileName* ends with:

- **.csv: import as a comma-separated-value file**
  (header + row x col)
- **.xlsx: import as Excel 2007 (xlsx) files.**
  No support for older (.xls) is planned.
- **.pkl: import from a pickle file as list of lists**
  (header + row x col)

The file should contain one row per type of trial needed and one column for each parameter that defines the trial type. The first row should give parameter names, which should:

- be unique
- begin with a letter (upper or lower case)
- contain no spaces or other punctuation (underscores are permitted)

selection is used to select a subset of condition indices to be used It can be a list/array of indices, a python *slice* object or a string to be parsed as either option. e.g.:

- “1,2,4” or [1,2,4] or (1,2,4) are the same
- “2:5” # 2, 3, 4 (doesn’t include last whole value)
- “-10:2:” # tenth from last to the last in steps of 2
- slice(-10, 2, None) # the same as above
- random(5) * 8 # five random vals 0-7

11.13.15 functionFromStaircase()

psychopy.data.**functionFromStaircase**(*intensities*, *responses*, *bins=10*)

Create a psychometric function by binning data from a staircase procedure. Although the default is 10 bins Jon now always uses ‘unique’ bins (fewer bins looks pretty but leads to errors in slope estimation)

usage:

```
intensity, meanCorrect, n = functionFromStaircase(intensities,
                                                  responses, bins)
```

where:

intensities
are a list (or array) of intensities to be binned

responses
are a list of 0,1 each corresponding to the equivalent intensity value

bins
can be an integer (giving that number of bins) or 'unique' (each bin is made from aa data for exactly one intensity value)

intensity
a numpy array of intensity values (where each is the center of an intensity bin)

meanCorrect
a numpy array of mean % correct in each bin

n
a numpy array of number of responses contributing to each mean

11.13.16 `bootStraps()`

`psychopy.data.`**`bootStraps`**`(`*dat*, *n=1*`)`

Create a list of n bootstrapped resamples of the data

SLOW IMPLEMENTATION (Python for-loop)

Usage:

```
out = bootStraps(dat, n=1)
```

Where:

dat
an NxM or 1xN array (each row is a different condition, each column is a different trial)

n
number of bootstrapped resamples to create

out

- dim[0]=conditions
- dim[1]=trials
- dim[2]=resamples

11.14 Encryption

Some labs may wish to better protect their data from casual inspection or accidental disclosure. This is possible within using a separate python package, pyFileSec, which grew out of . pyFileSec is distributed with the StandAlone versions of , or can be installed using pip or easy_install via https://pypi.python.org/pypi/PyFileSec/

Some elaboration of pyFileSec usage and security strategy can be found here: https://pythonhosted.org/PyFileSec

Basic usage is illustrated in the Coder demo > misc > encrypt_data.py

11.15 `psychopy.event` - for keypresses and mouse clicks

`class psychopy.event.`**`Mouse`**`(`*visible=None*, *newPos=None*, *win=None*`)`

Easy way to track what your mouse is doing.

It needn't be a class, but since Joystick works better as a class this may as well be one too for consistency

Create your *visual.Window* before creating a Mouse.

Parameters

visible
[bool or None] Show the mouse if True, hide it if False, leave it as is if None (default)

newPos
[**None** or [x,y]] gives the mouse a particular starting position (pygame *Window* only)

win
[**None** or *Window*] the window to which this mouse is attached (the first found if None provided)

`clickReset`(*buttons=(0, 1, 2)*)

Reset a 3-item list of core.Clocks use in timing button clicks.

The pyglet mouse-button-pressed handler uses their clock.getLastResetTime() when a button is pressed so the user can reset them at stimulus onset or offset to measure RT. The default is to reset all, but they can be reset individually as specified in buttons list

`getPos`()

Returns the current position of the mouse, in the same units as the `Window` (0,0) is at centre

`getPressed`(*getTime=False*)

Returns a 3-item list indicating whether or not buttons 0,1,2 are currently pressed.

If *getTime=True* (False by default) then *getPressed* will return all buttons that have been pressed since the last call to *mouse.clickReset* as well as their time stamps:

```
buttons = mouse.getPressed()
buttons, times = mouse.getPressed(getTime=True)
```

Typically you want to call mouse.clickReset() at stimulus onset, then after the button is pressed in reaction to it, the total time elapsed from the last reset to click is in mouseTimes. This is the actual RT, regardless of when the call to *getPressed()* was made.

`getRel`()

Returns the new position of the mouse relative to the last call to getRel or getPos, in the same units as the `Window`.

`getVisible`()

Gets the visibility of the mouse (1 or 0)

`getWheelRel`()

Returns the travel of the mouse scroll wheel since last call. Returns a numpy.array(x,y) but for most wheels y is the only value that will change (except Mac mighty mice?)

`isPressedIn`(*shape*, *buttons=(0, 1, 2)*)

Returns *True* if the mouse is currently inside the shape and one of the mouse buttons is pressed. The default is that any of the 3 buttons can indicate a click; for only a left-click, specify *buttons=[0]*:

```
if mouse.isPressedIn(shape):
if mouse.isPressedIn(shape, buttons=[0]):  # left-clicks only
```

Ideally, *shape* can be anything that has a *.contains()* method, like *ShapeStim* or *Polygon*. Not tested with *ImageStim*.

`mouseMoveTime`()

`mouseMoved`(*distance=None*, *reset=False*)

Determine whether/how far the mouse has moved.

With no args returns true if mouse has moved at all since last getPos() call, or distance (x,y) can be set to pos or neg distances from x and y to see if moved either x or y that far from lastPos, or distance can be an int/float to test if new coordinates are more than that far in a straight line from old coords.

Retrieve time of last movement from self.mouseClock.getTime().

Reset can be to 'here' or to screen coords (x,y) which allows measuring distance from there to mouse when moved. If reset is (x,y) and distance is set, then prevPos is set to (x,y) and distance from (x,y) to here is checked, mouse.lastPos is set as current (x,y) by getPos(), mouse.prevPos holds lastPos from last time mouseMoved was called.

`setExclusive`(*exclusivity*)

Binds the mouse to the experiment window. Only works in Pyglet.

In multi-monitor settings, or with a window that is not fullscreen, the mouse pointer can drift, and thereby PsychoPy might not get the events from that window. setExclusive(True) works with Pyglet to bind the mouse to the experiment window.

Note that binding the mouse pointer to a window will cause the pointer to vanish, and absolute positions will no longer be meaningful getPos() returns [0, 0] in this case.

`setPos`(*newPos=(0, 0)*)

Sets the current position of the mouse, in the same units as the `Window`. (0,0) is the center.

Parameters

newPos
[(x,y) or [x,y]] the new position on the screen

`setVisible`(*visible*)

Sets the visibility of the mouse to 1 or 0

NB when the mouse is not visible its absolute position is held at (0, 0) to prevent it from going off the screen and getting lost! You can still use getRel() in that case.

`property units`

The units for this mouse (will match the current units for the Window it lives in)

`property visible`

Gets the visibility of the mouse (1 or 0)

`psychopy.event.clearEvents`(*eventType=None*)

Clears all events currently in the event buffer.

Optional argument, eventType, specifies only certain types to be cleared.

Parameters

eventType
[**None**, 'mouse', 'joystick', 'keyboard'] If this is not None then only events of the given type are cleared

`psychopy.event.waitKeys`(*maxWait=inf*, *keyList=None*, *modifiers=False*, *timeStamped=False*, *clearEvents=True*)

Same as *~psychopy.event.getKeys*, but halts everything (including drawing) while awaiting input from keyboard.

Parameters

maxWait
[any numeric value.] Maximum number of seconds period and which keys to wait for. Default is float('inf') which simply waits forever.

keyList
[**None** or []] Allows the user to specify a set of keys to check for. Only keypresses from this set of keys will be removed from the keyboard buffer. If the keyList is *None*, all keys will be checked and the key buffer will be cleared completely. NB, pygame doesn't return timestamps (they are always 0)

modifiers
[**False** or True] If True will return a list of tuples instead of a list of keynames. Each tuple has (keyname, modifiers). The modifiers are a dict of keyboard modifier flags keyed by the modifier name (eg. 'shift', 'ctrl').

timeStamped
[**False**, True, or *Clock*] If True will return a list of tuples instead of a list of keynames. Each tuple has (keyname, time). If a *core.Clock* is given then the time will be relative to the *Clock*'s last reset.

clearEvents
[**True** or False] Whether to clear the keyboard event buffer (and discard preceding keypresses) before starting to monitor for new keypresses.

Returns None if times out.

`psychopy.event.`**`getKeys`**(*keyList=None*, *modifiers=False*, *timeStamped=False*)

Returns a list of keys that were pressed.

Parameters

keyList
[**None** or []] Allows the user to specify a set of keys to check for. Only keypresses from this set of keys will be removed from the keyboard buffer. If the keyList is *None*, all keys will be checked and the key buffer will be cleared completely. NB, pygame doesn't return timestamps (they are always 0)

modifiers
[**False** or True] If True will return a list of tuples instead of a list of keynames. Each tuple has (keyname, modifiers). The modifiers are a dict of keyboard modifier flags keyed by the modifier name (eg. 'shift', 'ctrl').

timeStamped
[**False**, True, or *Clock*] If True will return a list of tuples instead of a list of keynames. Each tuple has (keyname, time). If a *core.Clock* is given then the time will be relative to the *Clock*'s last reset.

Author

- 2003 written by Jon Peirce
- 2009 keyList functionality added by Gary Strangman
- 2009 timeStamped code provided by Dave Britton
- 2016 modifiers code provided by 5AM Solutions

`psychopy.event.`**`xydist`**(*p1=(0.0, 0.0)*, *p2=(0.0, 0.0)*)

Helper function returning the cartesian distance between p1 and p2

11.16 `psychopy.filters` - helper functions for creating filters

This module has moved to *psychopy.visual.filters* but you can still (currently) import it as *psychopy.filters* Various useful functions for creating filters and textures (e.g. for PatchStim)

psychopy.visual.filters.**butter2d_bp**(*size*, *cutin*, *cutoff*, *n*)

Bandpass Butterworth filter in two dimensions.

Parameters

size
[tuple] size of the filter

cutin
[float] relative cutin frequency of the filter (0 - 1.0)

cutoff
[float] relative cutoff frequency of the filter (0 - 1.0)

n
[int, optional] order of the filter, the higher n is the sharper the transition is.

Returns

numpy.ndarray
filter kernel in 2D centered

psychopy.visual.filters.**butter2d_hp**(*size*, *cutoff*, *n=3*)

Highpass Butterworth filter in two dimensions.

Parameters

size
[tuple] size of the filter

cutoff
[float] relative cutoff frequency of the filter (0 - 1.0)

n
[int, optional] order of the filter, the higher n is the sharper the transition is.

Returns

numpy.ndarray:
filter kernel in 2D centered

psychopy.visual.filters.**butter2d_lp**(*size*, *cutoff*, *n=3*)

Create lowpass 2D Butterworth filter.

Parameters

size
[tuple] size of the filter

cutoff
[float] relative cutoff frequency of the filter (0 - 1.0)

n
[int, optional] order of the filter, the higher n is the sharper the transition is.

Returns

numpy.ndarray
filter kernel in 2D centered

psychopy.visual.filters.**butter2d_lp_elliptic**(*size*, *cutoff_x*, *cutoff_y*, *n=3*, *alpha=0*, *offset_x=0*, *offset_y=0*)

Butterworth lowpass filter of any elliptical shape.

Parameters

size
[tuple] size of the filter

cutoff_x, cutoff_y
[float, float] relative cutoff frequency of the filter (0 - 1.0) for x and y axes

alpha
[float, optional] rotation angle (in radians)

offset_x, offset_y
[float] offsets for the ellipsoid

n
[int, optional] order of the filter, the higher n is the sharper the transition is.

Returns

numpy.ndarray:
filter kernel in 2D centered

psychopy.visual.filters.**conv2d**(*smaller*, *larger*)

Convolve a pair of 2d numpy matrices.

Uses fourier transform method, so faster if larger matrix has dimensions of size 2**n

Actually right now the matrices must be the same size (will sort out padding issues another day!)

psychopy.visual.filters.**getRMScontrast**(*matrix*)

Returns the RMS contrast (the sample standard deviation) of a array

psychopy.visual.filters.**imfft**(*X*)

Perform 2D FFT on an image and center low frequencies

psychopy.visual.filters.**imifft**(*X*)

Inverse 2D FFT with decentering

psychopy.visual.filters.**make2DGauss**(*x*, *y*, *mean=0.0*, *sd=1.0*, *gain=1.0*, *base=0.0*)

Return the gaussian distribution for a given set of x-vals

Parameters

- **x** – should be x and y indexes as might be created by numpy.mgrid
- **y** – should be x and y indexes as might be created by numpy.mgrid
- **mean** (*float*) – the centre of the distribution - may be a tuple
- **sd** (*float*) – the width of the distribution - may be a tuple
- **gain** (*float*) – the height of the distribution
- **base** (*float*) – an offset added to the result

psychopy.visual.filters.**makeGauss**(*x*, *mean=0.0*, *sd=1.0*, *gain=1.0*, *base=0.0*)

Return the gaussian distribution for a given set of x-vals

Parameters

mean: float
the centre of the distribution

sd: float
the width of the distribution

gain: float
the height of the distribution

base: float
an offset added to the result

psychopy.visual.filters.**makeGrating**(*res*, *ori=0.0*, *cycles=1.0*, *phase=0.0*, *gratType='sin'*, *contr=1.0*)

Make an array containing a luminance grating of the specified params

Parameters

res: integer
the size of the resulting matrix on both dimensions (e.g 256)

ori: float or int (default=0.0)
the orientation of the grating in degrees

cycles:float or int (default=1.0)
the number of grating cycles within the array

phase: float or int (default=0.0)
the phase of the grating in degrees (NB this differs to most PsychoPy phase arguments which use units of fraction of a cycle)

gratType: 'sin', 'sqr', 'ramp' or 'sinXsin' (default="sin")
the type of grating to be 'drawn'

contr: float (default=1.0)
contrast of the grating

Returns
a square numpy array of size resXres

psychopy.visual.filters.**makeMask**(*matrixSize*, *shape='circle'*, *radius=1.0*, *center=(0.0, 0.0)*, *range=(-1, 1)*, *fringeWidth=0.2*)

Returns a matrix to be used as an alpha mask (circle,gauss,ramp).

Parameters

matrixSize: integer
the size of the resulting matrix on both dimensions (e.g 256)

shape: 'circle','gauss','ramp' (linear gradient from center),
'raisedCosine' (the edges are blurred by a raised cosine) shape of the mask

radius: float
scale factor to be applied to the mask (circle with radius of [1,1] will extend just to the edge of the matrix). Radius can asymmetric, e.g. [1.0,2.0] will be wider than it is tall.

center: 2x1 tuple or list (default=[0.0,0.0])
the centre of the mask in the matrix ([1,1] is top-right corner, [-1,-1] is bottom-left)

fringeWidth: float (0-1)
The proportion of the raisedCosine that is being blurred.

range: 2x1 tuple or list (default=[-1,1])
The minimum and maximum value in the mask matrix

`psychopy.visual.filters.`**`makeRadialMatrix`**(*matrixSize*, *center=(0.0, 0.0)*, *radius=1.0*)

Generate a square matrix where each element values is its distance from the centre of the matrix.

Parameters

- **`matrixSize`** (*`int`*) – Matrix size. Corresponds to the number of elements along each dimension. Must be >0.
- **`radius`** (*`float`*) – scale factor to be applied to the mask (circle with radius of [1,1] will extend just to the edge of the matrix). Radius can be asymmetric, e.g. [1.0,2.0] will be wider than it is tall.
- **`center`** (*`2x1 tuple or list (default=[0.0,0.0])`*) – the centre of the mask in the matrix ([1,1] is top-right corner, [-1,-1] is bottom-left)

Returns
Square matrix populated with distance values and *size == (matrixSize, matrixSize)*.

Return type
ndarray

`psychopy.visual.filters.`**`maskMatrix`**(*matrix*, *shape='circle'*, *radius=1.0*, *center=(0.0, 0.0)*)

Make and apply a mask to an input matrix (e.g. a grating)

Parameters

matrix: a square numpy array
array to which the mask should be applied

shape: 'circle','gauss','ramp' (linear gradient from center)
shape of the mask

radius: float
scale factor to be applied to the mask (circle with radius of [1,1] will extend just to the edge of the matrix). Radius can be asymmetric, e.g. [1.0,2.0] will be wider than it is tall.

center: 2x1 tuple or list (default=[0.0,0.0])
the centre of the mask in the matrix ([1,1] is top-right corner, [-1,-1] is bottom-left)

11.17 `psychopy.gui` - create dialogue boxes

11.17.1 `DlgFromDict`

`class psychopy.gui.`**`DlgFromDict`**(*dictionary*, *title=''*, *fixed=None*, *order=None*, *tip=None*, *screen=-1*, *sortKeys=True*, *copyDict=False*, *labels=None*, *show=True*, *alwaysOnTop=False*)

Creates a dialogue box that represents a dictionary of values. Any values changed by the user are change (in-place) by this dialogue box.

Parameters

- **`dictionary`** (*`dict`*) – A dictionary defining the input fields (keys) and pre-filled values (values) for the user dialog
- **`title`** (*`str`*) – The title of the dialog window

- **labels** (*dict*) – A dictionary defining labels (values) to be displayed instead of key strings (keys) defined in *dictionary*. Not all keys in *dictionary* need to be contained in labels.
- **fixed** (*list*) – A list of keys for which the values shall be displayed in non-editable fields
- **order** (*list*) – A list of keys defining the display order of keys in *dictionary*. If not all keys in *dictionary*` are contained in *order*, those will appear in random order after all ordered keys.
- **tip** (*list*) – A dictionary assigning tooltips to the keys
- **screen** (*int*) – Screen number where the Dialog is displayed. If -1, the Dialog will be displayed on the primary screen.
- **sortKeys** (*bool*) – A boolean flag indicating that keys are to be sorted alphabetically.
- **copyDict** (*bool*) – If False, modify *dictionary* in-place. If True, a copy of the dictionary is created, and the altered version (after user interaction) can be retrieved from :attr:~`psychopy.gui.DlgFromDict.dictionary`.
- **labels** – A dictionary defining labels (dict values) to be displayed instead of key strings (dict keys) defined in *dictionary*. Not all keys in `dictionary´ need to be contained in labels.
- **show** (*bool*) – Whether to immediately display the dialog upon instantiation. If False, it can be displayed at a later time by calling its *show()* method.
- **e.g.**
- **::** –

  info = {'Observer':'jwp', 'GratingOri':45,
  'ExpVersion': 1.1, 'Group': ['Test', 'Control']}

  infoDlg = gui.DlgFromDict(dictionary=info,
  title='TestExperiment', fixed=['ExpVersion'])

  if infoDlg.OK:
  print(info)

  else:
  print('User Cancelled')
- **above** (*In the code*)
- **values** (*the contents of info will be updated to the*)
- **box.** (*returned by the dialogue*)
- **OK)** (*If the user cancels (rather than pressing*)

:param : :param then the dictionary remains unchanged. If you want to check whether: :param the user hit OK: :param then check whether DlgFromDict.OK equals: :param True or False: :param See GUI.py for a usage demo: :param including order and tip (tooltip).:

show()

Display the dialog.

11.17.2 Dlg

class psychopy.gui.**Dlg**(*title='PsychoPy Dialog'*, *pos=None*, *size=None*, *style=None*, *labelButtonOK=' OK '*, *labelButtonCancel=' Cancel '*, *screen=-1*, *alwaysOnTop=False*)

A simple dialogue box. You can add text or input boxes (sequentially) and then retrieve the values.

see also the function *dlgFromDict* for an **even simpler** version

Example

```python
from psychopy import gui

myDlg = gui.Dlg(title="JWP's experiment")
myDlg.addText('Subject info')
myDlg.addField('Name:')
myDlg.addField('Age:', 21)
myDlg.addText('Experiment Info')
myDlg.addField('Grating Ori:',45)
myDlg.addField('Group:', choices=["Test", "Control"])
ok_data = myDlg.show()  # show dialog and wait for OK or Cancel
if myDlg.OK:  # or if ok_data is not None
    print(ok_data)
else:
    print('user cancelled')
```

addField(*key*, *initial=''*, *color=''*, *choices=None*, *tip=''*, *required=False*, *enabled=True*, *label=None*)

Adds a (labelled) input field to the dialogue box, optional text color and tooltip.

If 'initial' is a bool, a checkbox will be created. If 'choices' is a list or tuple, a dropdown selector is created. Otherwise, a text line entry box is created.

Returns a handle to the field (but not to the label).

addFixedField(*key*, *label=''*, *initial=''*, *color=''*, *choices=None*, *tip=''*)

Adds a field to the dialog box (like addField) but the field cannot be edited. e.g. Display experiment version.

show()

Presents the dialog and waits for the user to press OK or CANCEL.

If user presses OK button, function returns a list containing the updated values coming from each of the input fields created. Otherwise, None is returned.

Returns

self.data

validate()

Make sure that required fields have a value.

11.17.3 fileOpenDlg()

psychopy.gui.**fileOpenDlg**(*tryFilePath=''*, *tryFileName=''*, *prompt='Select file to open'*, *allowed=None*)

A simple dialogue allowing read access to the file system.

Parameters

tryFilePath: string

default file path on which to open the dialog

tryFileName: string

default file name, as suggested file

prompt: string (default "Select file to open")

can be set to custom prompts

allowed: string (available since v1.62.01)

a string to specify file filters. e.g. "Text files (*.txt) ;; Image files (*.bmp *.gif)" See https://

www.riverbankcomputing.com/static/Docs/PyQt4/qfiledialog.html #getOpenFileNames for further details

If tryFilePath or tryFileName are empty or invalid then current path and empty names are used to start search.

If user cancels, then None is returned.

11.17.4 `fileSaveDlg()`

`psychopy.gui.`**`fileSaveDlg`**`(`*initFilePath=''*, *initFileName=''*, *prompt='Select file to save'*, *allowed=None*`)`

A simple dialogue allowing write access to the file system. (Useful in case you collect an hour of data and then try to save to a non-existent directory!!)

Parameters

initFilePath: string
default file path on which to open the dialog

initFileName: string
default file name, as suggested file

prompt: string (default "Select file to open")
can be set to custom prompts

allowed: string
a string to specify file filters. e.g. "Text files (*.txt) ;; Image files (*.bmp *.gif)" See https://www.riverbankcomputing.com/static/Docs/PyQt4/qfiledialog.html #getSaveFileName for further details

If initFilePath or initFileName are empty or invalid then current path and empty names are used to start search.

If user cancels the None is returned.

11.18 `psychopy.info` - functions for getting information about the system

This module has tools for fetching data about the system or the current Python process. Such info can be useful for understanding the context in which an experiment was run.

`class psychopy.info.`**`RunTimeInfo`**`(`*author=None*, *version=None*, *win=None*, *refreshTest='grating'*, *userProcsDetailed=False*, *verbose=False*`)`

Returns a snapshot of your configuration at run-time, for immediate or archival use.

Returns a dict-like object with info about PsychoPy, your experiment script, the system & OS, your window and monitor settings (if any), python & packages, and openGL.

If you want to skip testing the refresh rate, use 'refreshTest=None'

Example usage: see runtimeInfo.py in coder demos.

Parameters

- **win** (*Window*, False or None) – What window to use for refresh rate testing (if any) and settings. *None* -> temporary window using defaults; *False* -> no window created, used, nor profiled; a *Window()* instance you have already created one.
- **author** (*str or None*) – *None* will try to autodetect first __author__ in sys.argv[0], whereas a *str* being user-supplied author info (of an experiment).
- **version** (*str or None*) – *None* try to autodetect first __version__ in sys.argv[0] or *str* being the user-supplied version info (of an experiment).

- **verbose** (*bool*) – Show additional information. Default is *False*.
- **refreshTest** (*str, bool or None*) – True or 'grating' = assess refresh average, median, and SD of 60 win.flip()s, using visual.getMsPerFrame() 'grating' = show a visual during the assessment; *True* = assess without a visual. Default is *'grating'*.
- **userProcsDetailed** (*bool*) – Get details about concurrent user's processes (command, process-ID). Default is *False*.

Returns

- *A flat dict (but with several groups based on key names)*
- **psychopy** (*version, rush() availability*) – psychopyVersion, psychopyHaveExtRush, git branch and current commit hash if available
- **experiment** (*author, version, directory, name, current time-stamp, SHA1*) – digest, VCS info (if any, svn or hg only), experimentAuthor, experimentVersion, …
- **system** (*hostname, platform, user login, count of users,*) – user process info (count, cmd + pid), flagged processes systemHostname, systemPlatform, …
- **window** (*(see output; many details about the refresh rate, window,*) – and monitor; units are noted) windowWinType, windowWaitBlanking, …windowRefreshTimeSD_ms, … windowMonitor.<details>, …
- **python** (*version of python, versions of key packages*) – (wx, numpy, scipy, matplotlib, pyglet, pygame) pythonVersion, pythonScipyVersion, …
- **openGL** (*version, vendor, rendering engine, plus info on whether*) – several extensions are present openGLVersion, …, openGLextGL_EXT_framebuffer_object, …

_setCurrentProcessInfo(*verbose=False*, *userProcsDetailed=False*)

What other processes are currently active for this user?

_setExperimentInfo(*author*, *version*, *verbose*)

Auto-detect __author__ and __version__ in sys.argv[0] (= the # users's script)

_setPythonInfo()

External python packages, python details

_setSystemInfo()

System info

_setWindowInfo(*win*, *verbose=False*, *refreshTest='grating'*, *usingTempWin=True*)

Find and store info about the window: refresh rate, configuration info.

psychopy.info.**_getSha1hexDigest**(*thing*, *isfile=False*)

Returns base64 / hex encoded sha1 digest of str(thing), or of a file contents. Return None if a file is requested but no such file exists

Author

- 2010 Jeremy Gray; updated 2011 to be more explicit,
- 2012 to remove sha.new()

```
>>> _getSha1hexDigest('1')
'356a192b7913b04c54574d18c28d46e6395428ab'
>>> _getSha1hexDigest(1)
'356a192b7913b04c54574d18c28d46e6395428ab'
```

psychopy.info.**_getUserNameUID**()

Return user name, UID.

UID values can be used to infer admin-level: -1=undefined, 0=full admin/root, >499=assume non-admin/root (>999 on debian-based)

Author

- 2010 written by Jeremy Gray

psychopy.info.**getMemoryUsage**()

Get the memory (RAM) currently used by this Python process, in M.

psychopy.info.**getRAM**()

Return system's physical RAM & available RAM, in M.

11.19 psychopy.layout - For working with vectors and points

Classes and functions for working with coordinates systems.

11.19.1 Overview

Vector(value, units, win)	Class representing a vector.
Position(value, units[, win])	Class representing a position vector.
Size(value, units[, win])	Class representing a size.
Vertices(verts[, obj, size, pos, units, ...])	Class representing an array of vertices.

11.19.2 Details

class psychopy.layout.**Vector**(*value*, *units*, *win*)

Class representing a vector.

A vector is a mathematical construct that specifies a length (or magnitude) and direction within a given coordinate system. This class provides methods to manipulate vectors and convert them between coordinates systems.

This class may be used to assist in positioning stimuli on a screen.

Parameters

- **value** (*ArrayLike*) – Array of vector lengths along each dimension of the space the vector is within. Vectors are specified as either 1xN for single vectors, and Nx2 or Nx3 for multiple vectors.
- **units** (*str or None*) – Units which *value* has been specified in. Applicable values are *'pix'*, *'deg'*, *'degFlat'*, *'degFlatPos'*, *'cm'*, *'pt'*, *'norm'*, *'height'*, or *None*.
- **win** (*~psychopy.visual.Window* or None) – Window associated with this vector. This value must be specified if you wish to map vectors to coordinate systems that require additional information about the monitor the window is being displayed on.

Examples

Create a new vector object using coordinates specified in pixel (*'pix'*) units:

```
my_vector = Vector([256, 256], 'pix')
```

Multiple vectors may be specified by supplying a list of vectors:

```
my_vector = Vector([[256, 256], [640, 480]], 'pix')
```

Operators can be used to compare the magnitudes of vectors:

```
mag_is_same = vec1 == vec2  # same magnitude
mag_is_greater = vec1 > vec2  # one greater than the other
```

1xN vectors return a boolean value while Nx2 or Nx3 arrays return N-length arrays of boolean values.

property cm

Values in units of 'cm' (centimeters).

copy()

Create a copy of this object

property deg

Values in units of 'deg' (degrees of visual angle).

property degFlat

Values in units of 'degFlat' (degrees of visual angle corrected for screen curvature).

When dealing with positions/sizes in isolation; 'deg', 'degFlat' and 'degFlatPos' are synonymous - as the conversion is done at the vertex level.

property degFlatPos

Values in units of 'degFlatPos'.

When dealing with positions/sizes in isolation; 'deg', 'degFlat' and 'degFlatPos' are synonymous - as the conversion is done at the vertex level.

property dimensions

How many dimensions (x, y, z) are specified?

property direction

Direction of vector (i.e. angle between vector and the horizontal plane).

property height

Value in units of 'height' (normalized to the height of the window).

property magnitude

Magnitude of vector (i.e. length of the line from vector to (0, 0) in pixels).

property monitor

The monitor used for calculations within this object (*~psychopy.monitors.Monitor*).

property norm

Value in units of 'norm' (normalized device coordinates).

property pix

Values in units of 'pix' (pixels).

property pt

Vector coordinates in 'pt' (points).

Points are commonly used in print media to define text sizes. One point is equivalent to 1/72 inches, or around 0.35 mm.

set(*value*, *units*, *win=None*)

`validate`(*value*, *units*)

Validate input values.

Ensures the values are in the correct format.

Returns
Parameters *value* and *units*.

Return type
tuple

`class psychopy.layout.Position`(*value*, *units*, *win=None*)

Class representing a position vector.

This class is used to specify the location of a point within some coordinate system (e.g., *(x, y)*).

Parameters

- `value` (*ArrayLike*) – Array of coordinates representing positions within a coordinate system. Positions are specified in a similar manner to *~psychopy.layout.Vector* as either 1xN for single vectors, and Nx2 or Nx3 for multiple positions.
- `units` (*str or None*) – Units which *value* has been specified in. Applicable values are *'pix'*, *'deg'*, *'degFlat'*, *'degFlatPos'*, *'cm'*, *'pt'*, *'norm'*, *'height'*, or *None*.
- `win` (*~psychopy.visual.Window* or None) – Window associated with this position. This value must be specified if you wish to map positions to coordinate systems that require additional information about the monitor the window is being displayed on.

`property cm`

Values in units of 'cm' (centimeters).

`copy()`

Create a copy of this object

`property deg`

Values in units of 'deg' (degrees of visual angle).

`property degFlat`

Values in units of 'degFlat' (degrees of visual angle corrected for screen curvature).

When dealing with positions/sizes in isolation; 'deg', 'degFlat' and 'degFlatPos' are synonymous - as the conversion is done at the vertex level.

`property degFlatPos`

Values in units of 'degFlatPos'.

When dealing with positions/sizes in isolation; 'deg', 'degFlat' and 'degFlatPos' are synonymous - as the conversion is done at the vertex level.

`property dimensions`

How many dimensions (x, y, z) are specified?

`property direction`

Direction of vector (i.e. angle between vector and the horizontal plane).

`property height`

Value in units of 'height' (normalized to the height of the window).

`property magnitude`

Magnitude of vector (i.e. length of the line from vector to (0, 0) in pixels).

property monitor

The monitor used for calculations within this object (*~psychopy.monitors.Monitor*).

property norm

Value in units of 'norm' (normalized device coordinates).

property pix

Values in units of 'pix' (pixels).

property pt

Vector coordinates in 'pt' (points).

Points are commonly used in print media to define text sizes. One point is equivalent to 1/72 inches, or around 0.35 mm.

set(*value*, *units*, *win=None*)

validate(*value*, *units*)

Validate input values.

Ensures the values are in the correct format.

Returns

Parameters *value* and *units*.

Return type

tuple

class psychopy.layout.**Size**(*value*, *units*, *win=None*)

Class representing a size.

Parameters

- **value** (*ArrayLike*) – Array of values representing size axis-aligned bounding box within a coordinate system. Sizes are specified in a similar manner to *~psychopy.layout.Vector* as either 1xN for single vectors, and Nx2 or Nx3 for multiple positions.
- **units** (*str or None*) – Units which *value* has been specified in. Applicable values are *'pix'*, *'deg'*, *'degFlat'*, *'degFlatPos'*, *'cm'*, *'pt'*, *'norm'*, *'height'*, or *None*.
- **win** (*~psychopy.visual.Window* or None) – Window associated with this size object. This value must be specified if you wish to map sizes to coordinate systems that require additional information about the monitor the window is being displayed on.

property cm

Values in units of 'cm' (centimeters).

copy()

Create a copy of this object

property deg

Values in units of 'deg' (degrees of visual angle).

property degFlat

Values in units of 'degFlat' (degrees of visual angle corrected for screen curvature).

When dealing with positions/sizes in isolation; 'deg', 'degFlat' and 'degFlatPos' are synonymous - as the conversion is done at the vertex level.

`property degFlatPos`

Values in units of 'degFlatPos'.

When dealing with positions/sizes in isolation; 'deg', 'degFlat' and 'degFlatPos' are synonymous - as the conversion is done at the vertex level.

`property dimensions`

How many dimensions (x, y, z) are specified?

`property direction`

Direction of vector (i.e. angle between vector and the horizontal plane).

`property height`

Value in units of 'height' (normalized to the height of the window).

`property magnitude`

Magnitude of vector (i.e. length of the line from vector to (0, 0) in pixels).

`property monitor`

The monitor used for calculations within this object (*~psychopy.monitors.Monitor*).

`property norm`

Value in units of 'norm' (normalized device coordinates).

`property pix`

Values in units of 'pix' (pixels).

`property pt`

Vector coordinates in 'pt' (points).

Points are commonly used in print media to define text sizes. One point is equivalent to 1/72 inches, or around 0.35 mm.

`set`(*value*, *units*, *win=None*)

`validate`(*value*, *units*)

Validate input values.

Ensures the values are in the correct format.

Returns
Parameters *value* and *units*.

Return type
tuple

`class psychopy.layout.Vertices`(*verts*, *obj=None*, *size=None*, *pos=None*, *units=None*, *flip=None*, *anchor=None*)

Class representing an array of vertices.

Parameters

- **verts** (*ArrayLike*) – Array of coordinates specifying the locations of vertices.
- **obj** (*object or None*)
- **size** (*ArrayLike or None*) – Scaling factors for vertices along each dimension.
- **pos** (*ArrayLike or None*) – Offset for vertices along each dimension.
- **units** (*str or None*) – Units which *verts* has been specified in. Applicable values are *'pix'*, *'deg'*, *'degFlat'*, *'degFlatPos'*, *'cm'*, *'pt'*, *'norm'*, *'height'*, or *None*.

- **flip** (*ArrayLike or None*) – Array of boolean values specifying which dimensions to flip/mirror. Mirroring is applied prior to any other transformation.
- **anchor** (*str or None*) – Anchor location for vertices, specifies the origin for the vertices.

property anchor

Anchor location (*str*).

Possible values are on of *'top'*, *'bottom'*, *'left'*, *'right'*, *'center'*. Combinations of these values may also be specified (e.g., *'top_center'*, *'center-right'*, *'topleft'*, etc. are all valid).

property anchorAdjust

Map anchor values to numeric vertices adjustments.

property cm

Get absolute positions of vertices in 'cm' units.

property deg

Get absolute positions of vertices in 'deg' units.

property degFlat

Get absolute positions of vertices in 'degFlat' units.

property flip

1x2 array for flipping vertices along each axis; set as *True* to flip or *False* to not flip (*ArrayLike*).

If set as a single value, will duplicate across both axes. Accessing the protected attribute (*._flip*) will give an array of 1s and -1s with which to multiply vertices.

property flipHoriz

Apply horizontal mirroring (*bool*)?

property flipVert

Apply vertical mirroring (*bool*)?

getas(*units*)

property height

Get absolute positions of vertices in 'height' units.

property norm

Get absolute positions of vertices in 'norm' units.

property pix

Get absolute positions of vertices in 'pix' units.

property pos

Positional offset of the vertices (*~psychopy.layout.Vector* or ArrayLike).

setas(*value*, *units*)

property size

Scaling factors for vertices (*~psychopy.layout.Vector* or ArrayLike).

property units

Units which the vertices are specified in (*str*).

11.20 `psychopy.logging` - control what gets logged

Provides functions for logging error and other messages to one or more files and/or the console, using python's own logging module. Some warning messages and error messages are generated by PsychoPy itself. The user can generate more using the functions in this module.

There are various levels for logged messages with the following order of importance: ERROR, WARNING, DATA, EXP, INFO and DEBUG.

When setting the level for a particular log target (e.g. LogFile) the user can set the minimum level that is required for messages to enter the log. For example, setting a level of INFO will result in INFO, EXP, DATA, WARNING and ERROR messages to be recorded but not DEBUG messages.

By default, PsychoPy will record messages of WARNING level and above to the console. The user can silence that by setting it to receive only CRITICAL messages, (which PsychoPy doesn't use) using the commands:

```
from psychopy import logging
logging.console.setLevel(logging.CRITICAL)
```

class psychopy.logging.**LogFile**(*f=None*, *level=30*, *filemode='a'*, *logger=None*, *encoding='utf8'*)

A text stream to receive inputs from the logging system

Create a log file as a target for logged entries of a given level

Parameters

- **f:**
  this could be a string to a path, that will be created if it doesn't exist. Alternatively this could be a file object, sys.stdout or any object that supports .write() and .flush() methods
- **level:**
  The minimum level of importance that a message must have to be logged by this target.
- **filemode: 'a', 'w'**
  Append or overwrite existing log file

setLevel(*level*)

Set a new minimal level for the log file/stream

write(*txt*)

Write directly to the log file (without using logging functions). Useful to send messages that only this file receives

class psychopy.logging.**_Logger**(*format='{t:.4f} \t{levelname} \t{message}'*)

Maintains a set of log targets (text streams such as files of stdout)

self.targets is a list of dicts {'stream':stream, 'level':level}

The string-formatted elements {xxxx} can be used, where each xxxx is an attribute of the LogEntry. e.g. t, t_ms, level, levelname, message

addTarget(*target*)

Add a target, typically a `LogFile` to the logger

flush()

Process all current messages to each target

log(*message*, *level*, *t=None*, *obj=None*, *levelname=None*)

Add the *message* to the log stack at the appropriate *level*

If no relevant targets (files or console) exist then the message is simply discarded.

`removeTarget`(*target*)

Remove a target, typically a `LogFile` from the logger

`psychopy.logging.`**`addLevel`**(*level*, *levelName*)

Associate 'levelName' with 'level'.

This is used when converting levels to text during message formatting.

`psychopy.logging.`**`critical`**(*message*)

Send the message to any receiver of logging info (e.g. a LogFile) of level *log.CRITICAL* or higher

`psychopy.logging.`**`data`**(*msg*, *t=None*, *obj=None*)

Log a message about data collection (e.g. a key press)

usage::
log.data(message)

Sends the message to any receiver of logging info (e.g. a LogFile) of level *log.DATA* or higher

`psychopy.logging.`**`debug`**(*msg*, *t=None*, *obj=None*)

Log a debugging message (not likely to be wanted once experiment is finalised)

usage::
log.debug(message)

Sends the message to any receiver of logging info (e.g. a LogFile) of level *log.DEBUG* or higher

`psychopy.logging.`**`error`**(*message*)

Send the message to any receiver of logging info (e.g. a LogFile) of level *log.ERROR* or higher

`psychopy.logging.`**`exp`**(*msg*, *t=None*, *obj=None*)

Log a message about the experiment (e.g. a new trial, or end of a stimulus)

usage::
log.exp(message)

Sends the message to any receiver of logging info (e.g. a LogFile) of level *log.EXP* or higher

`psychopy.logging.`**`fatal`**(*msg*, *t=None*, *obj=None*)

log.critical(message) Send the message to any receiver of logging info (e.g. a LogFile) of level *log.CRITICAL* or higher

`psychopy.logging.`**`flush`**(*logger=<psychopy.logging._Logger object>*)

Send current messages in the log to all targets

`psychopy.logging.`**`getLevel`**(*level*)

Return the textual representation of logging level 'level'.

If the level is one of the predefined levels (CRITICAL, ERROR, WARNING, INFO, DEBUG) then you get the corresponding string. If you have associated levels with names using addLevelName then the name you have associated with 'level' is returned.

If a numeric value corresponding to one of the defined levels is passed in, the corresponding string representation is returned.

Otherwise, the string "Level %s" % level is returned.

`psychopy.logging.`**`info`**(*msg*, *t=None*, *obj=None*)

Log some information - maybe useful, maybe not

usage::
log.info(message)

Sends the message to any receiver of logging info (e.g. a LogFile) of level *log.INFO* or higher

`psychopy.logging.`**`log`**(*msg*, *level*, *t=None*, *obj=None*)

Log a message

usage::
log(msg, level, t=t, obj=obj)

Log the msg, at a given level on the root logger

`psychopy.logging.`**`setDefaultClock`**(*clock*)

Set the default clock to be used to reference all logging times. Must be a *`psychopy.core.Clock`* object. Beware that if you reset the clock during the experiment then the resets will be reflected here. That might be useful if you want your logs to be reset on each trial, but probably not.

`psychopy.logging.`**`warn`**(*msg*, *t=None*, *obj=None*)

log.warning(message)

Sends the message to any receiver of logging info (e.g. a LogFile) of level *log.WARNING* or higher

`psychopy.logging.`**`warning`**(*message*)

Sends the message to any receiver of logging info (e.g. a LogFile) of level *log.WARNING* or higher

11.20.1 `flush()`

`psychopy.logging.`**`flush`**(*logger=<psychopy.logging._Logger object>*)

Send current messages in the log to all targets

11.20.2 `setDefaultClock()`

`psychopy.logging.`**`setDefaultClock`**(*clock*)

Set the default clock to be used to reference all logging times. Must be a *`psychopy.core.Clock`* object. Beware that if you reset the clock during the experiment then the resets will be reflected here. That might be useful if you want your logs to be reset on each trial, but probably not.

11.21 Deprecated Microphone

Deprecated Use *`Microphone`* for new projects.

11.22 `psychopy.misc` - miscellaneous routines for converting units etc

Wrapper for all miscellaneous functions and classes from psychopy.tools

psychopy.misc has gradually grown very large and the underlying code for its functions are distributed in multiple files. You can still (at least for now) import the functions here using *from psychopy import misc* but you can also import them from the *tools* sub-modules.

11.22.1 From `psychopy.tools.filetools`

`toFile`(filename, data)	Save data (of any sort) as a pickle file.
`fromFile`(filename[, encoding])	Load data from a psydat, pickle or JSON file.
`mergeFolder`(src, dst[, pattern])	Merge a folder into another.

11.22.2 From psychopy.tools.colorspacetools

dkl2rgb(dkl[, conversionMatrix])	Convert from DKL color space (Derrington, Krauskopf & Lennie) to RGB.
dklCart2rgb(LUM, LM, S[, conversionMatrix])	Like dkl2rgb except that it uses cartesian coords (LM,S,LUM) rather than spherical coords for DKL (elev, azim, contr).
rgb2dklCart(picture[, conversionMatrix])	Convert an RGB image into Cartesian DKL space.
hsv2rgb(hsv_Nx3)	Convert from HSV color space to RGB gun values.
lms2rgb(lms_Nx3[, conversionMatrix])	Convert from cone space (Long, Medium, Short) to RGB.
rgb2lms(rgb_Nx3[, conversionMatrix])	Convert from RGB to cone space (LMS).
dkl2rgb(dkl[, conversionMatrix])	Convert from DKL color space (Derrington, Krauskopf & Lennie) to RGB.

11.22.3 From psychopy.tools.coordinatetools

cart2pol(x, y[, units])	Convert from cartesian to polar coordinates.
cart2sph(z, y, x)	Convert from cartesian coordinates (x,y,z) to spherical (elevation, azimuth, radius).
pol2cart(theta, radius[, units])	Convert from polar to cartesian coordinates.
sph2cart(*args)	Convert from spherical coordinates (elevation, azimuth, radius) to cartesian (x,y,z).

11.22.4 From psychopy.tools.monitorunittools

convertToPix(vertices, pos, units, win)	Takes vertices and position, combines and converts to pixels from any unit
cm2pix(cm, monitor)	Convert size in cm to size in pixels for a given Monitor object.
cm2deg(cm, monitor[, correctFlat])	Convert size in cm to size in degrees for a given Monitor object
deg2cm(degrees, monitor[, correctFlat])	Convert size in degrees to size in pixels for a given Monitor object.
deg2pix(degrees, monitor[, correctFlat])	Convert size in degrees to size in pixels for a given Monitor object
pix2cm(pixels, monitor)	Convert size in pixels to size in cm for a given Monitor object
pix2deg(pixels, monitor[, correctFlat])	Convert size in pixels to size in degrees for a given Monitor object

11.22.5 From psychopy.tools.imagetools

array2image(a)	Takes an array and returns an image object (PIL).
image2array(im)	Takes an image object (PIL) and returns a numpy array.
makeImageAuto(inarray)	Combines float_uint8 and image2array operations ie.

11.22.6 From `psychopy.tools.plottools`

plotFrameIntervals(intervals)	Plot a histogram of the frame intervals.

11.22.7 From `psychopy.tools.typetools`

float_uint8(inarray)	Converts arrays, lists, tuples and floats ranging -1:1 into an array of Uint8s ranging 0:255
uint8_float(inarray)	Converts arrays, lists, tuples and UINTs ranging 0:255 into an array of floats ranging -1:1
float_uint16(inarray)	Converts arrays, lists, tuples and floats ranging -1:1 into an array of Uint16s ranging 0:2^16

11.22.8 From `psychopy.tools.unittools`

radians	radians(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])
degrees	degrees(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])

11.23 `psychopy.monitors` - for those that don't like Monitor Center

Most users won't need to use the code here. In general the Monitor Centre interface is sufficient and monitors setup that way can be passed as strings to *Window* s. If there is some aspect of the normal calibration that you wish to override. eg:

```
from psychopy import visual, monitors
mon = monitors.Monitor('SonyG55')#fetch the most recent calib for this monitor
mon.setDistance(114)#further away than normal?
win = visual.Window(size=[1024,768], monitor=mon)
```

You might also want to fetch the `Photometer` class for conducting your own calibrations

11.23.1 `Monitor`

class psychopy.monitors.**Monitor**(*name*, *width=None*, *distance=None*, *gamma=None*, *notes=None*, *useBits=None*, *verbose=True*, *currentCalib=None*, *autoLog=True*)

Creates a monitor object for storing calibration details. This will be loaded automatically from disk if the monitor name is already defined (see methods).

Many settings from the stored monitor can easily be overridden either by adding them as arguments during the initial call.

arguments:

- `width, distance, gamma` are details about the calibration
- `notes` is a text field to store any useful info
- `useBits` True, False, None

- `verbose` True, False, None
- **`currentCalib` is a dictionary object containing various**
  fields for a calibration. Use with caution since the dictionary may not contain all the necessary fields that a monitor object expects to find.

eg:

`myMon = Monitor('sony500', distance=114)` Fetches the info on the sony500 and overrides its usual distance to be 114cm for this experiment.

These can be saved to the monitor file using *`save()`* or not (in which case the changes will be lost)

`_loadAll``()`

Fetches the calibrations for this monitor from disk, storing them as self.calibs

`copyCalib``(`*calibName=None*`)`

Stores the settings for the current calibration settings as new monitor.

`delCalib``(`*calibName*`)`

Remove a specific calibration from the current monitor. Won't be finalised unless monitor is saved

`gammaIsDefault``()`

Determine whether we're using the default gamma values

`getCalibDate``()`

As a python date object (convert to string using calibTools.strFromDate

`getDKL_RGB``(`*RECOMPUTE=False*`)`

Returns the DKL->RGB conversion matrix. If one has been saved this will be returned. Otherwise, if power spectra are available for the monitor a matrix will be calculated.

`getDistance``()`

Returns distance from viewer to the screen in cm, or None if not known

`getGamma``()`

Returns just the gamma value (not the whole grid)

`getGammaGrid``()`

Gets the min,max,gamma values for the each gun

`getLMS_RGB``(`*recompute=False*`)`

Returns the LMS->RGB conversion matrix. If one has been saved this will be returned. Otherwise (if power spectra are available for the monitor) a matrix will be calculated.

`getLevelsPost``()`

Gets the measured luminance values from last calibration TEST

`getLevelsPre``()`

Gets the measured luminance values from last calibration

`getLinearizeMethod``()`

Gets the method that this monitor is using to linearize the guns

`getLumsPost``()`

Gets the measured luminance values from last calibration TEST

`getLumsPre``()`

Gets the measured luminance values from last calibration `getMeanLum()`

Returns the mean luminance of the screen if explicitly stored

`getNotes()`

Notes about the calibration

`getPsychopyVersion()`

Returns the version of PsychoPy that was used to create this calibration

`getSizePix()`

Returns the size of the current calibration in pixels, or None if not defined

`getSpectra()`

Gets the wavelength values from the last spectrometer measurement (if available)

usage:

- nm, power = monitor.getSpectra()

`getUseBits()`

Was this calibration carried out with a bits++ box

`getWidth()`

Of the viewable screen in cm, or None if not known

`lineariseLums`(*desiredLums*, *newInterpolators=False*, *overrideGamma=None*)

Equivalent of `linearizeLums()`.

`linearizeLums`(*desiredLums*, *newInterpolators=False*, *overrideGamma=None*)

lums should be uncalibrated luminance values (e.g. a linear ramp) ranging 0:1

`newCalib`(*calibName=None*, *width=None*, *distance=None*, *gamma=None*, *notes=None*, *useBits=False*, *verbose=True*)

create a new (empty) calibration for this monitor and makes this the current calibration

`save()`

Save the current calibrations to disk.

This will write a *json* file to the *monitors* subfolder of your PsychoPy configuration folder (typically *~/.psychopy3/monitors* on Linux and macOS, and *%APPDATA%psychopy3monitors* on Windows).

`saveMon()`

Equivalent of `save()`.

`setCalibDate`(*date=None*)

Sets the current calibration to have a date/time or to the current date/time if none given. (Also returns the date as set)

`setCurrent`(*calibration=-1*)

Sets the current calibration for this monitor. Note that a single file can hold multiple calibrations each stored under a different key (the date it was taken)

The argument is either a string (naming the calib) or an integer **eg**:

> `myMon.setCurrent('mainCalib')` fetches the calibration named mainCalib. You can name calibrations what you want but PsychoPy will give them names of date/time by default. In Monitor Center you can 'copy…' a calibration and give it a new name to keep a second version.
>
> `calibName = myMon.setCurrent(0)` fetches the first calibration (alphabetically) for this monitor

`calibName = myMon.setCurrent(-1)` fetches the last **alphabetical** calibration for this monitor (this is default). If default names are used for calibrations (ie date/time stamp) then this will import the most recent.

`setDKL_RGB`(*dkl_rgb*)

Sets the DKL->RGB conversion matrix for a chromatically calibrated monitor (matrix is a 3x3 num array).

`setDistance`(*distance*)

To the screen (cm)

`setGamma`(*gamma*)

Sets the gamma value(s) for the monitor. This only uses a single gamma value for the three guns, which is fairly approximate. Better to use setGammaGrid (which uses one gamma value for each gun)

`setGammaGrid`(*gammaGrid*)

Sets the min,max,gamma values for the each gun

`setLMS_RGB`(*lms_rgb*)

Sets the LMS->RGB conversion matrix for a chromatically calibrated monitor (matrix is a 3x3 num array).

`setLevelsPost`(*levels*)

Sets the last set of luminance values measured AFTER calibration

`setLevelsPre`(*levels*)

Sets the last set of luminance values measured during calibration

`setLineariseMethod`(*method*)

Sets the method for linearising 0 uses y=a+(bx)^gamma 1 uses y=(a+bx)^gamma 2 uses linear interpolation over the curve

`setLumsPost`(*lums*)

Sets the last set of luminance values measured AFTER calibration

`setLumsPre`(*lums*)

Sets the last set of luminance values measured during calibration

`setMeanLum`(*meanLum*)

Records the mean luminance (for reference only)

`setNotes`(*notes*)

For you to store notes about the calibration

`setPsychopyVersion`(*version*)

To store the version of PsychoPy that this calibration used

`setSizePix`(*pixels*)

Set the size of the screen in pixels x,y

`setSpectra`(*nm*, *rgb*)

Sets the phosphor spectra measured by the spectrometer

`setUseBits`(*usebits*)

DEPRECATED: Use the new hardware classes to control these devices

`setWidth`(*width*)

Of the viewable screen (cm)

11.23.2 GammaCalculator

class psychopy.monitors.**GammaCalculator**(*inputs=()*, *lums=()*, *gamma=None*, *bitsIN=8*, *bitsOUT=8*, *eq=1*)

Class for managing gamma tables

Parameters:

- **inputs (required)= values at which you measured screen luminance either**
  in range 0.0:1.0, or range 0:255. Should include the min and max of the monitor

Then give EITHER "lums" or "gamma":

- lums = measured luminance at given input levels
- gamma = your own gamma value (single float)
- bitsIN = number of values in your lookup table
- bitsOUT = number of bits in the DACs

myTable.gammaModel myTable.gamma

fitGammaErrFun(*params*, *x*, *y*, *minLum*, *maxLum*)

Provides an error function for fitting gamma function

(used by fitGammaFun)

fitGammaFun(*x*, *y*)

Fits a gamma function to the monitor calibration data.

Parameters:
-xVals are the monitor look-up-table vals, either 0-255 or 0.0-1.0 -yVals are the measured luminances from a photometer/spectrometer

11.23.3 getAllMonitors()

psychopy.monitors.**getAllMonitors**()

Find the names of all monitors for which calibration files exist

11.23.4 getLumSeriesPR650()

psychopy.monitors.**getLumSeriesPR650**(*lumLevels=8*, *winSize=(800, 600)*, *monitor=None*, *gamma=1.0*, *allGuns=True*, *useBits=False*, *autoMode='auto'*, *stimSize=0.3*, *photometer='COM1'*)

DEPRECATED (since v1.60.01): Use `psychopy.monitors.getLumSeries()` instead

11.23.5 getRGBspectra()

psychopy.monitors.**getRGBspectra**(*stimSize=0.3*, *winSize=(800, 600)*, *photometer='COM1'*)

usage:
getRGBspectra(stimSize=0.3, winSize=(800,600), photometer='COM1')

Params

- 'photometer' could be a photometer object or a serial port name on which a photometer might be found (not recommended)

11.23.6 `gammaFun()`

`psychopy.monitors.`**`gammaFun`**`(`*xx*, *minLum*, *maxLum*, *gamma*, *eq=1*, *a=None*, *b=None*, *k=None*`)`

Returns gamma-transformed luminance values. y = gammaFun(x, minLum, maxLum, gamma)

a and b are calculated directly from minLum, maxLum, gamma

Parameters:

- **xx** are the input values (range 0-255 or 0.0-1.0)
- **minLum** = the minimum luminance of your monitor
- **maxLum** = the maximum luminance of your monitor (for this gun)
- **gamma** = the value of gamma (for this gun)

11.23.7 `gammaInvFun()`

`psychopy.monitors.`**`gammaInvFun`**`(`*yy*, *minLum*, *maxLum*, *gamma*, *b=None*, *eq=1*`)`

Returns inverse gamma function for desired luminance values. x = gammaInvFun(y, minLum, maxLum, gamma)

a and b are calculated directly from minLum, maxLum, gamma **Parameters:**

- **xx** are the input values (range 0-255 or 0.0-1.0)
- **minLum** = the minimum luminance of your monitor
- **maxLum** = the maximum luminance of your monitor (for this gun)
- **gamma** = the value of gamma (for this gun)
- **eq determines the gamma equation used;**
  eq==1[default]: yy = a + (b * xx)**gamma eq==2: yy = (a + b*xx)**gamma

11.23.8 `makeDKL2RGB()`

`psychopy.monitors.`**`makeDKL2RGB`**`(`*nm*, *powerRGB*`)`

Creates a 3x3 DKL->RGB conversion matrix from the spectral input powers

11.23.9 `makeLMS2RGB()`

`psychopy.monitors.`**`makeLMS2RGB`**`(`*nm*, *powerRGB*`)`

Creates a 3x3 LMS->RGB conversion matrix from the spectral input powers

11.24 `psychopy.parallel` - functions for interacting with the parallel port

This module provides read / write access to the parallel port for Linux or Windows.

The `Parallel` class described below will attempt to load whichever parallel port driver is first found on your system and should suffice in most instances. If you need to use a specific driver then, instead of using `ParallelPort` shown below you can use one of the following as drop-in replacements, forcing the use of a specific driver:

- *psychopy.parallel.PParallelInpOut*
- *psychopy.parallel.PParallelDLPortIO*
- *psychopy.parallel.PParallelLinux*

Either way, each instance of the class can provide access to a different parallel port.

There is also a legacy API which consists of the routines which are directly in this module. That API assumes you only ever want to use a single parallel port at once.

11.24.1 Legacy functions

We would strongly recommend you use the class above instead: these are provided for backwards compatibility only.

`parallel.setPortAddress()`

Set the memory address or device node for your parallel port of your parallel port, to be used in subsequent commands

Common port addresses:

```
LPT1 = 0x0378 or 0x03BC
LPT2 = 0x0278 or 0x0378
LPT3 = 0x0278
```

or for Linux::

/dev/parport0

This routine will attempt to find a usable driver depending on your platform

`parallel.setData()`

Set the data to be presented on the parallel port (one ubyte). Alternatively you can set the value of each pin (data pins are pins 2-9 inclusive) using *setPin()*

Examples:

```
parallel.setData(0)  # sets all pins low
parallel.setData(255)  # sets all pins high
parallel.setData(2)  # sets just pin 3 high (remember that pin2=bit0)
parallel.setData(3)  # sets just pins 2 and 3 high
```

You can also convert base 2 to int v easily in python:

```
parallel.setData(int("00000011", 2))  # pins 2 and 3 high
parallel.setData(int("00000101", 2))  # pins 2 and 4 high
```

`parallel.setPin(`*state*`)`

Set a desired pin to be high (1) or low (0).

Only pins 2-9 (incl) are normally used for data output:

```
parallel.setPin(3, 1)  # sets pin 3 high
parallel.setPin(3, 0)  # sets pin 3 low
```

`parallel.readPin()`

Determine whether a desired (input) pin is high(1) or low(0).

Pins 2-13 and 15 are currently read here

11.25 psychopy.plugins - utilities for extending with plugins

11.25.1 Overview

scanPlugins()	Scan the system for installed plugins.
listPlugins([which])	Get a list of installed or loaded PsychoPy plugins.
loadPlugin(plugin)	Load a plugin to extend PsychoPy.
requirePlugin(plugin)	Require a plugin to be already loaded.
isPluginLoaded(plugin)	Check if a plugin has been previously loaded successfully by a *loadPlugin()* call.
startUpPlugins(plugins[, add, verify])	Specify which plugins should be loaded automatically when a PsychoPy session starts.
isStartUpPlugin(plugin)	Check if a plugin is registered to be loaded when PsychoPy starts.
pluginMetadata(plugin)	Get metadata from a plugin package.
pluginEntryPoints(plugin[, parse])	Get the entry point mapping for a specified plugin.
computeChecksum(fpath[, method, writeOut])	Compute the checksum hash/key for a given package.

11.25.2 Details

psychopy.plugins.**scanPlugins**()

Scan the system for installed plugins.

This function scans installed packages for the current Python environment and looks for ones that specify PsychoPy entry points in their metadata. Afterwards, you can call *listPlugins()* to list them and *loadPlugin()* to load them into the current session. This function is called automatically when PsychoPy starts, so you do not need to call this unless packages have been added since the session began.

Returns

Number of plugins found during the scan. Calling *listPlugins()* will return the names of the found plugins.

Return type

int

psychopy.plugins.**listPlugins**(*which='all'*)

Get a list of installed or loaded PsychoPy plugins.

This function lists either all potential plugin packages installed on the system, those registered to be loaded automatically when PsychoPy starts, or those that have been previously loaded successfully this session.

Parameters

which (*str*) – Category to list plugins. If 'all', all plugins installed on the system will be listed, whether they have been loaded or not. If 'loaded', only plugins that have been previously loaded successfully this session will be listed. If 'startup', plugins registered to be loaded when a PsychoPy session starts will be listed, whether or not they have been loaded this session. If 'unloaded', plugins that have not been loaded but are installed will be listed. If 'failed', returns a list of plugin names that attempted to load this session but failed for some reason.

Returns

Names of PsychoPy related plugins as strings. You can load all installed plugins by passing list elements to *loadPlugin*.

Return type

list

See also

loadPlugin
Load a plugin into the current session.

Examples

Load all plugins installed on the system into the current session (assumes all plugins don't require any additional arguments passed to them):

```
for plugin in plugins.listPlugins():
    plugins.loadPlugin(plugin)
```

If certain plugins take arguments, you can do this give specific arguments when loading all plugins:

```
pluginArgs = {'some-plugin': (('someArg',), {'setup': True, 'spam': 10})}
for plugin in plugins.listPlugins():
    try:
        args, kwargs = pluginArgs[plugin]
        plugins.loadPlugin(plugin, *args, **kwargs)
    except KeyError:
        plugins.loadPlugin(plugin)
```

Check if a plugin package named *plugin-test* is installed on the system and has entry points into PsychoPy:

```
if 'plugin-test' in plugins.listPlugins():
    print("Plugin installed!")
```

Check if all plugins registered to be loaded on startup are currently active:

```
if not all([p in listPlugins('loaded') for p in listPlugins('startup')]):
    print('Please restart your PsychoPy session for plugins to take effect.')
```

psychopy.plugins.**loadPlugin**(*plugin*)

Load a plugin to extend PsychoPy.

Plugins are packages which extend upon PsychoPy's existing functionality by dynamically importing code at runtime, without modifying the existing installation files. Plugins create or redefine objects in the namespaces of modules (eg. *psychopy.visual*) and unbound classes, allowing them to be used as if they were part of PsychoPy. In some cases, objects exported by plugins will be registered for a particular function if they define entry points into specific modules.

Plugins are simply Python packages,`loadPlugin` will search for them in directories specified in *sys.path*. Only packages which define entry points in their metadata which pertain to PsychoPy can be loaded with this function. This function also permits passing optional arguments to a callable object in the plugin module to run any initialization routines prior to loading entry points.

This function is robust, simply returning *True* or *False* whether a plugin has been fully loaded or not. If a plugin fails to load, the reason for it will be written to the log as a warning or error, and the application will continue running. This may be undesirable in some cases, since features the plugin provides may be needed at some point and would lead to undefined behavior if not present. If you want to halt the application if a plugin fails to load, consider using *requirePlugin()* to assert that a plugin is loaded before continuing.

It is advised that you use this function only when using PsychoPy as a library. If using the Builder or Coder GUI, it is recommended that you use the plugin dialog to enable plugins for PsychoPy sessions spawned by the experiment runner. However, you can still use this function if you want to load additional plugins for a given experiment, having their effects isolated from the main application and other experiments.

Parameters

plugin (*str*) – Name of the plugin package to load. This usually refers to the package or project name.

Returns

True if the plugin has valid entry points and was loaded successfully. Also returns *True* if the plugin was already loaded by a previous *loadPlugin* call this session, this function will have no effect in this case. *False* is returned if the plugin defines no entry points specific to PsychoPy or crashed (an error is logged).

Return type

bool

Warning

Make sure that plugins installed on your system are from reputable sources, as they may contain malware! PsychoPy is not responsible for undefined behaviour or bugs associated with the use of 3rd party plugins.

See also

listPlugins

Search for and list installed or loaded plugins.

requirePlugin

Require a plugin be previously loaded.

Examples

Load a plugin by specifying its package/project name:

```
loadPlugin('psychopy-hardware-box')
```

You can give arguments to this function which are passed on to the plugin:

```
loadPlugin('psychopy-hardware-box', switchOn=True, baudrate=9600)
```

You can use the value returned from *loadPlugin* to determine if the plugin is installed and supported by the platform:

```
hasPlugin = loadPlugin('psychopy-hardware-box')
if hasPlugin:
    # initialize objects which require the plugin here ...
```

psychopy.plugins.**requirePlugin**(*plugin*)

Require a plugin to be already loaded.

This function can be used to ensure if a plugin has already been loaded and is ready for use, raising an exception and ending the session if not.

This function compliments `loadPlugin()`, which does not halt the application if plugin fails to load. This allows PsychoPy to continue working, giving the user a chance to deal with the problem (either by disabling or fixing the plugins). However, *requirePlugin()* can be used to guard against undefined behavior caused by a failed or partially loaded plugin by raising an exception before any code that uses the plugin's features is executed.

Parameters
plugin (*str*) – Name of the plugin package to require. This usually refers to the package or project name.

Raises
RuntimeError – Plugin has not been previously loaded this session.

See also

loadPlugin
Load a plugin into the current session.

Examples

Ensure plugin *psychopy-plugin* is loaded at this point in the session:

```
requirePlugin('psychopy-plugin')  # error if not loaded
```

You can catch the error and try to handle the situation by:

```
try:
    requirePlugin('psychopy-plugin')
except RuntimeError:
    # do something about it ...
```

psychopy.plugins.**isPluginLoaded**(*plugin*)

Check if a plugin has been previously loaded successfully by a *loadPlugin()* call.

Parameters
plugin (*str*) – Name of the plugin package to check if loaded. This usually refers to the package or project name.

Returns
True if a plugin was successfully loaded and active, else *False*.

Return type
bool

See also

loadPlugin
Load a plugin into the current session.

psychopy.plugins.**startUpPlugins**(*plugins*, *add=True*, *verify=True*)

Specify which plugins should be loaded automatically when a PsychoPy session starts.

This function edits `psychopy.preferences.prefs.general['startUpPlugins']` and provides a means to verify if entries are valid. The PsychoPy session must be restarted for the plugins specified to take effect.

If using PsychoPy as a library, this function serves as a convenience to avoid needing to explicitly call *loadPlugin()* every time to use your favorite plugins.

Parameters

- **plugins** (*str*, *list* or *None*) – Name(s) of plugins to have load on startup.
- **add** (*bool*) – If *True* names of plugins will be appended to *startUpPlugins* unless a name is already present. If *False*, *startUpPlugins* will be set to *plugins*, overwriting the previous value. If *add=False* and *plugins=[]* or *plugins=None*, no plugins will be loaded in the next session.
- **verify** (*bool*) – Check if *plugins* are installed and have valid entry points to PsychoPy. Raises an error if any are not. This prevents undefined behavior arsing from invalid plugins being loaded in the next session. If *False*, plugin names will be added regardless if they are installed or not.

Raises

RuntimeError – If *verify=True*, any of *plugins* is not installed or does not have entry points to PsychoPy. This is raised to prevent issues in future sessions where invalid plugins are written to the config file and are automatically loaded.

⚠ **Warning**

Do not use this function within the builder or coder GUI! Use the plugin dialog to specify which plugins to load on startup. Only use this function when using PsychoPy as a library!

Examples

Adding plugins to load on startup:

```
startUpPlugins(['plugin1', 'plugin2'])
```

Clearing the startup plugins list, no plugins will be loaded automatically at the start of the next session:

```
plugins.startUpPlugins([], add=False)
# or ..
plugins.startUpPlugins(None, add=False)
```

If passing *None* or an empty list with *add=True*, the present value of *prefs.general['startUpPlugins']* will remain as-is.

psychopy.plugins.**isStartUpPlugin**(*plugin*)

Check if a plugin is registered to be loaded when PsychoPy starts.

Parameters

plugin (*str*) – Name of the plugin package to check. This usually refers to the package or project name.

Returns

True if a plugin is registered to be loaded when a PsychoPy session starts, else *False*.

Return type

bool

Examples

Check if a plugin was loaded successfully at startup:

```
pluginName = 'psychopy-plugin'
if isStartUpPlugin(pluginName) and isPluginLoaded(pluginName):
    print('Plugin successfully loaded at startup.')
```

psychopy.plugins.**pluginMetadata**(*plugin*)

Get metadata from a plugin package.

Reads the package's PKG_INFO and gets fields as a dictionary. Only packages that have valid entry points to PsychoPy can be queried.

Parameters
plugin (*str*) – Name of the plugin package to retrieve metadata from.

Returns
Metadata fields.

Return type
dict

psychopy.plugins.**pluginEntryPoints**(*plugin*, *parse=False*)

Get the entry point mapping for a specified plugin.

You must call *scanPlugins* before calling this function to get the entry points for a given plugin.

Note this function is intended for internal use by the PsychoPy plugin system only.

Parameters

- **plugin** (*str*) – Name of the plugin package to get advertised entry points.
- **parse** (*bool*) – Parse the entry point specifiers and convert them to fully-qualified names.

Returns
Dictionary of target groups/attributes and entry points objects.

Return type
dict

psychopy.plugins.**computeChecksum**(*fpath*, *method='sha256'*, *writeOut=None*)

Compute the checksum hash/key for a given package.

Authors of PsychoPy plugins can use this function to compute a checksum hash and users can use it to check the integrity of their packages.

Parameters

- **fpath** (*str*) – Path to the plugin package or file.
- **method** (*str*) – Hashing method to use, values are 'md5' or 'sha256'. Default is 'sha256'.
- **writeOut** (*str*) – Path to a text file to write checksum data to. If the file exists, the data will be written as a line at the end of the file.

Returns
Checksum hash digested to hexadecimal format.

Return type
str

Examples

Compute a checksum for a package and write it to a file:

```
with open('checksum.txt', 'w') as f:
    f.write(computeChecksum(
        '/path/to/plugin/psychopy_plugin-1.0-py3.6.egg'))
```

11.26 psychopy.preferences - getting and setting preferences

You can set preferences on a per-experiment basis. For example, if you would like to use a specific audio library, but don't want to touch your user settings in general, you can import preferences and set the option *audioLib* accordingly:

```
from psychopy import prefs
prefs.hardware['audioLib'] = ['pyo']
from psychopy import sound
```

!!IMPORTANT!! You must import the sound module **AFTER** setting the preferences. To check that you are getting what you want (don't do this in your actual experiment):

```
print sound.Sound
```

The output should be `<class 'psychopy.sound.SoundPyo'>` for pyo, or `<class 'psychopy.sound.SoundPygame'>` for pygame.

You can find the names of the *preferences sections and their different options here*.

11.26.1 Preferences

Class for loading / saving prefs

class psychopy.preferences.**Preferences**

Users can alter preferences from the dialog box in the application, by editing their user preferences file (which is what the dialog box does) or, within a script, preferences can be controlled like this:

```
import psychopy
psychopy.prefs.hardware['audioLib'] = ['ptb', 'pyo','pygame']
print(psychopy.prefs)
# prints the location of the user prefs file and all the current vals
```

Use the instance of *prefs*, as above, rather than the *Preferences* class directly if you want to affect the script that's running.

getPaths()

Get the paths to various directories and files used by PsychoPy.

If the paths are not found, they are created. Usually, this is only necessary on the first run of PsychoPy. However, if the user has deleted or moved the preferences directory, this method will recreate those directories.

loadAll()

Load the user prefs and the application data

loadAppData()

Fetch app data config (unless this is a lib-only installation)

loadUserPrefs()

load user prefs, if any; don't save to a file because doing so will break easy_install. Saving to files within the psychopy/ is fine, eg for key-bindings, but outside it (where user prefs will live) is not allowed by easy_install (security risk)

resetPrefs()

removes userPrefs.cfg, does not touch appData.cfg

restoreBadPrefs(*cfg*, *result*)

result = result of validate

saveAppData()

Save the various setting to the appropriate files (or discard, in some cases)

saveUserPrefs()

Validate and save the various setting to the appropriate files (or discard, in some cases)

validate()

Validate (user) preferences and reset invalid settings to defaults

11.27 psychopy.serial - functions for interacting with the serial port

is compatible with Chris Liechti's pyserial package. You can use it like this:

```
import serial
ser = serial.Serial(0, 19200, timeout=1)  # open first serial port
#ser = serial.Serial('/dev/ttyS1', 19200, timeout=1)#or something like this for Mac/Linux␣
↪machines
ser.write('someCommand')
line = ser.readline()   # read a '\n' terminated line
ser.close()
```

Ports are fully configurable with all the options you would expect of RS232 communications. See https://pyserial.readthedocs.io for further details and documentation.

pyserial is packaged in the Standalone (Windows and Mac distributions), for manual installations you should install this yourself.

11.28 psychopy.web - Web methods

11.28.1 Test for access

psychopy.web.**haveInternetAccess**(*forceCheck=False*)

Detect active internet connection or fail quickly.

If forceCheck is False, will rely on a cached value if possible.

psychopy.web.**requireInternetAccess**(*forceCheck=False*)

Checks for access to the internet, raise error if no access.

11.28.2 Proxy set-up and testing

psychopy.web.**setupProxy**(*log=True*)

Set up the urllib proxy if possible.

The function will use the following methods in order to try and determine proxies:

1. standard urllib.request.urlopen (which will use any statically-defined http-proxy settings)
2. previous stored proxy address (in prefs)
3. proxy.pac files if these have been added to system settings
4. auto-detect proxy settings (WPAD technology)

Return type

True (success) or False (failure)

PsychoPy now supports Plugins , making the application even more flexible and extendable! Check out the range of additional features supported via our plugins pages .

Further information:

CHAPTER

TWELVE

TIMING INFORMATION FOR PSYCHOPY

There is documentation about how to optimize timing in at *Timing Issues and synchronisation*

We recently ran a study testing the timing on a wide range of software packages, online and offline. The data for that study are available below:

12.1 Mega-timing study data

Here are the data summaries for our paper, The timing mega-study: comparing a range of experiment generators, both lab-based and online

You can read the full preprint of the paper at https://peerj.com/articles/9414/

12.1.1 Table2: lab-based timing data

This is a sortable version of Table2 from The timing mega-study: comparing a range of experiment generators, both lab-based and online

Timing summaries of lab-based software by package and platform. The Var(iability) measures are the inter-trial standard deviations of the various latencies for that configuration. The table is sorted by the mean of those variabilities (Mean Var). The Lag/Bias measures are the mean latency values, for that configuration. In the case of audiovisual sync, a negative bias indicates the audio lead the visual stimulus, a positive bias means the visual lead the audio. Each of the values with a hyperlink will lead to a plot of the distribution of values leading to that summary value.

12.1.2 Table3: online timing data

This is a sortable version of Table3 from The timing mega-study: comparing a range of experiment generators, both lab-based and online

Timing summaries of web-based software by package, platform, and browser. The Var(iability) measures are the inter-trial standard deviations of the various latencies for that configuration. The table is sorted by the mean of those variabilities (Mean Var). The Lag/Bias measures are the mean latency values, for that configuration. In the case of audiovisual sync, a negative bias indicates the audio lead the visual stimulus, a positive bias means the visual lead the audio. Each of the values with a hyperlink will lead to a plot of the distribution of values leading to that summary value.

CHAPTER

THIRTEEN

TROUBLESHOOTING

Regrettably, occasionally has bugs. Running on all possible hardware and all platforms is a big ask. That said, a huge number of bugs have been resolved by the fact that there are literally 1000s of people using the software that have *contributed either bug reports and/or fixes*.

Below are some of the more common problems and their workarounds, as well as advice on how to get further help.

13.1 The application doesn't start

You may find that you try to launch the application, the splash screen appears and then goes away and nothing more happens. What this means is that an error has occurred during startup itself.

Commonly, the problem is that a preferences file is somehow corrupt. To fix that see *Cleaning preferences and app data*, below.

If resetting the preferences files doesn't help then we need to get to an error message in order to work out why the application isn't starting. The way to get that message depends on the platform (see below).

Windows users (starting from the Command Prompt):

1. Did you get an error message that "This application failed to start because the application configuration is incorrect. Reinstalling the application may fix the problem"? If so that indicates you need to update your .NET installation to SP1 .
2. **open a Command Prompt (terminal):**
    1. go to the Windows Start menu
    2. select Run… and type in cmd <Return>
3. paste the following into that window (Ctrl-V doesn't work in Cmd.exe but you can right-click and select Paste):

    ```
    "C:\Program Files\PsychoPy2\python.exe" -m psychopy.app.psychopyApp
    ```

4. when you hit <return> you will hopefully get a moderately useful error message that you can *Contribute to the Forum (mailing list)*

***Mac users*:**

1. open the Console app (open spotlight and type console)
2. if there are a huge number of messages there you might find it easiest to clear them (the brush icon) and then start again to generate a new set of messages

13.2 I run a Builder experiment and nothing happens

An error message may have appeared in a dialog box that is hidden (look to see if you have other open windows somewhere).

An error message may have been generated that was sent to output of the Coder view:

1. go to the Coder view (from the Builder>View menu if not visible)
2. if there is no Output panel at the bottom of the window, go to the View menu and select Output
3. try running your experiment again and see if an error message appears in this Output view

If you still don't get an error message but the application still doesn't start then manually turn off the viewing of the Output (as below) and try the above again.

13.3 Manually turn off the viewing of output

Very occasionally an error will occur that crashes the application *after* the application has opened the Coder Output window. In this case the error message is still not sent to the console or command prompt.

To turn off the Output view so that error messages are sent to the command prompt/terminal on startup, open your appData.cfg file (see *Cleaning preferences and app data*), find the entry:

```
[coder]
showOutput = True
```

and set it to *showOutput* = *False* (note the capital 'F').

13.4 Use the source (Luke?)

comes with all the source code included. You may not think you're much of a programmer, but have a go at reading the code. You might find you understand more of it than you think!

To have a look at the source code do one of the following:

- when you get an error message in the *Coder* click on the hyperlinked error lines to see the relevant code
- **on Windows**
  - go to *<location of PsychoPy app>\Lib\site-packages\psychopy*
  - have a look at some of the files there
- **on Mac**
  - right click the app and select *Show Package Contents*
  - navigate to *Contents/Resources/lib/pythonX.X/psychopy*

13.5 Cleaning preferences and app data

Every time you shut down (by normal means) your current preferences and the state of the application (the location and state of the windows) are saved to disk. If is crashing during startup you may need to edit those files or delete them completely.

The exact location of those files varies by machine but on windows it will be something like *%APPDATA%psychopy3* and on Linux/MacOS it will be something like *~/.psychopy3*. You can find it running this in the commandline (if you have multiple Python installations then make sure you change *python* to the appropriate one for :

```
python -c "from psychopy import prefs; print(prefs.paths['userPrefsDir'])"
```

Within that folder you will find *userPrefs.cfg* and *appData.cfg*. The files are simple text, which you should be able to edit in any text editor.

If the problem is that you have a corrupt experiment file or script that is trying and failing to load on startup, you could simply delete the *appData.cfg* file. Please *also Contribute to the Forum (mailing list)* a copy of the file that isn't working so that the underlying cause of the problem can be investigated (google first to see if it's a known issue).

13.6 Errors with getting/setting the Gamma ramp

There are two common causes for errors getting/setting gamma ramps depending on whether you're running Windows or Linux (we haven't seen these problems on Mac).

13.6.1 MS Windows bug in release 1903

In Windows release 1903 Microsoft added a bug that prevents getting/setting the gamma ramp. This only occurs in certain scenarios, like when the screen orientation is in portrait, or when it is extended onto a second monitor, but it does affect **all versions of PsychoPy**.

For the Windows bug the workarounds are as follows:

If you don't need gamma correction then, as of 3.2.4, you can go to the preferences and set the *defaultGammaFailPolicy* to be be 'warn' (rather than 'abort') and then your experiment will still at least run, just without gamma correction.

If you do need gamma correction then there isn't much that the team can do until Microsoft fixes the underlying bug. You'll need to do one of:

- Not using Window 1903 (e.g. revert the update) until a fix is listed on the status of the gamma bug
- Altering your monitor settings in Windows (e.g. turning off extended desktop) until it works . Unfortunately that might mean you can't use dual independent displays for vision science studies until Microsoft fix it.

13.6.2 Linux missing xorg.conf

On Linux some systems appear to be missing a configuration file and adding this back in and restarting should fix things.

Create the following file (including the folders as needed):

/etc/X11/xorg.conf.d/20-intel.conf

and put the following text inside (assuming you have an intel card, which is where we've typically seen the issue crop up):

```
Section "Device"
    Identifier "Intel Graphics"
    Driver "intel"
EndSection
```

For further information on the discussion of this (Linux) issue see
https://github.com/psychopy/psychopy/issues/2061

CHAPTER

FOURTEEN

ALERTS

The Alerts system is designed to provide you information of varying levels about things that may be a concern in your study (but equally may not be - it depends on your study!). Alerts show up within with a code and that code should take you here for detailed information about the root cause of the problem, the workarounds, and the type of study that might be badly affected by this issue.

14.1 2xxx: Issues with units

These issue usually result in problems with stimuli not appearing (too fast, too small, off-screen) or being an unexpected size or color.

14.1.1 2115: Stimulus size is bigger than the window dimensions

Synopsis

Your stimulus size exceeds the X or Y window dimensions. Stimuli sized greater than the window size will not be completely visible.

Details

This issue is often caused by an inconsistency between the units of your stimulus and the values being requested. For instance a size of 3, when the units are *deg* is sensible (3 degrees would be within the screen dimensions) but a size of 3 when the units are *height* would not be sensible (3 times bigger than the height of the screen).

PsychoPy versions affected

All versions.

Solutions

Check the size and the *units of the stimulus* carefully. You may also need to check the monitor calibration if you're using units that depend on the monitor size and resolution (like *cm* and *deg*).

14.1.2 2120: Stimulus size is smaller than 1 pixel

Synopsis

Stimuli size requested is smaller than 1 pixel on X and/or Y dimensions. Stimuli sized smaller than 1 pixel will not be visible.

Details

This issue is often caused by an inconsistency between the units of your stimulus and the values being requested. For instance a size of 0.1, when the units are *height* is sensible (1/10th the height of the screen) but a size of 0.1 when the units are *pix* would not be sensible (1/10th of a pixel).

PsychoPy versions affected

All versions

Solutions

Check the size and the *units of the stimulus* carefully. You may also need to check the monitor calibration if you're using units that depend on the monitor size and resolution (like *cm* and *deg*).

14.1.3 2155: Stimulus position is beyond the bounds of the window

Synopsis

Your stimulus position exceeds the X or Y window dimensions. Stimuli centered beyond the window will not be completely visible.

Details

This issue is often caused by an inconsistency between the units of your stimulus and the values being requested. For instance a position of (3, 0), when the units are *deg* is sensible (3 degrees to the right of the screen center) but a position of (3, 0) when the units are *height* would not be sensible (3 times the height of the screen to the right of the center).

PsychoPy versions affected

All versions

Solutions

Check the position and the *units of the stimulus* carefully. You may also need to check the monitor calibration if you're using units that depend on the monitor size and resolution (like *cm* and *deg*).

14.2 3xxx: Issues with timing

There are many ways for timing to go wrong, and these alerts will warn you about some of them (but we do always recommend that you test your timing carefully with hardware devices if precise timing is important to your study).

14.2.1 3110: Stimulus duration is less than one screen refresh

Synopsis

Your stimulus is scheduled to last for a duration that can't be achieved with a normal 60 Hz monitor. Duration will implicitly be round up (probably) to the next frame duration.

Requested start or stop times of visual components cannot be presented for times requested. Accurate presentation times must be in increments of your screen refresh rate.

Details

Stimuli can only be presented for a fixed number of screen refreshes. If you screen has a refresh rate of 60 Hz (common for standard monitors) then each screen refresh period lasts 1/60 s (roughly 16.6667 ms). That means you can present your stimulus for 16.7 ms but not for, say, 5 ms because that would require the stimulus to be presented for half of one screen refresh period.

PsychoPy versions affected

All versions.

Solutions

We recommend for brief stimuli that you simply specify your stimulus duration in terms of the number of frames it should be presented (e.g. 1, 2, 3, for 16.7, 33.3 and 50 ms respectively). That reminds you of what is possible and means that PsychoPy won't have to guess about what to do when the desired duration isn't achievable.

If you need stimuli to be presented for briefer durations than 16.7 ms then you should look into high-frame-rate displays (100, 120 and 144 Hz displays are all available). There are also now vraiable-frame-rate monitors that can vary the duration of each frame within limits. If you are *already* using a high- or variable-rate monitor then this alert may not be relevant to you.

14.2.2 3115: Stimulus duration is not possible on a standard monitor refresh

Synopsis

If using a 60Hz or 100Hz monitor, then for accurate presentation of visual stimuli, components must be presented in valid multiples of screen refresh for 60 Hz or 100 Hz.

Details

When presenting stimuli at, say, 60 Hz you stimulus can be presented for 1 frame (1/60 s = 16.667 ms), 2 frames (2/60 s = 33.333 ms) but not for intervening periods (20 ms is not possible because the stimulus would have to be presented for a little more than 1 frame, which isn't physically possible on standard fixed-framerate monitors.

PsychoPy versions affected

All versions.

Solutions

We recommend for brief stimuli that you simply specify your stimulus duration in terms of the number of frames it should be presented (e.g. 1, 2, 3, for 16.7, 33.3 and 50 ms respectively). That reminds you of what is possible and means that PsychoPy won't have to guess about what to do when the desired duration isn't achievable.

If you need stimuli to be presented for briefer durations than 16.7 ms then you should look into high-frame-rate displays (100, 120 and 144 Hz displays are all available). There are also now vraiable-frame-rate monitors that can vary the duration of each frame within limits. If you are *already* using a high- or variable-rate monitor then this alert may not be relevant to you.

14.2.3 3210: Exclusive latency mode

Synopsis

Speaker is set to "Exclusive low latency" mode. As resampling is enabled, this mode has little benefit over "Shared low latency" mode, with some drawbacks.

Details

“Exclusive low latency” mode differs from “Shared low latency” mode in two ways: - PsychoPy will take exclusive control of the speaker, meaning that other apps won’t be able to access it while your experiment is running, and may crash. - Because of this exclusive control, we have more flexibility in the range of sample rates which we can play on the speaker.

However, if “resampling” is ticked, then audio is automatically resampled to match the speaker’s sample rate anyway, so “Exclusive low latency” offers little benefit over “Shared low latency”.

PsychoPy versions affected

>2025.1.0

Solutions

If you actively want to prevent the speaker from being used by other applications (for example, you want to make sure the participant isn’t playing music in the background), then no action is required and you can ignore this alert.

If you don’t want this, then you most likely want “Shared low latency” mode instead.

14.2.4 3610: Multiple Components in same Routine validated by same validator

Synopsis

A validator can only validate one stimulus at a time, but it looks like you have multiple stimuli in the same Routine which are validated by the same validator. This may be fine, but you will need to be sure that they don’t overlap in their timing.

Details

A validator can only validate one stimulus at a time, as they only respond True/False for whether there’s light/sound, not distinguishing between different sources. If you have two stimuli at the same time, then they will obscure the start/stop times of one another, meaning your validator will give the wrong results.

PsychoPy versions affected

>2025.1.0

Solutions

This may be fine! You just need to make sure that the multiple Components in the same Routine which are validated by the same validator aren’t overlapping in their timing. If they’re not, you can ignore this!

14.3 4xxx: Issues with Builder experiments

Issues specifically around Builder experiments doing unexpected things.

14.3.1 4051: Experiment from future version

Synopsis

It looks like you’re trying to open an experiment built in a newer version of PsychoPy than you currently have installed. This can cause problems, your experiment may run differently to how you expect or may not run at all.

Details

Between different version of PsychoPy, we make a number of changes to improve usability and performance, but which mean that newer experiments may contain code which older versions do not know how to handle. We always try to maintain "backwards compatibility" - so that experiments built in older versions run the same in newer versions. However, we cannot predict what changes we will make years down the line, so cannot guarantee "forwards compatibility" in the same way. This means that experiments built on newer versions of PsychoPy may run differently on older versions, or may not run at all.

PsychoPy versions affected

Any

Solutions

To fix this, we recommend updating to the newest version of PsychoPy. Or, if you need an older version for other reasons, you can set this specific experiment to run in a different version by changing the "Use Version" parameter in Experiment Settings to the version it was created in (or a newer version).

14.3.2 4052: Experiment fixed to past version

Synopsis

It looks like your experiment is set to run in a version before 2021.1.0, the version of PsychoPy you have currently installed is newer than this, so saving the experiment in your current version may add new types of parameters which the version it is set to cannot interpret.

Details

Between different version of PsychoPy, we make a number of changes to improve usability and performance, but which mean that newer experiments may contain code which older versions do not know how to handle. We always try to maintain "backwards compatibility" - so that experiments built in older versions run the same in newer versions. However, we cannot predict what changes we will make years down the line, so cannot guarantee "forwards compatibility" in the same way. This means that experiments built in the current version of PsychoPy but set to run in an older version may not run as expected, or at all. This is a particular issue between 2020.2.10 and 2021.1.0 as between the two we added several new types of parameter, meaning if you run in 2020.2 PsychoPy will not recognise them.

PsychoPy versions affected

>2021.1.0

Solutions

To fix this, we recommend setting the experiment version in the Experiment Settings menu to be 2021.1.0 or newer, as these will have the ability to recognise the new parameter types.

14.3.3 4105: Component start time exceeds its stop time

Synopsis

A component start time/frame exceeds the stop time/frame. This means that the component starts *after* it is due to finish and the stimulus will most likely not be shown at all.

PsychoPy versions affected

All versions.

Solutions

Check your start and stop time carefully, including the units being used. For example your stimulus might be set to start at a certain *time* (say 36 seconds) rather than frame number 36.

14.3.4 4115: Component start/stop in units of frames must be whole numbers

Synopsis

Component start and stop times/durations in frames must be given as whole numbers.

Details

Since it isn't possible to start or stop a stimulus part-way through a screen refresh it would be unwise to request that PsychoPy attempts that.

PsychoPy versions affected

All versions.

Solutions

Check your stimulus start/stop time and set to an integer value instead of a decimal value.

14.3.5 4120: Component stop duration with no start time

Synopsis

In order for stop time to be set as a duration, PsychoPy needs to know the time at which the component started. When there's no start time, stop time will be calculated from the duration as if start time were 0s.

Details

PsychoPy versions affected

> 2022.1.0

Solutions

You can either add a start time to this component, or change the stop time to be set in a different way (e.g. *time (s)*)

14.3.6 4125: Microphone component given blank stop time

Synopsis

Mircophone components can't record forever - there is a "buffer" with a finite size which will eventually fill up. As you requested for a Microphone component to continue recording forever (by leaving the Stop field blank), PsychoPy has substituted in the maximum time which you Microphone can record for, given the size of its buffer and the sample rate it records at.

Details

PsychoPy versions affected

>2023.1.0

Solutions

No action is needed! Your Mircophone component will record for as long as it is able. To silence this alert, simply give your Microphone component a duration.

14.3.7 4130: Static Component given infinite stop time

Synopsis

Static Component pauses the frame loop while it's active, meaning that unlike other Components, it needs to have a fixed stop time to know when to finish. Otherwise, the experiment will simply stall forever, even if there's code in the Each Frame tab of a Code Component to end the experiment / stop the Static Component (as the frame loop is paused, so Each Frame code isn't happening). You won't even be able to end the experiment with the ESCAPE key as this is checked each frame!

Details

The only instance where you might want an infinite Static Component is if you're using threading, as threaded operations don't rely on a frame loop to execute their code. In this case, you can ignore this message. However, this is a rare and fairly advanced use case, so if the previous sentences sounded like jibberish, you almost certainly don't want an infinite Static Component!

PsychoPy versions affected

All

Solutions

Give your Static Component a fixed duration.

14.3.8 4205: Probable syntax error detected in your Python code

Synopsis

Python syntax error found in your code component.

Details

This *may* be spurious in that the code check may have failed to understand something that is a valid syntax (syntax in Python can change according to version) but this will become clear if you run your experiment and it fails to run.

PsychoPy versions affected

All versions.

Solutions

Check the code carefully at the indicated line and on the few lines above. Check especially for things like un-matched parentheses or quote symbols.

14.3.9 4210: Probable syntax error detected in your JavaScript code

Synopsis

JavaScript syntax error found in your code component.

Details

This alert *may* be spurious in that the code check may have failed to understand something that is a valid syntax (syntax in JavaScript can change according to version) but this will become clear if you run your experiment and it fails to run.

PsychoPy versions affected

All versions.

Solutions

Check the code carefully at the indicated line and on the few lines above. Check especially for things like un-matched parentheses or quote symbols.

14.3.10 4305: Component is currently disabled in your experiment

Synopsis

You have a disabled Component in your experiment. This alert is created to inform users that disabled components will not be written to your experiment script. This may not be intentional, and will therefore affect your desired outcome.

Details

Most likely you disabled the Component deliberately while testing things out, but this Alert is making sure you remember that it isn't currently operational.

PsychoPy versions affected

>= 3.1.0

Solutions

Re-enable the Component in the Builder view (component dialog boxes each have a "testing" tab: unselect the "Disable component" setting there). Otherwise just ignore this Alert if you intended it to be disabled.

14.3.11 4310: Builder cannot check your parameter further

Synopsis

This alert is received when an integrity check has failed to calculate a parameter. Most commonly, this is because a variable from code, or a conditions file, has been encountered, and the value of the variable is not available to the integrity checking system.

Details

Longer description with optional links to further information

PsychoPy versions affected

>= 3.2.3

Solutions

This alert is for information only, and does not require any action.

14.3.12 4315: Invalid dollar sign syntax.

Synopsis

This alert is received when a dollar sign has been used incorrectly in a stimulus parameter.

Details

By putting a dollar sign at the beginning of a parameter value, you can indicate that the parameter should be interpreted as code rather than as a string. However, a dollar sign should not appear anywhere else in the parameter value, unless it is either: - After a # when the parameter is interpreted as code (meaning it will be commented out) - Immediately after a ``, an escape character (escaped dollar signs are only valid when the parameter is not interpreted as code, or within quotation marks if it is)

PsychoPy versions affected

All

Solutions

If the dollar sign is a part of a string, you should add an escape character (`) *immediately before it. If it is supposed to be in a comment, it must appear after a* `#. Otherwise, remove any dollar signs which are not at the very beginning of the value.

14.3.13 4320: Non-local font.

Synopsis

This alert is received when a font has been specified which PsychoPy cannot find on your local machine.

Details

Google Fonts is a repository of thousands of free fonts, which PsychoPy is able to access on-the-fly, allowing you to use any font within the Google Fonts library (fonts.google.com) as if it were installed on your machine. However, retrieving this font requires PsychoPy to connect to the internet and send/receive some data, so we raised this alert to give you some warning that this will happen. Font files are tiny, so in most cases this will not noticeable, however if you are on a strictly metered connection or you are not connected to the internet at all then this may cause some issues.

PsychoPy versions affected

All

Solutions

If your computer is connected to the internet and you don't have any limits on how much data you can send/receive, then no action is needed! PsychoPy will happily go off and find this font for you. However, if you are running offline or you have strict data limits, then you should install this font locally instead. Google Fonts can be downloaded for free from fonts.google.com as .ttf files, once downloaded these can be copied to a memory stick and installed on any machine you need by simply opening the file and clicking "Install".

14.3.14 4325: Font not available

Synopsis

The font you requested could not be found in the weight and style you requested, this alert is to warn you that your component will use Open Sans Regular (fonts.google.com/specimen/Open+Sans)

Details

There are a few reasons why you might be receiving this alert: 1. The font you requested does not exist at all, there may be a typo in the font name. 2. The font is one which is not installed on your local machine or available on Google Fonts. 3. The font you requested exists, but not in the style you requested (bold, italic, etc.). For example, if you requested the font Raleway Dots (fonts.google.com/specimen/Raleway+Dots) in bold, you will always receive this error as this font only exists in regular. 4. The font you requested exists on Google Fonts, but could not be retrieved, for example if you are connected to the internet.

PsychoPy versions affected

All

Solutions

The first things to check are that the name of the font is spelled correctly, that the font exists and that you are connected to the internet. You can check which fonts are installed on your machine through the Settings for your operating system, you can check whether the font exists on Google Fonts by going to fonts.google.com and searching for it. If the font exists, you can check (either on Google Fonts or your operating system's font manager) what weights and styles it exists in. If *bold* is ticked in Builder, then the requested font weight is 700 (Bold), if not then the requested weight is 400 (Regular). You can set the font weight more precisely by supplying a numeric font weight to the component rather than just True or False, for example if you wanted the font to be Extra-Light you could supply the value 200.

14.3.15 4330: Recording device not found

Synopsis

The recording device specified in your Microphone component could not be found on your current machine, so when writing the Python code for this experiment to run, the default device will be used instead.

Details

When writing Python code for Microphone components, PsychoPy needs to know the numeric index of its recording device. If the device isn't connected, then this information isn't available, so the Python code is written with the default device index instead.

PsychoPy versions affected

> 2021.2.0

Solutions

The information in your *.psyexp* file won't be changed, just the compiled Python file. If you open the same *.psyexp* file on a machine with the device connected, the Python code generated will contain that device's index, so if you're just testing the experiment on a different machine to the one it will run on then you can ignore this alert.

14.3.16 4335: Component or routine not implemented in Python

Synopsis

This component is not yet implemented in Python, meaning that it will do nothing in local experiments.

Details

You are receiving this alert as you are compiling an experiment to JavaScript, but the experiment contains a component which only works in JavaScript.

PsychoPy versions affected

> 2022.1.0

Solutions

No action required.

14.3.17 4340: Component or routine not implemented in JavaScript

Synopsis

This component is not yet implemented in JavaScript, meaning that it will do nothing in online experiments.

Details

You are receiving this alert as you are compiling an experiment to JavaScript, but the experiment contains a component which only works in Python.

PsychoPy versions affected

> 2022.1.0

Solutions

No action required.

14.3.18 4405: Textbox and keyboard conflict

Synopsis

As editable Textbox components and Keyboard components both listen for key presses, the two can often come into conflict and cause problems.

Details

As editable Textbox components and Keyboard components both listen for key presses, the two can often come into conflict and cause problems. For example, if a Keyboard component is listening for *Enter* to end the routine, what happens when the participant presses *Enter* to start a new line?

PsychoPy versions affected

> 2020.1.0

Solutions

In general, we recommend ending routines containing editable textboxes using a Button, Mouse or ROI component rather than Keyboard. If you are trying to gather keyboard responses as well as text input, consider splitting the two into separate routines to avoid conflicts.

14.3.19 4505: Eyetracking not configured

Synopsis

Experiment includes components or routines which use eyetracking, but no eye tracker is configured.

Details

In order for an Eyetracker Record, Eyetracker Calibrate or Eyetracker Validate routine to work, there needs to be an eyetracker set up in Experiment Settings.

PsychoPy versions affected

>2021.2

Solutions

Either remove any eyetracking components and routines from the experiment flow or set up an eyetracker in the Eyetracker tab of Experiment Settings. If you are testing an eye tracking experiment but do not have an eye tracker connected, you can use MouseGaze to simulate eye movements with the mouse.

14.3.20 4510: Eyetracker not calibrated

Synopsis

The experiment is set up to use an eye tracker, but there is no calibration routine in the experiment flow.

Details

In order to get accurate readings, an eye tracker needs to know what points on the screen correspond to what eye movements. It learns this during a "calibration" routine - in which the participant looks at different points on the screen whose positions are known to the eye tracker. While an eyetracker can sometimes guess, readings will be much more accurate if the eyetracker has been calibrated.

PsychoPy versions affected

>2021.2

Solutions

Either set the *Eyetracker* setting in *Experiment Settings* to be *None* (no eye tracking) or *MouseGaze* (use the mouse as if it were an eye tracker) so that no calibration is needed, or add a calibration routine to the experiment flow. You can find the button to create a calibration routine in the Eyetracking section of the Components panel and can add it to the flow via the Add Routine button.

14.3.21 4520: Animation params not used

Synopsis

Some eyetrackers do not support animations between target stimuli in their calibration routine, so although you have enabled animation in your calibration routine, due to the limitations of the eyetracker you are using any settings for animation will not be used.

Details

PsychoPy versions affected

> 2021.2.0

Solutions

If you are only using the current eyetracker to test an experiment and will be using one which supports target animations, then you don't need to do anything. This alert will stop appearing when you use the other eyetracker.

Otherwise, you just need to be aware that the settings you've specified for animation will not have any effect.

14.3.22 4530: Auto pace param not used

Synopsis

Some eyetrackers do not support manual pacing for calibration, as pacing is always handled automatically. If you have set "Auto-Pacing" to False while using one of these eyetrackers, this setting will be ignored.

Details

PsychoPy versions affected

> 2021.2.0

Solutions

If you are only using the current eyetracker to test an experiment and will be using one which supports manual pacing, then you don't need to do anything. This alert will stop appearing when you use the other eyetracker.

Otherwise, you just need to be aware that pacing will be automatic.

14.3.23 4540: Eyetracking requires window to be fullscreen

Synopsis

Eyetracking requires the expriment window to be fullscreen, otherwise the coordinates returned by the eyetracker won't line up with the coordinates of stimuli within your experiment

Details

PsychoPy versions affected

> 2021.2.0

Solutions

In Experiment Settings, make sure that "Full-screen window" is checked.

14.3.24 4545: Eyetracking requires a monitor config

Synopsis

In order for eyetracking measurements to be accurate, the eyetracker needs to know about the monitor you are using - how wide is it? How tall? What is its resolution? This is so that it can translate eye movements (in degrees of visual angle) into usable data (in height, pix, etc. units). Without this information, the eyetracker can only guess and will therefore give only approximate data.

Details

PsychoPy versions affected

> 2021.2.0

Solutions

Using the Monitor Centre, configure your monitor settings.

14.3.25 4550: Input -> Keyboard Backend not set to 'ioHub'

Synopsis

Experiment is configured to use an eye tracker but Input -> Keyboard Backend experiment setting is set to 'PsychToolbox'.

Details

Eye trackers run using ioHub, which also handles Keyboard events, so Input -> Keyboard Backend experiment setting should be 'ioHub'.

PsychoPy versions affected

>=2022.1

Solutions

Switch the Input -> Keyboard Backend experiment setting to use 'ioHub'

14.3.26 4605: Transcription service not compatible online

Synopsis

Some audio transcription services can only be run online, some can only be run locally and some can be run either way.

If you are receiving this alert, it means that the audio transcription service you selected is one which either only works locally, but you are trying to run your experiment online, or it is not supported at all.

Details

PsychoPy versions affected

>2021.2.0

Solutions

To silence this alert, choose a transcription service which can be run online, such as Google or Azure.

14.3.27 4610: Transcription service not compatible locally

Synopsis

Some audio transcription services can only be run online, some can only be run locally and some can be run either way.

If you are receiving this alert, it means that the audio transcription service you selected is one which only works online, but you are trying to run your experiment locally, or it is not supported at all.

Details

PsychoPy versions affected

>2021.2.0

Solutions

To silence this alert, choose a transcription service which can be run online, such as the built-in transcriber.

14.3.28 4615: API key not found

Synopsis

If you're receiving this alert, it means you have selected an audio transcriber which requires an API key to function but have not supplied an API key in preferences.

Details

Some audio transcribers work fine without a key, they're just a publicly available Python or JavaScript function which runs using your own computer's processing power, but others use algorithms which are either confidential or require huge amounts of processing power. This means that they need to be run on a server, often one which you've paid to access. If you have such a subscription, then your chosen transcription service will have provided you with an "API key" - this is a unique code, like a password, which tells their server who you are. If you are using such a transcription service within PsychoPy, then PsychoPy needs to be able to use your API key to request transcription.

PsychoPy versions affected

> 2021.2.0

Solutions

In PsychoPy, go to File -> Preferences -> General. Here you will find some preferences starting with *transcrKey* - in the one matching your choice of transcription service, copy and paste the API key you were given when you subscribed to that service.

14.3.29 4705: Column name from conditions file clashes with variable name

Synopsis

The name of a column in your conditions file already exists in this experiment.

Details

The name of a column in your conditions file already exists in this experiment. This means that either the existing variable will be overwritten by the value of this parameter, or that this parameter will be overwritten by the existing variable.

PsychoPy versions affected

All

Solutions

Please choose a different column name.

14.3.30 4710: Column name from conditions file likely to clash with derived variable names

Synopsis

The name of a column in your conditions file already exists in this experiment.

Details

The name of a column in your conditions file already exists in this experiment. This means that either the existing variable will be overwritten by the value of this parameter, or that this parameter will be overwritten by the existing variable.

PsychoPy versions affected

All

Solutions

Please choose a different column name.

CHAPTER

FIFTEEN

FREQUENTLY ASKED QUESTIONS (FAQS)

15.1 Why is the bits++ demo not working?

So far supports bits++ only in the bits++ mode (rather than mono++ or color++). In this mode, a code (the T-lock code) is written to the lookup table on the bits++ device by drawing a line at the top of the window. The most likely reason that the demo isn't working for you is that this line is not being detected by the device, and so the lookup table is not being modified. Most of these problems are actually nothing to do with /per se/, but to do with your graphics card and the CRS bits++ box itself.

There are a number of reasons why the T-lock code is not being recognised:

- the bits++ device is in the wrong mode. Open the utility that CRS supply and make sure you're in the right mode. Try resetting the bits++ (turn it off and on).
- the T-lock code is not fully on the screen. If you create a window that's too big for the screen or badly positioned then the code will be broken/not visible to the device.
- the T-lock code is on an 'odd' pixel.
- the graphics card is doing some additional filtering (win32). Make sure you turn off any filtering in the advanced display properties for your graphics card
- the gamma table of the graphics card is not set to be linear (but this should normally be handled by , so don't worry so much about it).
- you've got a Mac that's performing temporal dithering (new Macs, around 2009). Apple have come up with a new, very annoying idea, where they continuously vary the pixel values coming out of the graphics card every frame to create additional intermediate colours. This will break the T-lock code on 1/2-2/3rds of frames.

15.2 Why is my stimulus not showing?

If your stimulus in PsychoPy isn't displaying and there's no error message, consider these often overlooked factors:

Units Setting:

The units used for your stimulus (e.g., pixels, height, norm) can significantly impact its display. If the units are set incorrectly, the stimulus might be too small or positioned off-screen. For example, specifying a stimulus size in pixels when your experiment uses height units could result in an inappropriately scaled stimulus.

Duplicate Variable Names in Conditions File:

Using the same name for a variable in your conditions file and a component in your experiment can lead to conflicts. For instance, if you have an image component named 'image' and also a column in your conditions file named 'image', PsychoPy may get confused when trying to present the 'image' variable. This doesn't typically generate an error, but it can result in your stimulus not being displayed as expected. To avoid this, ensure that your variable names in the conditions file are unique and not identical to any of your component names.

15.3 My experiment is crashing - how do I know where the problem is?

If your experiment in PsychoPy is crashing, the most effective way to identify the issue is by checking the Runner window. This window displays real-time logs and error messages while your experiment runs. Follow these steps to use the Runner window for troubleshooting:

Run Your Experiment: Start your experiment as you normally would. If it crashes, the Runner window will capture and display relevant error messages.

Locate the Error Message: In the Runner window, look for messages that are marked as errors. These are typically highlighted in red for visibility and will usually contain details about the nature of the problem.

Understand the Error Message: The error message will often indicate where in your script or routine the problem occurred. It can provide clues such as the line number in your code or the specific component in your routine that caused the crash.

Using the last_app_load.log file: The last_app_load.log file in PsychoPy is a log file that records information about the software's operations and events, particularly during the startup or loading phase of the application. It can be really useful if the error message you're receiving isn't too helpful, or if your experiment is causing the whole app to crash. It can be found by:

Windows: - Opening a File Explorer window and typing %AppData% in the address bar. Open the folder called psychopy3 which should reveal the last_app_load.log file. You can open this in any text editor. *Mac:* - Open a Terminal and type cat ~/.psychopy3/last_app_load.log

Suggested Troubleshooting Steps:

- Syntax Errors: If the issue is a syntax error in your code, the error message will typically point to the exact line or command that needs correction.
- Resource Issues: If the error relates to missing files or resources, ensure all required files are in the correct location and properly linked in your script.
- Logical Errors: For logical errors, where the syntax is correct but the experiment does not behave as expected, re-check the logic of your code or experiment flow.
- Consult Documentation and Community: If the error message is unclear or you are unable to resolve the issue, consult the PsychoPy documentation for further guidance. 'The PsychoPy forum <https://discourse.psychopy.org>`_ can also be a valuable resource for seeking help from other users who might have faced similar issues.

15.4 Can run my experiment with sub-millisecond timing?

This question is common enough and complex enough to have a section of the manual all of its own. See *Timing Issues and synchronisation*

15.5 Why am I getting a variable not defined error?

Receiving a "variable not defined" error in PsychoPy is a common issue, often resulting from a few specific causes. Here are some points to consider when troubleshooting this error:

Capital Letters and Spaces in Variable Names: Check your variable names for capital letters and spaces. PsychoPy is case-sensitive, meaning that "VariableName" and "variablename" are treated as different variables. Also, spaces in variable names are not allowed, so ensure your variable names are continuous strings.

Variable Scope - 'Set Every Repeat':

Ensure that your variables are set to update at the correct time. If a variable changes every trial, it should be set to 'set every repeat' in the component settings. This ensures the variable updates its value with each iteration of your experiment's loop, rather than right at the start of your experiment where it might not have been defined yet.

Defining Variables in the 'Begin Experiment' Tab:

A common mistake is placing code that relies on variables from condition files in the 'Begin Experiment' tab instead of the 'Begin Routine' tab. Variables read from condition files should typically be used in the 'Begin Routine' tab, as they are not yet available at the experiment's start ('Begin Experiment'). Placing such code in the 'Begin Experiment' tab will result in a "variable not defined" error, as the variable has not been read in yet.

Loop and Routine Configuration:

Check if you've forgotten to wrap a loop around a routine or attach condition files correctly. Loops in PsychoPy are used to repeat a routine with different conditions. If your experiment expects a variable from a conditions file but the loop isn't set up correctly, PsychoPy won't be able to find the variable.

15.6 Will updating PsychoPy break my existing experiments?

Updating PsychoPy is generally a positive step towards enhancing your experimental setup, and it's unlikely to break your existing experiments, especially if you are using the Builder. Here's why:

Control Over Versioning with 'Use Version' Setting: PsychoPy gives you control over which version you run your experiment with, thanks to the 'Use Version' setting in the experiment settings. This feature allows you to specify a particular version of PsychoPy for each experiment, ensuring compatibility and stability, even after updating the software.

Advantages of Using Builder: Experiments created with PsychoPy's Builder are especially robust against version updates. The Builder's graphical interface abstracts much of the underlying code, making experiments less prone to issues related to code changes in new versions. This means your experiments are more likely to run smoothly, even after an update.

Benefits of Updating: Updating PsychoPy brings you the latest features, performance improvements, and bug fixes, enhancing the overall functionality and user experience. New versions often include optimized components, new capabilities, and improved efficiency, which can significantly benefit your experimental design and data collection.

Continued Compatibility and Testing: While updates aim for backward compatibility, it's good practice to test your experiments after updating. This ensures everything runs as expected. The community and developers continually work to maintain compatibility across versions, so you can update with confidence.

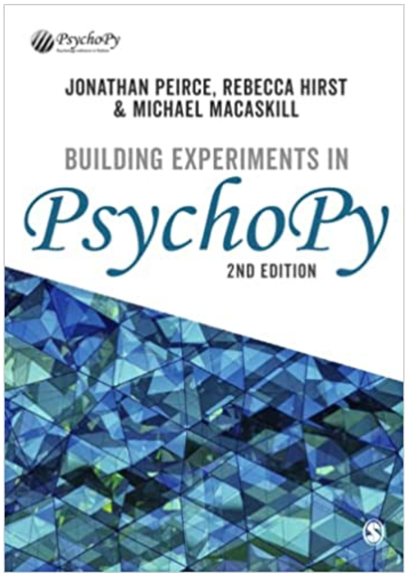
Enhanced Performance and Features: Each update to PsychoPy not only aims to fix known issues but also introduces new functionalities that can make your experiment design more efficient and effective. Leveraging these new features can lead to more sophisticated experimental designs and smoother execution.

CHAPTER SIXTEEN

RESOURCES (E.G. FOR TEACHING)

There are a number of further resources to help learn/teach about PsychoPy.

If you also have PsychoPy materials/course then please let us know so that we can link to them from here too!



16.1 Workshops

We are currently running virtual workshops in several formats, see our workshops pages for details

16.2 Youtube tutorials

- There is our YouTube PsychoPy playlist showing how to build basic experiments in the *Builder* interface.
- Jason Ozubko has added a series of great PsychoPy Builder video tutorials too
- Damien Mannion added a similarly great series of PsychoPy programming videos on YouTube
- … and a searching YouTube for PsychoPy reveals many more!

16.3 Materials for Builder

The most comprehensive guide is the book Building Experiments in PsychoPy by Peirce, Hirst, and MacAskill.
The book is suitable for a wide range of needs and skill sets, with 3 sections for:

- The Beginner (suitable for undergraduate teaching)
- The Professional (more detail for creating more precise studies)
- The Specialist (with info about specialist needs such as studies in fMRI, EEG, . . .)

At School of Psychology, University of Nottingham, PsychoPy is now used for all first year practical class teaching. The classes that comprise that first year course are provided below. They were created partially with funding from the former Higher Education Academy Psychology Network. Note that the materials here will be updated frequently as they are further developed (e.g. to update screenshots etc) so make sure you have the latest version of them!

There's a set of tools for teaching psychophysics using PsychoPy and a PsychoPysics poster from VSS. Thanks James Ferwerda

16.4 Materials for Coder

- Please see the page on officialWorkshops for further details on coming to an intensive residential Python workshop in Nottingham.
- Marco Bertamimi's book, Programming Illusions for Everyone is a fun way to learn about stimulus rendering in PsychoPy by learning how to create visual illusions
- Programming for Psychology in Python - Vision Science has lessons and screencasts on PsychoPy (by Damien Mannion, UNSW Australia).

16.5 Previous events

- ECEM, August 2013 : Python for eye-tracking workshop with (Sol Simpson, Michael MacAskill and Jon Peirce).
- VSS

For developers:

CHAPTER

SEVENTEEN

CONTRIBUTING TO OPEN SOURCE CODE

There are many ways that you can contribute to - even if you're not a developer! See below for handy how-tos on open-scource contribution to .

To discuss ideas in depth, check out the dedicated developers section of the forum.

Happy Coding Folks!!

17.1 Adding a plugin

From v2023.2, has supported plugins - these are entirely separate Python modules which, when installed, add content to PsychoPy (such as new Components, functions and classes). See below for an in-depth guide on creating and publishing your own plugin:

17.1.1 Creating Plugins for

Plugins provide a means for developers to extend , adding new features and customizations without directly modifying the installation.

The plugin system works by searching for "*:ref:_entryPoints*". If your package defines an entry point targeting , it will be found and imported when `psychopy.plugins.activatePlugins` is called - which the PsychoPy app calls on starting, as do any Builder experiments. Any changes made to with plugins do not persist across sessions, meaning if Python is restarted, will return to its default behaviour unless `activatePlugins()` is called again.

Read usingplugins for more information about plugins before proceeding on this page.

Creating your plugin package

A plugin is ultimately just a type of Python package, just one which interacts with , so you can create one in the same way you would make any other Python package. Check out the Python Packaging User Guide for a comprehensive guide to making Python packages in general. Below we'll go through the key steps of making a plugin package specifically.

Naming your plugin package

To help users find your plugin and to make it clear that it is a plugin, your plugin package name should start with `psychopy-` followed by a word or a few words which describe what it is (separated by `-`). The module within your plugin package (the folder that gets imported when a user does `import <your plugin>`) should have the same name as the package but with `-` replaced by `_`. Below are some example plugin names and what they do:

Table 17.1: Title

Package name	Module name	Description
psychopy-visionscience	psychopy_visionsci	Adds various stimuli which are useful for vision scientists
psychopy-face-api	psychopy-face-api	Add support for Face API in PsychoJS experiments created using Builder
psychopy-cedrus	psychopy_cedrus	Adds support for Cedrus devices (button boxes, photodiodes, voicekeys, etc.)
psychopy-monkeys	psychopy_monkeys	Adds various “response monkey” Components, which save time testing by immitating participant responses when running in pilot mode

Structuring your plugin package

A plugin package is no different structurally from any other package, so see the official Packaging Python Projects guide for details on how to structure a Python package in general. For plugins specifically, we strongly recommend starting by cloning the template plugin repo and modifying it rather than starting from scratch.

pyproject.toml

Python looks for a file at the root level of your plugin package called `pyproject.toml`, which contains the information it needs (name, version, etc.) to set up the package. Essentially, it’s what turns a Python module into a Python package. For an example of a finished `pyproject.toml` file, check out the one from the template plugin repo.

Entry points

Defining an entry point is essentially telling Python “pretend that `x.y` is also located at `z.y`”, similar to if the file `z.py` had `from x import y` at the top. This allows an external package to edit what can be imported from without changing any of ’s code. can also get a list of all plugin packages which define entry points to a certain place, which in some cases will help it find your plugin. For example, if you wanted to add a Component, you would define an entry point to `psychopy.experiment.components` and Builder would then find your plugin Component by looking for that entry point.

Base classes

In many cases, the element your plugin package adds may need to be a subclass of a particular base class to be detected and used properly. For example, a new Component should be a subclass of `psychopy.experiment.components.BaseComponent`. Similarly, new backend for a hardware Component which supports multiple backends (such as :ref:_buttonboxcomponent or :ref:_voicekeycomponent) would need to be a subclass of `psychopy.experiment.components.plugins.DeviceBackend`.

If you’re unsure what to subclass, try looking for another similar element in or in another plugin and see what they subclass (for example, `psychopy.experiment.components.buttonBox.KeyboardButtonBoxBackend` is a subclass of `DeviceBackend`).

Testing your plugin package

Once you’re ready to try your plugin package out, you can install it via the Plugins & Packages Manager in Builder (opened via the Tools menu item or the “Get more…” button at the top of the Components panel). In the “Packages” tab, click the “Install from file” button and select the `pyproject.toml` file for your plugin (you will need to change the file type dropdown to look for “Python projects” rather than a “Wheel files” to see it). Doing so will prompt PsychoPy to perform an editable install of your plugin, meaning that any edits you make to your plugin will be immediately visible once you restart PsychoPy, without requiring the plugin to be reinstalled. The only exception is any changes made to the pyproject.toml file itself - these do require a reinstall to register unfortunately.

Publishing your plugin package

plugin packages are built like any other package and hosted via the Python Package Index (PyPI). While you can absolutely build and package it yourself if you're comfortable and familiar doing so, we recommend copying the .github folder from the template plugin repo, as this defines a GitHub action to build and publish your plugin package for you whenever you make a new release on GitHub. To allow this to work, there's just a few configuration steps to follow.

Give action permissions

In order for the publishing action to work, it needs to be given certain permissions within your plugin's repo. To enable these, go to your repo, then "Settings" at the top, then "Actions -> General" on the left. Then:

- Make sure that "Allow all actions and reusable workflows" is ticked (or, if you need to disable some actions for other reasons, at least allow actions from yourself or allow specific actions and specify pypi.yml as enabled).
- Make sure "Read and write permissions" (under "Workflow permissions") is ticked

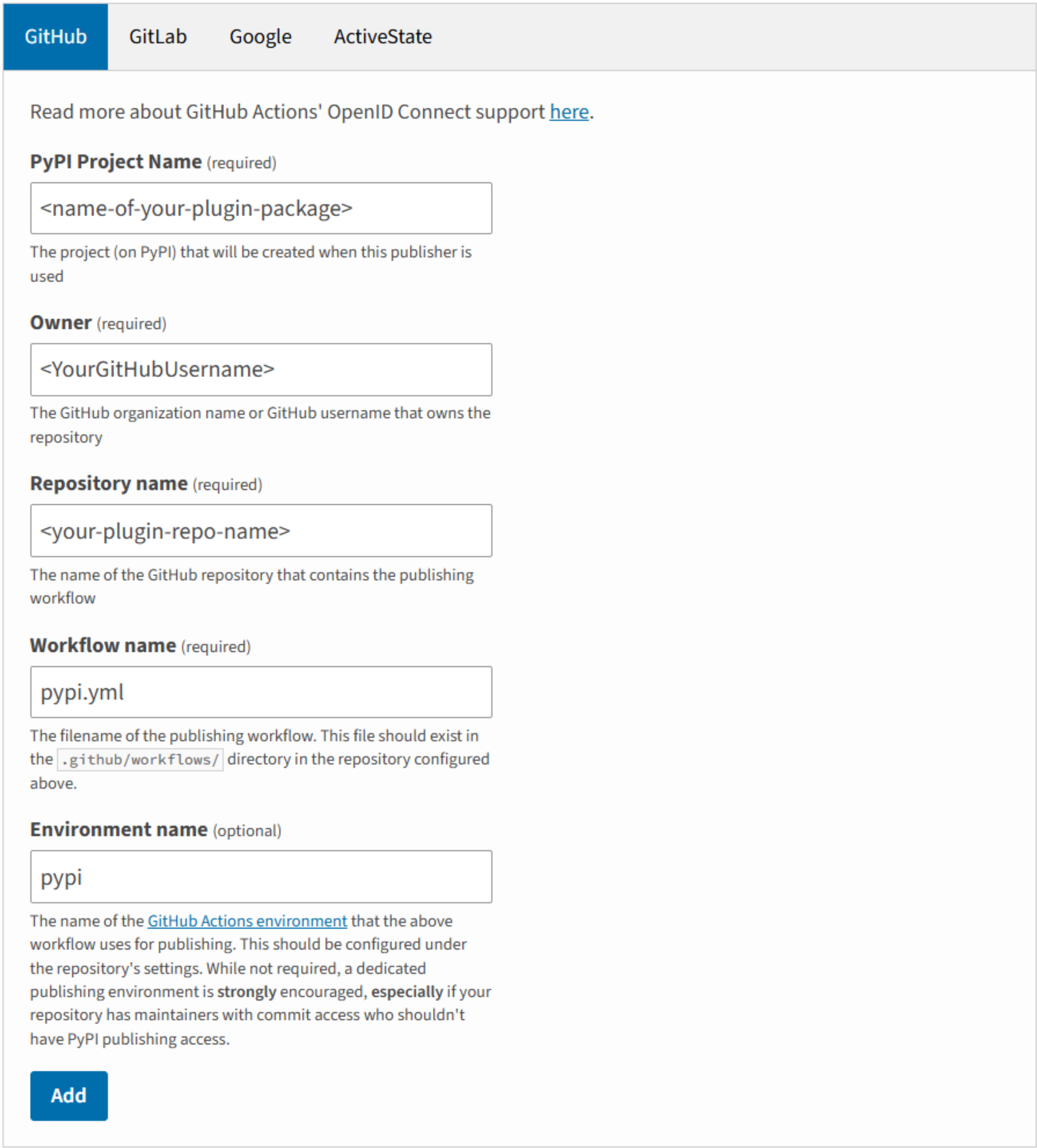
Create a GitHub environment

In order for PyPI to recognise your plugin, and prevent anyone else pushing to it from any old repo, it will check that the GitHub action is running in the correct "environment". This environment doesn't need any special configuration, it just needs to exist and have a name (we recommend just calling it pypi). To create an environment, just go to your repo, then "Settings" at the top, then "Environments" on the left. Click "New Environment" in the top right, give it a name and click "Save". That's all you need!

Set up a trusted publisher

Now that your GitHub repo is all set up, you need to setup PyPI to look for your repo as the publisher of your plugin package. If you don't have an account with PyPI, you can create one here. Once logged in, click on your username in the top right and then "Your Projects". On the left, click "Publishing". This should take you to an interface for managing "publishers" - this is essentially a mapping which tells PyPI which GitHub accounts and environments to accept pushes from when publishing a new version of a specific package.

Scroll down to the "Pending publishers" section and choose "GitHub" from the tabs on the control there. It should look like this:



Read more about GitHub Actions' OpenID Connect support here.

PyPI Project Name (required)

<name-of-your-plugin-package>

The project (on PyPI) that will be created when this publisher is used

Owner (required)

<YourGitHubUsername>

The GitHub organization name or GitHub username that owns the repository

Repository name (required)

<your-plugin-repo-name>

The name of the GitHub repository that contains the publishing workflow

Workflow name (required)

pypi.yml

The filename of the publishing workflow. This file should exist in the `.github/workflows/` directory in the repository configured above.

Environment name (optional)

pypi

The name of the GitHub Actions environment that the above workflow uses for publishing. This should be configured under the repository's settings. While not required, a dedicated publishing environment is **strongly** encouraged, **especially** if your repository has maintainers with commit access who shouldn't have PyPI publishing access.

but with the fields flanked by <> replaced by the relevant information for your plugin. Once you click “Add”, you should be good to go!

Make a release on GitHub

To trigger the relevant GitHub action to publish your plugin package, you need to make a release. See the documentation from GitHub for information on how to do this. Remember to tag the release with the version number! The first release will most likely be `0.0.1`.

Listing a plugin in Builder

Once your plugin is published on PyPI, it can be installed by anyone (via `pip install <package-name>` - but it won't appear in the list of plugins from Builder as at this point it's no different than any of the thousands of other Python packages on PyPI. Builder gets its list of plugins from a file in the PsychoPy Plugins repo called `plugins.json`. This file contains a list of plugins alongwith information about them, links to documentation, an icon for the plugin and author, etc.

To add your plugin to this list, simply fork this repo, edit the file to include information about your plugin, and submit it as a pull request. We'll give your plugin a quick check over for malicious code and, assuming it's all fine, will accept your pull request and your plugin will be immediately available from Builder!

17.2 Contributing directly to

17.2.1 Getting set up

Follow the guides below to get your local setup ready to start making, testing & submitting changes to .

Fork the respository   Create your own linked copy of the code to edit

./setupfork

Setup a dev install   Setup your preferred coding environment to run from code

./environment/index

Open a pull request   Request for the changes on your fork to be pulled into the main code

./pullrequest

17.2.2 Ways to contribute

Improving our documentation pages is a great way to become a contributor to , or any other Open Source software!

For a simple step-by-step guide to correcting typos, or adding an extra paragraph to our documentation etc., click the 'Adding Documentation' link below!

Making edits to PsychoPy's documentation

First you'll need to have your own "fork" of - this is a copy of the code which is yours to edit, without affecting the main code. Click here for a how-to for forking |PsychoPy|

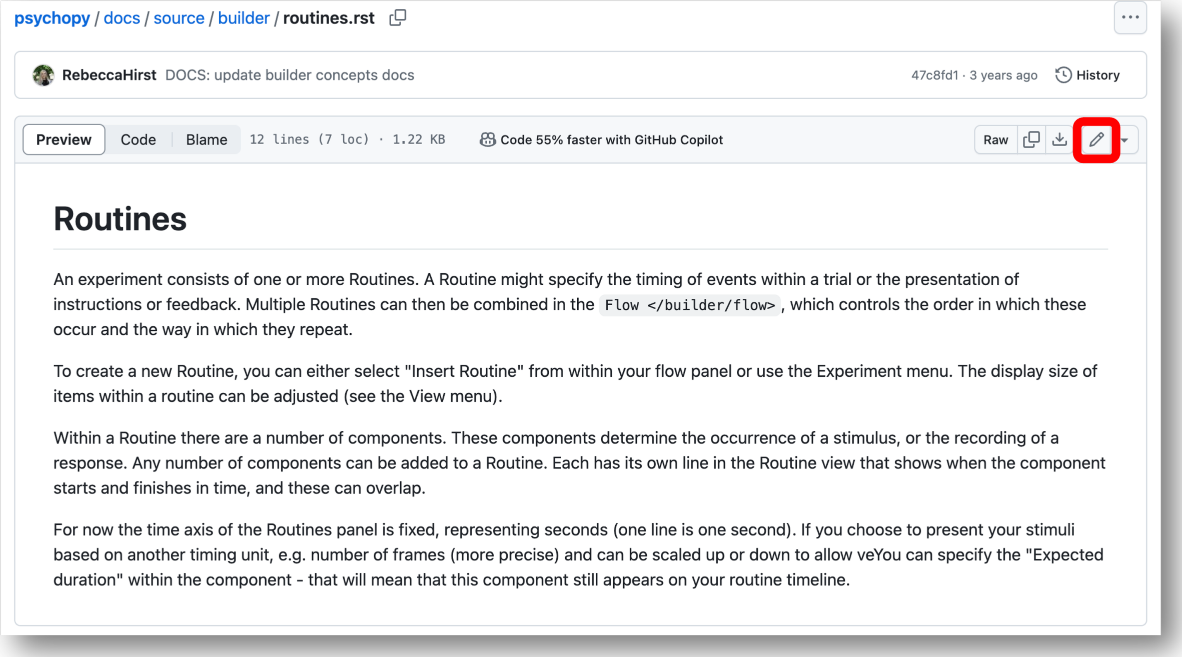
If you only want to change a single file:

For example, you've spotted a typo in one of the documentation pages, want to fix a broken link, or add in a few extra paragraphs to improve the clarity of the page. Essentially anything that involves you changing just one file. The easiest way to do this is directly within GitHub itself.

If you're not already aware, it's worth noting how the URLs in the documentation are linked to the folder structure of the PsychoPy repository: All of the documentation files are located in the folder docs > source. Each html page is created from a reStructured text (rst) file, and these files are stored in several folders. For instance, the page: https://psychopy.org/builder/routines.html is created from the routines.rst file located in: docs/source/builder.

Let's imagine that you want to change that routines.rst file:

1. In your fork of the PsychoPy repository, click through to the routines.rst file located in: docs/source/builder and click on the 'edit' pencil icon:
2. Make the changes you need to.


psychopy / docs / source / builder / routines.rst
RebeccaHirst DOCS: update builder concepts docs
47c8fd1 · 3 years ago
History
Preview
Code
Blame
12 lines (7 loc) · 1.22 KB
Code 55% faster with GitHub Copilot
Raw

Routines

An experiment consists of one or more Routines. A Routine might specify the timing of events within a trial or the presentation of instructions or feedback. Multiple Routines can then be combined in the `Flow </builder/flow>`, which controls the order in which these occur and the way in which they repeat.

To create a new Routine, you can either select "Insert Routine" from within your flow panel or use the Experiment menu. The display size of items within a routine can be adjusted (see the View menu).

Within a Routine there are a number of components. These components determine the occurrence of a stimulus, or the recording of a response. Any number of components can be added to a Routine. Each has its own line in the Routine view that shows when the component starts and finishes in time, and these can overlap.

For now the time axis of the Routines panel is fixed, representing seconds (one line is one second). If you choose to present your stimuli based on another timing unit, e.g. number of frames (more precise) and can be scaled up or down to allow veYou can specify the "Expected duration" within the component - that will mean that this component still appears on your routine timeline.

3. Click on the 'Commit changes…' button that becomes active when you've made changes:
4. Add a commit message: For documentation changes we use 'DOC:' followed by a brief description of what we've changed (see usingRepos for more on commit messages):
5. Commit your changes: This commits the changes you've made to your PsychoPy repository. You now need to make a pull request so that those changes can be merged into the main PsychoPy repository.

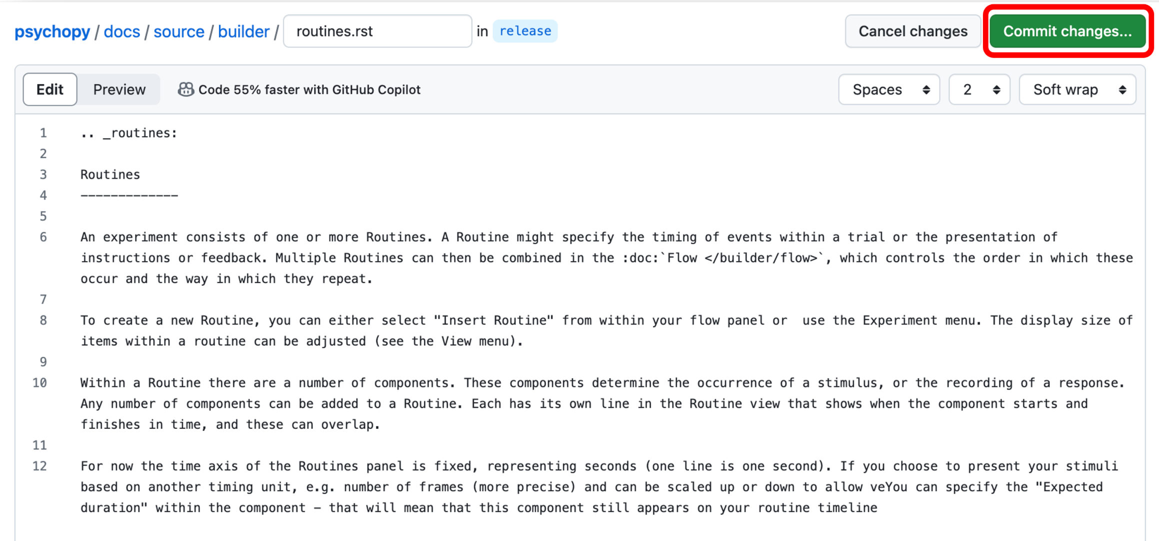
Make a pull request:

1. Click back into your main PsychoPy repository:
2. Click on the message that says 'Compare and pull request':
3. As you want to contribute to the documentation on the website, you'll need to select 'release' as your base branch:
4. Make sure the title of your pull request matches the one that you put as your commit message. You can add a description of your changes if you like, too. Then click 'Create pull request':
5. You've made a pull request! Your code will be reviewed, and you'll receive an email when your changes have been pulled in!

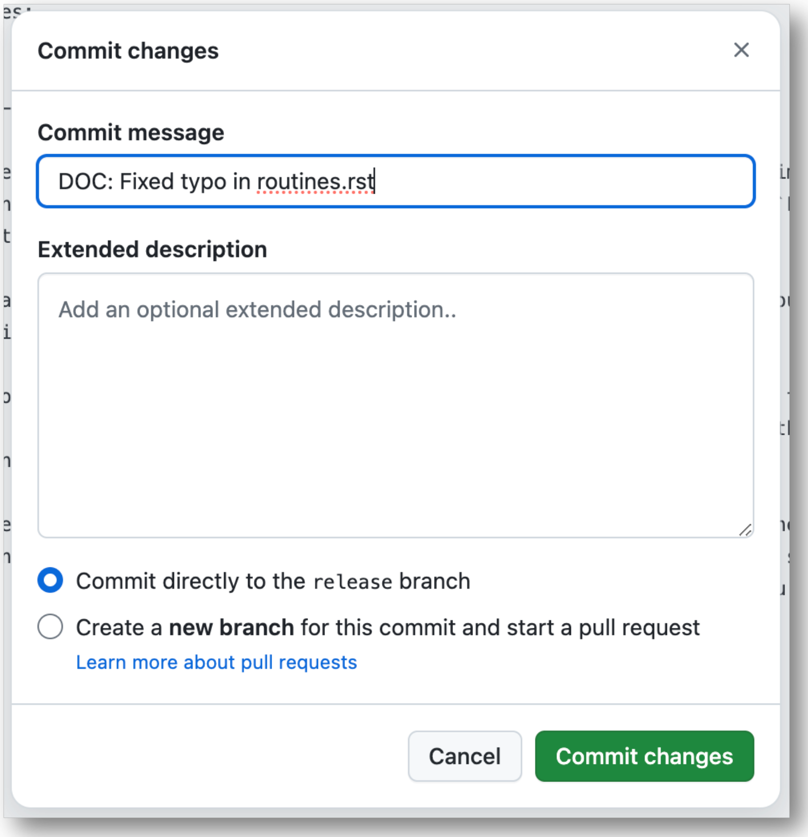
Contributing to the Test Suite

Why do we need a test suite?

With any bit of software, no matter how perfect the code seems as you're writing it, there will be bugs. We use a test suite to make sure that we find as many of those bugs as we can before users do, it's always better to catch them in development than to have them mess up someone's experiment once the software is out in the wild. Remember - when a user finds a bug, they react like this:


psychopy / docs / source / builder / routines.rst in release
Cancel changes
Commit changes...
Edit
Preview
Code 55% faster with GitHub Copilot
Spaces
2
Soft wrap

```
.. _routines:

Routines
-------------

An experiment consists of one or more Routines. A Routine might specify the timing of events within a trial or the presentation of instructions or feedback. Multiple Routines can then be combined in the :doc:`Flow </builder/flow>`, which controls the order in which these occur and the way in which they repeat.

To create a new Routine, you can either select "Insert Routine" from within your flow panel or  use the Experiment menu. The display size of items within a routine can be adjusted (see the View menu).

Within a Routine there are a number of components. These components determine the occurrence of a stimulus, or the recording of a response. Any number of components can be added to a Routine. Each has its own line in the Routine view that shows when the component starts and finishes in time, and these can overlap.

For now the time axis of the Routines panel is fixed, representing seconds (one line is one second). If you choose to present your stimuli based on another timing unit, e.g. number of frames (more precise) and can be scaled up or down to allow veYou can specify the "Expected duration" within the component – that will mean that this component still appears on your routine timeline
```



Commit message

DOC: Fixed typo in routines.rst

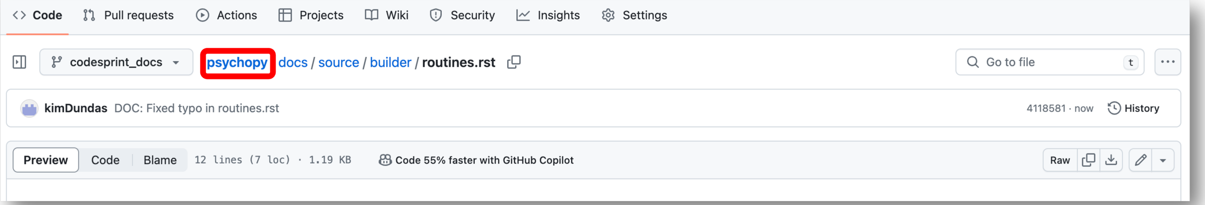
Extended description

Add an optional extended description..

● Commit directly to the `release` branch

○ Create a **new branch** for this commit and start a pull request

Learn more about pull requests

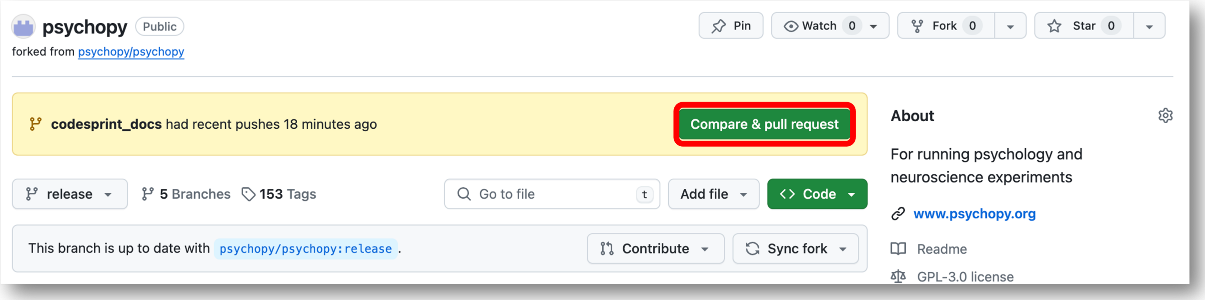


psychopy / docs / source / builder / routines.rst

kimDundas DOC: Fixed typo in routines.rst 4118581 · now

12 lines (7 loc) · 1.19 KB

Code 55% faster with GitHub Copilot

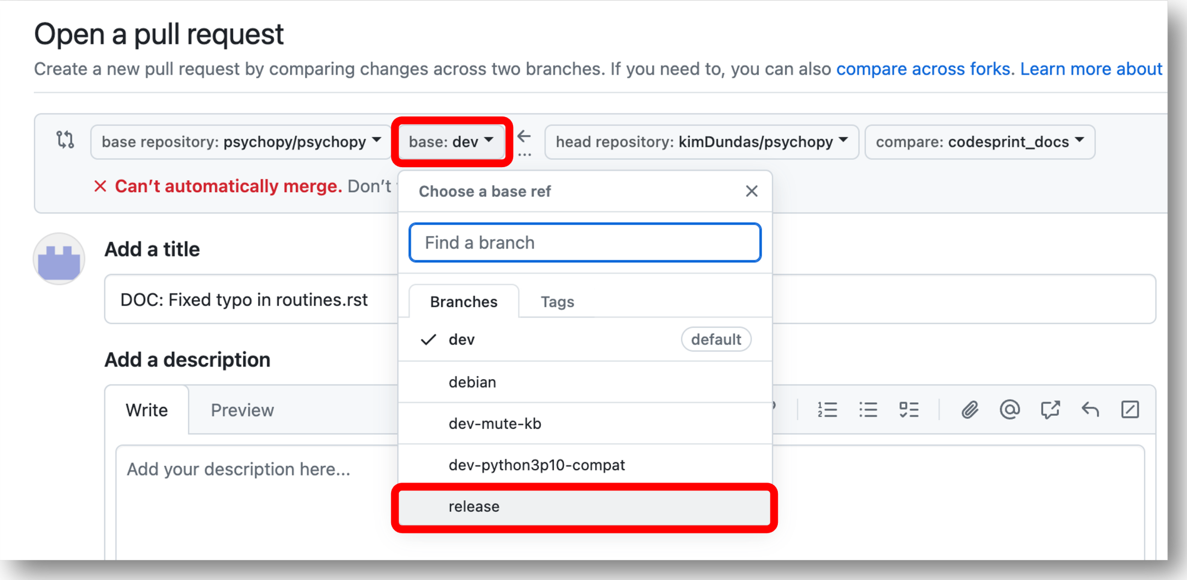


forked from psychopy/psychopy

codesprint_docs had recent pushes 18 minutes ago

This branch is up to date with `psychopy/psychopy:release` .

For running psychology and neuroscience experiments



Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forks. Learn more about

base repository: psychopy/psychopy ▾ base: dev ▾ ← head repository: kimDundas/psychopy ▾ compare: codesprint_docs ▾

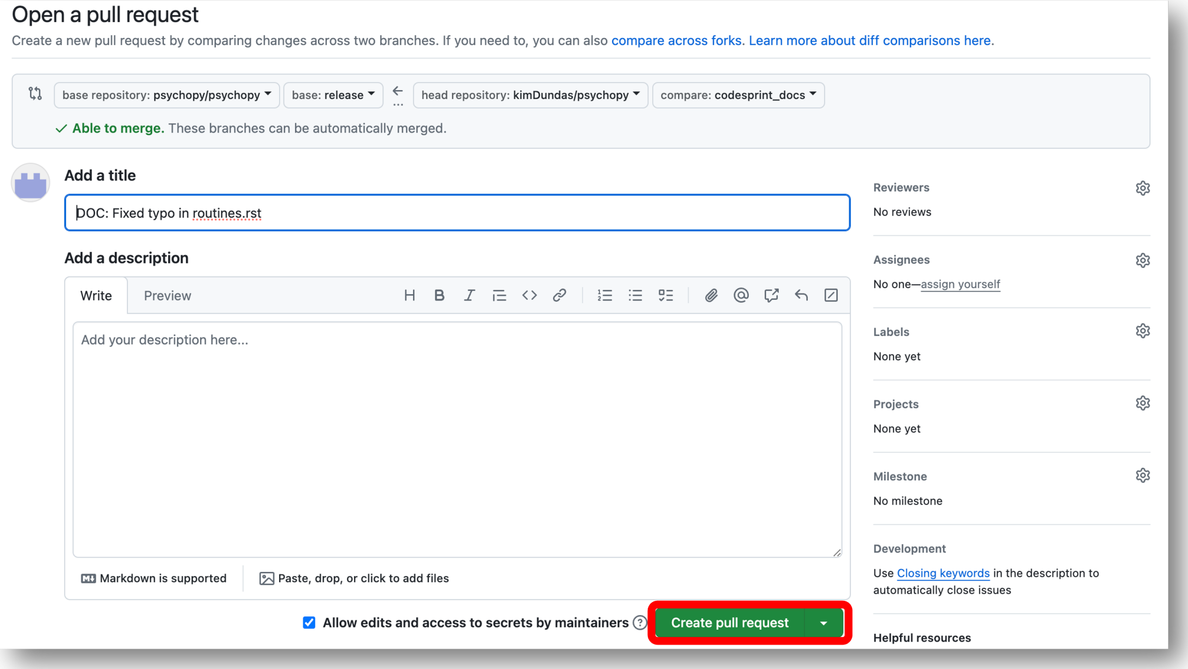
✕ Can't automatically merge. Don't

Add a title

DOC: Fixed typo in routines.rst

Add a description

Add your description here...



Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forks. Learn more about diff comparisons here.

base repository: psychopy/psychopy ▾ base: release ▾ ← head repository: kimDundas/psychopy ▾ compare: codesprint_docs ▾

✓ Able to merge. These branches can be automatically merged.

Add a title

DOC: Fixed typo in routines.rst

Add a description

Add your description here...

Markdown is supported

Paste, drop, or click to add files

☑ Allow edits and access to secrets by maintainers

Reviewers

No reviews

Assignees

No one—assign yourself

Labels

None yet

Projects

None yet

Milestone

No milestone

Development

Use Closing keywords in the description to automatically close issues

Helpful resources

. . . but when the test suite finds a bug, developers react like this:

The more bugs the test suite finds, the better!

How does it work?

The test suite uses a Python module called pytest to run tests on various parts of the code. These tests work by calling functions, initialising objects and generally trying to use as much of the code in the PsychoPy repo as possible - then, if an uncaught error is hit at any point, *pytest* will spit out some informative text on what went wrong. This means that, if the test suite can run without error, then the software can do everything done in the test suite without error.

To mark something as a test, it needs three things:

1. It must be somewhere in the folder *psychopy/psychopy/tests*
2. It must contain the word *test* in its name (i.e. the class name and function names)
3. It must be executable in code, a function or a method

So, for example, if you were to make a test for the *visual.Rect* class, you might call the file *test_rect.py* and put it in *psychopy/psychopy/tests/test_all_visual*, and the file might look like this:

```
from psychopy import visual  # used to draw stimuli

def test_rect():
    # Test that we can create a window and a rectangle without error
    win = visual.Window()
    rect = visual.Rect(win)
    # Check that they draw without error
    rect.draw()
    win.flip()
    # End test
    win.close()
```

Using *assert*

Sometimes there's more to a bit of code than just running without error - we need to check not just that it doesn't crash, but that the output is as expected. The *assert* function allows us to do this. Essentially, *assert* will throw an *AssertionError* if the first input is *False*, with the text of this error determined by the second input. So, for example:

```
assert 2 < 1, "2 is not less than 1"
```

will raise:

```
AssertionError: 2 is not less than 1
```

In essence, an *assert* call is the same as saying:

```
if condition == False:
    raise AssertionError(msg)
```

What this means is that we can raise an error if a value is not what we expect it to be, which will cause the test to fail if the output of a function is wrong, even if the function ran without error.

You could use *assert* within the *test_rect* example like so:

```
# Set the rectangle's fill color
rect.colorSpace = 'rgb'
rect.fillColor = (1, -1, -1)
# Check that the rgb value of its fill color is consistent with what we set
assert rect._fillColor == colors.Color('red'), f"Was expecting rect._fillColor to have␣
↪an rgb value of '(1, -1, -1)', but instead it was '{rect._fillColor.rgb}'"
```

Meaning that, if something was wrong with *visual.Rect* such that setting its *fillColor* attribute didn't set the rgb value of its fill color correctly, this test would raise an *AssertionError* and would print both the expected and actual values. This process of comparing actual outputs against expected outputs is known as "end-to-end" (e2e) testing, while simply supplying values to see if they cause an error is called "unit" testing.

Using classes

In addition to individual methods, you can also create a *class* for tests. This approach is useful when you want to avoid making loads of objects for each test, as you can simple create an object once and then refer back to it. For example:

```
class TestRect:
    """ A class to test the Rect class """
    @classmethod
    def setup_class(self):
        """ Initialise the rectangle and window objects """
        # Create window
        self.win = visual.Window()
        # Create rect
        self.rect = visual.Rect(self.win)

    def test_color(self):
        """ Test that the color or a rectangle sets correctly """
        # Set the rectangle's fill color
        self.rect.colorSpace = 'rgb'
        self.rect.fillColor = (1, -1, -1)
        # Check that the rgb value of its fill color is consistent with what we set
```

(continues on next page)

(continued from previous page)

```
        assert self.rect._fillColor == colors.Color('red'), f"Was expecting rect._
↪fillColor to have an rgb value of '(1, -1, -1)'," \
                                                    f" but instead it was '{self.rect._
↪fillColor.rgb}'"
```

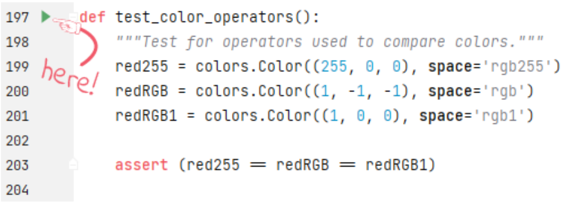
Of course, you could create a window and a rectangle for each function and it would work just the same, but only creating one means the test suite doesn't have as much to do so it will run faster. Test classes work the same as any other class definition, except that rather than *__init__*, the constructor function should be *setup_class*, and this should be marked as a *@classmethod* as in the example above.

Exercise

Practicing writing tests? Try extending the above class to test if a created rectangle has 4 vertices.

Running tests in PyCharm

One of the really useful features on PyCharm is its ability to run tests with just a click. If you have *pytest* installed, then any valid test will have a green play button next to its name, in the line margins:



```
def test_color_operators():
    """Test for operators used to compare colors."""
    red255 = colors.Color((255, 0, 0), space='rgb255')
    redRGB = colors.Color((1, -1, -1), space='rgb')
    redRGB1 = colors.Color((1, 0, 0), space='rgb1')

    assert (red255 == redRGB == redRGB1)
```

Clicking this button will start all the necessary processes to run this test, just like it would run in our test suite. This button also appears next to test classes, clicking the run button next to the class name will create an instance of that class, then run each of its methods which are valid tests.

Test utils

The test suite comes with some handy functions and variables to make testing easier, all of which can be accessed by importing *psychopy.tests.utils*.

Paths

The test utils module includes the following paths:

- *TESTS_PATH* : A path to the root tests folder
- *TESTS_DATA_PATH* : A path to the data folder within the tests folder - here is where all screenshots, example conditions files, etc. for use by the test suite are stored

Compare screenshot

This function allows you to compare the appearance of a *visual.Window* to an image file, raising an *AssertionError* if they aren't sufficiently similar. This takes three arguments:

- *fileName* : A path to the image you want to compare against
- *win* : The window you want to check
- *crit* (optional) : A measure of how lenient to be - this defaults to 5, but we advise increasing it to 20 for anything involving fonts as these can vary between machines

If *filename* points to a file which doesn’t exist, then this function will instead save the window and assume true. Additionally, if the comparison fails, the window will be saved as the same path as *filename*, but with *_local* appended to the name.

Compare pixel color

Sometimes, comparing an entire image may be excessive for what you want to check. For example, if you just want to make sure that a fill color has applied, you could just compare the color of one pixel. This means there doesn’t need to be a *.png* file in the PsychoPy repository, and the test suite also doesn’t have to load a entire image just to compare one color. In these instances, it’s better to use *utils.comparePixelColor*. This function takes three arguments:

- *screen* : The window you want to check
- *color* : The color you expect the pixel to be (ideally, this should be a *colors.Color* object)
- *coord* (optional) : The coordinates of the pixel within the image which you’re wanting to compare (defaults to *(0, 0)*)

Contained within this function is an *assert* call - so if the two colors are not the same, it will raise an *AssertionError* giving you information on both the target color and the pixel color.

Exemplars and tykes

While you’re welcome to lay out your tests however makes the most sense for that test, a useful format in some cases it to define *list`s of “exemplars” and “tykes” - `dict`s of attributes for use in a `for* loop, to save yourself from manually writing the same code over and over, with “exemplars” being very typical use cases which should definitely work as a bare minimum, and “tykes” being edge cases which should work but are not necessarily likely to occur. Here’s an example of this structure:

```
from psychopy import visual, colors  # used to draw stimuli


class TestRect:
    """ A class to test the Rect class """
    @classmethod
    def setup_class(self):
        """ Initialise the rectangle and window objects """
        # Create window
        self.win = visual.Window()
        # Create rect
        self.rect = visual.Rect(self.win)

    def test_color(self):
        """ Test that the color or a rectangle sets correctly """
        # Set the rectangle's fill color
        self.rect.colorSpace = 'rgb'
        self.rect.fillColor = (1, -1, -1)
        # Check that the rgb value of its fill color is consistent with what we set
        assert self.rect._fillColor == colors.Color('red'), f"Was expecting rect._
↪fillColor to have an rgb value of '(1, -1, -1)'," \
                                                            f" but instead it was '{self.rect._
↪fillColor.rgb}'"

    def test_rect_colors(self):
        """Test a range of known exemplar colors as well as colors we know to be␣
```

(continues on next page)

(continued from previous page)

```
↪troublesome AKA tykes"""
    # Define exemplars
    exemplars = [
        { # Red with a blue outline
            'fill': 'red',
            'border': 'blue',
            'colorSpace': 'rgb',
            'targetFill': colors.Color((1, -1, -1), 'rgb'),
            'targetBorder': colors.Color((-1, -1, 1), 'rgb'),
        },
        { # Blue with a red outline
            'fill': 'blue',
            'border': 'red',
            'colorSpace': 'rgb',
            'targetFill': colors.Color((-1, -1, 1), 'rgb'),
            'targetBorder': colors.Color((1, -1, -1), 'rgb'),
        },
    ]
    # Define tykes
    tykes = [
        { # Transparent fill with a red border when color space is hsv
            'fill': None,
            'border': 'red',
            'colorSpace': 'rgb',
            'targetFill': colors.Color(None, 'rgb'),
            'targetBorder': colors.Color((0, 1, 1), 'hsv'),
        }
    ]
    # Iterate through all exemplars and tykes
    for case in exemplars + tykes:
        # Set colors
        self.rect.colorSpace = case['colorSpace']
        self.rect.fillColor = case['fill']
        self.rect.borderColor = case['border']
        # Check values are the same
        assert self.rect._fillColor == case['targetFill'], f"Was expecting rect._
↪fillColor to be '{case['targetFill']}', but instead it was '{self.rect._fillColor}'"
        assert self.rect._borderColor == case['targetBorder'], f"Was expecting rect._
↪borderColor to be '{case['targetBorder']}', but instead it was '{self.rect._
↪borderColor}'"
```

Cleanup

After opening any windows, initialising objects or opening any part of the app, it's important to do some cleanup afterwards - otherwise these won't close and the test suite will just keep running forever. This just means calling *.Close()* on any *wx.Frame`s, `.close()* on any *visual.Window`s, and using `del* to get rid of any objects.

For functions, you can just do this at the end of the function, before it terminates. For classes, this needs to be done in a method called *teardown_class*; as *pytest* will call this method when the tests have completed. This method also needs to have a decorator marking it as a *classfunction*, like so:

```
from psychopy import visual
```

(continues on next page)

(continued from previous page)

```
class ExampleTest:
    def __init__(self):
        # Start an app
        wx.App()
        # Create a frame
        self.frame = wx.Frame()
        # Create a window
        self.win = visual.Window()
        # Create an object
        self.rect = visual.Rect(win)

    @classmethod
    def teardown_class(self):
        # Close the frame
        self.frame.Close()
        # Close the window
        self.win.close()
        # Delete the object
        del self.rect
```

Exercise

Add a *teardown_class* method to your TestRect class.

CodeCov

CodeCov is a handy tool which runs the full test suite and keeps track of which lines of code are executed - giving each file in the PsychoPy repo a percentage score for "coverage". If more lines of code in that file are executed when the test suite runs, then it has a higher coverage score. You can view the full coverage report for the repo [here](https://app.codecov.io/gh/psychopy/psychopy/).

Some areas of the code are more important than others, so it's important not to make decisions purely based on what most increases coverage, but coverage can act as a good indicator for what areas the test suite is lacking in. If you want to make a test but aren't sure what to do, finding a file or folder with a poor coverage score is a great place to start!

Solutions

Testing if a created rectangle has 4 vertices:

```
def test_rect(self):
    """ Test that a rect object has 4 vertices """
    assert len(self.rect.vertices) == 4, f"Was expecting 4 vertices in a Rect object,␣
↪got {len(self.rect.vertices)}"
```

Adding a *teardown_class* method to your TestRect class:

```
class TestRect:
    """ A class to test the Rect class """
    @classmethod
    def setup_class(self):
        """ Initialise the rectangle and window objects """
```

(continues on next page)

(continued from previous page)

```
        # Create window
        self.win = visual.Window()
        # Create rect
        self.rect = visual.Rect(self.win)

    def test_color(self):
        """ Test that the color or a rectangle sets correctly """
        # Set the rectangle's fill color
        self.rect.colorSpace = 'rgb'
        self.rect.fillColor = (1, -1, -1)
        # Check that the rgb value of its fill color is consistent with what we set
        assert self.rect._fillColor == colors.Color('red'), f"Was expecting rect._
↪fillColor to have an rgb value of '(1, -1, -1)'," \
                                              f" but instead it was '{self.rect._
↪fillColor.rgb}'"

    def test_rect(self):
        """ Test that a rect object has 4 vertices """
        assert len(self.rect.vertices) == 4, f"Was expecting 4 vertices in a Rect object,
↪ got {len(self.rect.vertices)}"

    def test_rect_colors(self):
        """Test a range of known exemplar colors as well as colors we know to be␣
↪troublesome AKA tykes"""
        # Define exemplars
        exemplars = [
            { # Red with a blue outline
                'fill': 'red',
                'border': 'blue',
                'colorSpace': 'rgb',
                'targetFill': colors.Color((1, -1, -1), 'rgb'),
                'targetBorder': colors.Color((-1, -1, 1), 'rgb'),
            },
            { # Blue with a red outline
                'fill': 'blue',
                'border': 'red',
                'colorSpace': 'rgb',
                'targetFill': colors.Color((-1, -1, 1), 'rgb'),
                'targetBorder': colors.Color((1, -1, -1), 'rgb'),
            },
        ]
        # Define tykes
        tykes = [
            { # Transparent fill with a red border when color space is hsv
                'fill': None,
                'border': 'red',
                'colorSpace': 'rgb',
                'targetFill': colors.Color(None, 'rgb'),
                'targetBorder': colors.Color((0, 1, 1), 'hsv'),
            }
        ]
        # Iterate through all exemplars and tykes
```

(continues on next page)

(continued from previous page)

```
        for case in exemplars + tykes:
            # Set colors
            self.rect.colorSpace = case['colorSpace']
            self.rect.fillColor = case['fill']
            self.rect.borderColor = case['border']
            # Check values are the same
            assert self.rect._fillColor == case['targetFill'], f"Was expecting rect._
↪fillColor to be '{case['targetFill']}', but instead it was '{self.rect._fillColor}'"
            assert self.rect._borderColor == case['targetBorder'], f"Was expecting rect._
↪borderColor to be '{case['targetBorder']}', but instead it was '{self.rect._
↪borderColor}'"

    @classmethod
    def teardown_class(self):
        """clean-up any objects, wxframes or windows opened by the test"""
        # Close the window
        self.win.close()
        # Delete the object
        del self.rect
```

Adding a new Menu Item

Adding a new menu-item to the Builder (or Coder) is relatively straightforward, but there are several files that need to be changed in specific ways.

1. makeMenus()

The code that constructs the menus for the Builder is within a method named *makeMenus()*, within class builder.BuilderFrame(). Decide which submenu your new command fits under, and look for that section (e.g., File, Edit, View, and so on). For example, to add an item for making the Routine panel items larger, I added two lines within the View menu, by editing the *makeMenus()* method of class *BuilderFrame* within *psychopy/app/builder/builder.py* (similar for Coder):

```
self.viewMenu.Append(self.IDs.tbIncrRoutineSize, _("&Routine Larger\t%s") %self.app.keys[
↪'largerRoutine'], _("Larger routine items"))
wx.EVT_MENU(self, self.IDs.tbIncrRoutineSize, self.routinePanel.increaseSize)
```

Note the use of the translation function, _(), for translating text that will be displayed to users (menu listing, hint).

2. wxIDs.py

A new item needs to have a (numeric) ID so that *wx* can keep track of it. Here, the number is *self.IDs.tbIncrRoutineSize*, which I had to define within the file *psychopy/app/wxIDs.py*:

```
tbIncrRoutineSize=180
```

It’s possible that, instead of hard-coding it like this, it’s better to make a call to *wx.NewIdRef()* – wx will take care of avoiding duplicate IDs, presumably.

3. Key-binding prefs

I also defined a key to use to as a keyboard short-cut for activating the new menu item:

```
self.app.keys['largerRoutine']
```

The actual key is defined in a preference file. Because psychopy is multi-platform, you need to add info to four different .spec files, all of them being within the *psychopy/preferences/* directory, for four operating systems (Darwin, FreeBSD, Linux, Windows). For *Darwin.spec* (meaning macOS), I added two lines. The first line is not merely a comment: it is also automatically used as a tooltip (in the preferences dialog, under key-bindings), and the second being the actual short-cut key to use:

```
# increase display size of Routines
largerRoutine = string(default='Ctrl++') # on Mac Book Pro this is good
```

This means that the user has to hold down the *Ctrl* key and then press the + key. Note that on Macs, 'Ctrl' in the spec is automatically converted into 'Cmd' for the actual key to use; in the .spec, you should always specify things in terms of 'Ctrl' (and not 'Cmd'). The default value is the key-binding to use unless the user defines another one in her or his preferences (which then overrides the default). Try to pick a sensible key for each operating system, and update all four .spec files.

4. Your new method

The second line within *makeMenus()* adds the key-binding definition into wx's internal space, so that when the key is pressed, *wx* knows what to do. In the example, it will call the method *self.routinePanel.increaseSize*, which I had to define to do the desired behavior when the method is called (in this case, increment an internal variable and redraw the routine panel at the new larger size).

5. Documentation

To let people know that your new feature exists, add a note about your new feature in the CHANGELOG.txt, and appropriate documentation in .rst files.

Adding a new Builder Component

Builder Components are auto-detected and displayed to the experimenter as icons (in the right-most panel of the Builder interface panel). This makes it straightforward to add new ones.

All you need to do is create a list of parameters that the Component needs to know about (that will automatically appear in the Component's dialog) and a few pieces of code specifying what code should be called at different points in the script (e.g. beginning of the Routine, every frame, end of the study etc. . .). Many of these will come simply from subclassing the _base or _visual Components.

To get started, addFeature for the development of this component. (If this doesn't mean anything to you then see usingRepos)

You'll mainly be working in the directory . . . */psychopy/experiment/components/*. Take a look at several existing Components (such as *image.py*), and key files including *_base.py* and *_visual.py*.

There are three main steps, the first being by far the most involved.

1. Create the file defining the component: newcomp.py

It's most straightforward to model a new Component on one of the existing ones. Be prepared to specify what your Component needs to do at several different points in time: the first trial, every frame, at the end of each routine, and at the end of the experiment. In addition, you may need to sacrifice some complexity in order to keep things streamlined enough for a Builder (see e.g., *ratingscale.py*).

Your new Component class (in your file *newcomp.py*) should inherit from *BaseComponent* (in *_base.py*), *VisualComponent* (in *_visual.py*), or *KeyboardComponent* (in *keyboard.py*). You may need to rewrite some or all some of these methods, to override default behavior:

```
class NewcompComponent(BaseComponent): # or (VisualComponent)
    def __init__(...):
        super(NewcompComponent, self).__init__(...)
            ...
    def writeInitCode(self, buff):
    def writeRoutineStartCode(self, buff):
    def writeFrameCode(self, buff):
    def writeRoutineEndCode(self, buff):
```

Calling *super()* will create the basic default set of *params* that almost every component will need: *name*, *startVal*, *startType*, etc. Some of these fields may need to be overridden (e.g., *durationEstim* in *sound.py*). Inheriting from *VisualComponent* (which in turn inherits from *BaseComponent*) adds default visual params, plus arranges for Builder scripts to import *psychopy.visual*. If your component will need other libs, call *self.exp.requirePsychopyLib(['neededLib'])* (see e.g., *parallelPort.py*).

At the top of a component file is a dict named *_localized*. It contains mappings that allow a strict separation of internal string values (= used in logic, never displayed) from values used for display in the Builder interface (= for display only, possibly translated, never used in logic). The *.hint* and *.label* fields of *params['someParam']* should always be set to a localized value, either by using a dict entry such as *_localized['message']*, or via the globally available translation function, *_('message')*. Localized values must **not** be used elsewhere in a component definition.

Very occasionally, you may also need to edit *settings.py*, which writes out the set-up code for the whole experiment (e.g., to define the window). For example, this was necessary for the ApertureComponent, to pass *allowStencil=True* to the window creation.

Your new Component writes code into a buffer that becomes an executable python file, *xxx_lastrun.py* (where *xxx* is whatever the experimenter specifies when saving from the Builder, *xxx.psyexp*). You will do a bunch of this kind of call in your *newcomp.py* file:

```
buff.writeIndented(your_python_syntax_string_here)
```

You have to manage the indentation level of the output code, see *experiment.IndentingBuffer()*.

xxx_lastrun.py is the file that gets built when you run *xxx.psyexp* from the Builder. So you will want to look at *xxx_lastrun.py* frequently when developing your component.

Name-space

There are several internal variables (i.e. names of Python objects) that have a specific, hardcoded meaning within *xxx_lastrun.py*. You can expect the following to be there, and they should only be used in the original way (or something will break for the end-user, likely in a mysterious way):

```
win    # the window
t      # time within the trial loop, referenced to `trialClock`
x, y  # mouse coordinates, but only if the experimenter uses a mouse component
```

Handling of variable names is under active development, so this list may well be out of date. (If so, you might consider updating it or posting a note to the Discourse developer forum.)

Preliminary testing suggests that there are 600-ish names from numpy or numpy.random, plus the following:

```
['KeyResponse', '__builtins__', '__doc__', '__file__', '__name__', '__package__',
↪'buttons', 'core', 'data', 'dlg', 'event', 'expInfo', 'expName', 'filename', 'gui',
↪'logFile', 'os', 'psychopy', 'sound', 't', 'visual', 'win', 'x', 'y']
```

Yet other names get derived from user-entered names, like *trials* –> *thisTrial*.

Params

self.params is a key construct that you build up in *__init__*. You need name, startTime, duration, and several other params to be defined or you get errors. *'name'* should be of type *'code'*.

The *Param()* class is defined in *psychopy.app.builder.experiment.Param()*. A very useful thing that Params know is how to create a string suitable for writing into the .py script. In particular, the *__str__* representation of a Param will format its value (*.val*) based on its type (*.valType*) appropriately. This means that you don't need to check or handle whether the user entered a plain string, a string with a code trigger character (*$*), or the field was of type *code* in the first place. If you simply request the *str()* representation of the param, it is formatted correctly.

To indicate that a param (eg, *thisParam*) should be considered as an advanced feature, set its category to advanced: *self.params['thisParam'].categ = 'Advanced'*. Then the GUI shown to the experimenter will automatically place it on the 'Advanced' tab. Other categories work similarly (*Custom*, etc).

During development, it can sometimes be helpful to save the params into the *xxx_lastrun.py* file as comments, so you can see what is happening:

```
def writeInitCode(self,buff):
    # for debugging during Component development:
    buff.writeIndented("# self.params for aperture:\n")
    for p in self.params:
        try: buff.writeIndented("# %s: %s <type %s>\n" % (p, self.params[p].val, self.
↪params[p].valType))
        except: pass
```

A lot more detail can be inferred from existing components.

Making things loop-compatible looks interesting – see *keyboard.py* for an example, especially code for saving data at the end.

Notes & gotchas

syntax errors in new_comp.py:

The app will fail to start if there are syntax error in any of the components that are auto-detected. Just correct them and start the app again.

param[].val:

If you have a boolean variable (e.g., *my_flag*) as one of your params, note that *self.param["my_flag"]* is always True (the param exists –> True). So in a boolean context you almost always want the *.val* part, e.g., *if self.param["my_flag"].val:*.

However, you do not always want *.val*. Specifically, in a string/unicode context (= to trigger the self-formatting features of Param()s), you almost always want *"%s" % self.param['my_flag']*, without *.val*. Note that it's better to do this via *"%s"* than *str()* because *str(self.param["my_flag"])* coerces things to type str (squashing unicode) whereas *%s* works for both str and unicode.

Travis testing

Before submitting a pull request with the new component, you should regenerate the *componsTemplate.txt* file. This is a text file that lists the attributes of all of the user interface settings and options in the various components. It is used during the Travis automated testing process when a pull request is submitted to GitHub, allowing the detection of errors that may have been caused in refactoring. Your new component needs to have entries added to this file if the Travis testing is going to pass successfully.

To re-generate the file, cd to this directory *. . . /psychopy/tests/test_app/test_builder/* and run:

```
`python genComponsTemplate.py --out`
```

This will over-write the existing file so you might want to make a copy in case the process fails. *Compatibility issues:* As at May 2018, that script is not yet Python 3 compatible, and on a Mac you might need to use *pythonw*.

2. Icon: newcomp.png

Using your favorite image software, make an icon for your Component with a descriptive name, e.g., *newcomp.png*. Dimensions = 48 × 48. Put it in the components directory.

In *newcomp.py*, have a line near the top:

```
iconFile = path.join(thisFolder, 'newcomp.png')
```

3. Documentation: newcomp.rst

Just make a descriptively-named text file that ends in *.rst* (“restructured text”), and put it in *psychopy/docs/source/builder/components/* . It will get auto-formatted and end up at *https://www.psychopy.org/builder/components/newcomp.html*

CHAPTER

EIGHTEEN

EXPERIMENT FILE FORMAT (.PSYEXP)

The file format used to save experiments constructed in builder was created especially for the purpose, but is an open format, using a basic xml form, that may be of use to other similar software. Indeed the builder itself could be used to generate experiments on different backends (such as Vision Egg, PsychToolbox or PyEPL). The xml format of the file makes it extremely platform independent, as well as moderately(?!) easy to read by humans. There was a further suggestion to generate an XSD (or similar) schema against which psyexp files could be validated. That is a low priority but welcome addition if you wanted to work on it(!) There is a basic XSD (XML Schema Definition) available in *psychopy/app/builder/experiment.xsd*.

The simplest way to understand the file format is probably simply to create an experiment, save it and open the file in an xml-aware editor/viewer (e.g. change the file extension from .psyexp to .xml and then open it in Firefox). An example (from the stroop demo) is shown below.

The file format maps fairly obviously onto the *structure of experiments* constructed with the *Builder* interface. There are general *Settings* for the experiment, then there is a list of *Routines* and a *Flow* that describes how these are combined.

As with any xml file the format contains object *nodes* which can have direct properties and also child nodes. For instance the outermost node of the .psyexp file is the experiment node, with properties that specify the version of that was used to save the file most recently and the encoding of text within the file (ascii, unicode etc.), and with child nodes *Settings*, *Routines* and *Flow*.

18.1 Parameters

Many of the nodes described within this xml description of the experiment contain Param entries, representing different parameters of that Component. Nearly all parameter nodes have a *name* property and a *val* property. The parameter node with the name “advancedParams” does not have them. Most also have a *valType* property, which can take values ‘bool’, ‘code’, ‘extendedCode’, ‘num’, ‘str’ and an *updates* property that specifies whether this parameter is changing during the experiment and, if so, whether it changes ‘every frame’ (of the monitor) or ‘every repeat’ (of the Routine).

18.2 Settings

The Settings node contains a number of parameters that, in , would normally be set in the *Experiment settings* dialog, such as the monitor to be used. This node contains a number of *Parameters* that map onto the entries in that dialog.

18.3 Routines

This node provides a sequence of xml child nodes, each of which describes a *Routine*. Each Routine contains a number of children, each specifying a *Component*, such as a stimulus or response collecting device. In the *Builder* view, the *Routines* obviously show up as different tabs in the main window and the *Components* show up as tracks within that tab.

18.4 Components

Each *Component* is represented in the .psyexp file as a set of parameters, corresponding to the entries in the appropriate component dialog box, that completely describe how and when the stimulus should be presented or how and when the input device should be read from. Different *Components* have slightly different nodes in the xml representation which give rise to different sets of parameters. For instance the *TextComponent* nodes has parameters such as *colour* and *font*, whereas the *KeyboardComponent* node has parameters such as *forceEndTrial* and *correctIf*.

18.5 Flow

The Flow node is rather more simple. Its children simply specify objects that occur in a particular order in time. A Routine described in this flow must exist in the list of Routines, since this is where it is fully described. One Routine can occur once, more than once or not at all in the Flow. The other children that can occur in a Flow are LoopInitiators and LoopTerminators which specify the start and endpoints of a loop. All loops must have exactly one initiator and one terminator.

18.6 Names

For the experiment to generate valid code the name parameters of all objects (Components, Loops and Routines) must be unique and contain no spaces. That is, an experiment can not have two different Routines called 'trial', nor even a Routine called 'trial' and a Loop called 'trial'.

The Parameter names belonging to each Component (or the Settings node) must be unique within that Component, but can be identical to parameters of other Components or can match the Component name themselves. A TextComponent should not, for example, have multiple 'pos' parameters, but other Components generally will, and a Routine called 'pos' would also be also permissible.

```
<PsychoPy2experiment version="1.50.04" encoding="utf-8">
  <Settings>
    <Param name="Monitor" val="testMonitor" valType="str" updates="None"/>
    <Param name="Window size (pixels)" val="[1024, 768]" valType="code" updates="None"/>
    <Param name="Full-screen window" val="True" valType="bool" updates="None"/>
    <Param name="Save log file" val="True" valType="bool" updates="None"/>
    <Param name="Experiment info" val="{'participant':'s_001', 'session':001}" valType=
↪"code" updates="None"/>
    <Param name="Show info dlg" val="True" valType="bool" updates="None"/>
    <Param name="logging level" val="warning" valType="code" updates="None"/>
    <Param name="Units" val="norm" valType="str" updates="None"/>
    <Param name="Screen" val="1" valType="num" updates="None"/>
  </Settings>
  <Routines>
    <Routine name="trial">
      <TextComponent name="word">
        <Param name="name" val="word" valType="code" updates="constant"/>
        <Param name="text" val="thisTrial.text" valType="code" updates="set every repeat
↪"/>
        <Param name="colour" val="thisTrial.rgb" valType="code" updates="set every repeat
↪"/>
        <Param name="ori" val="0" valType="code" updates="constant"/>
        <Param name="pos" val="[0, 0]" valType="code" updates="constant"/>
        <Param name="times" val="[0.5,2.0]" valType="code" updates="constant"/>
        <Param name="letterHeight" val="0.2" valType="code" updates="constant"/>
```

(continues on next page)

(continued from previous page)

```
        <Param name="colourSpace" val="rgb" valType="code" updates="constant"/>
        <Param name="units" val="window units" valType="str" updates="None"/>
        <Param name="font" val="Arial" valType="str" updates="constant"/>
      </TextComponent>
      <KeyboardComponent name="resp">
        <Param name="storeCorrect" val="True" valType="bool" updates="constant"/>
        <Param name="name" val="resp" valType="code" updates="None"/>
        <Param name="forceEndTrial" val="True" valType="bool" updates="constant"/>
        <Param name="times" val="[0.5,2.0]" valType="code" updates="constant"/>
        <Param name="allowedKeys" val="['1','2','3']" valType="code" updates="constant"/>
        <Param name="storeResponseTime" val="True" valType="bool" updates="constant"/>
        <Param name="correctIf" val="resp.keys==str(thisTrial.corrAns)" valType="code"␣
↪updates="constant"/>
        <Param name="store" val="last key" valType="str" updates="constant"/>
      </KeyboardComponent>
    </Routine>
    <Routine name="instruct">
      <TextComponent name="instrText">
        <Param name="name" val="instrText" valType="code" updates="constant"/>
        <Param name="text" val="&quot;Please press;&#10;1 for red ink,&#10;2 for green␣
↪ink&#10;3 for blue ink&#10;(Esc will quit)&#10;&#10;Any key to continue&quot;" valType=
↪"code" updates="constant"/>
        <Param name="colour" val="[1, 1, 1]" valType="code" updates="constant"/>
        <Param name="ori" val="0" valType="code" updates="constant"/>
        <Param name="pos" val="[0, 0]" valType="code" updates="constant"/>
        <Param name="times" val="[0, 10000]" valType="code" updates="constant"/>
        <Param name="letterHeight" val="0.1" valType="code" updates="constant"/>
        <Param name="colourSpace" val="rgb" valType="code" updates="constant"/>
        <Param name="units" val="window units" valType="str" updates="None"/>
        <Param name="font" val="Arial" valType="str" updates="constant"/>
      </TextComponent>
      <KeyboardComponent name="ready">
        <Param name="storeCorrect" val="False" valType="bool" updates="constant"/>
        <Param name="name" val="ready" valType="code" updates="None"/>
        <Param name="forceEndTrial" val="True" valType="bool" updates="constant"/>
        <Param name="times" val="[0, 10000]" valType="code" updates="constant"/>
        <Param name="allowedKeys" val="" valType="code" updates="constant"/>
        <Param name="storeResponseTime" val="False" valType="bool" updates="constant"/>
        <Param name="correctIf" val="resp.keys==str(thisTrial.corrAns)" valType="code"␣
↪updates="constant"/>
        <Param name="store" val="last key" valType="str" updates="constant"/>
      </KeyboardComponent>
    </Routine>
    <Routine name="thanks">
      <TextComponent name="thanksText">
        <Param name="name" val="thanksText" valType="code" updates="constant"/>
        <Param name="text" val="&quot;Thanks!&quot;" valType="code" updates="constant"/>
        <Param name="colour" val="[1, 1, 1]" valType="code" updates="constant"/>
        <Param name="ori" val="0" valType="code" updates="constant"/>
        <Param name="pos" val="[0, 0]" valType="code" updates="constant"/>
        <Param name="times" val="[1.0, 2.0]" valType="code" updates="constant"/>
        <Param name="letterHeight" val="0.2" valType="code" updates="constant"/>
```

(continues on next page)

(continued from previous page)

```
        <Param name="colourSpace" val="rgb" valType="code" updates="constant"/>
        <Param name="units" val="window units" valType="str" updates="None"/>
        <Param name="font" val="arial" valType="str" updates="constant"/>
      </TextComponent>
    </Routine>
  </Routines>
  <Flow>
    <Routine name="instruct"/>
    <LoopInitiator loopType="TrialHandler" name="trials">
      <Param name="endPoints" val="[0, 1]" valType="num" updates="None"/>
      <Param name="name" val="trials" valType="code" updates="None"/>
      <Param name="loopType" val="random" valType="str" updates="None"/>
      <Param name="nReps" val="5" valType="num" updates="None"/>
      <Param name="trialList" val="[{'text': 'red', 'rgb': [1, -1, -1], 'congruent': 1,
↪'corrAns': 1}, {'text': 'red', 'rgb': [-1, 1, -1], 'congruent': 0, 'corrAns': 1}, {
↪'text': 'green', 'rgb': [-1, 1, -1], 'congruent': 1, 'corrAns': 2}, {'text': 'green',
↪'rgb': [-1, -1, 1], 'congruent': 0, 'corrAns': 2}, {'text': 'blue', 'rgb': [-1, -1, 1],
↪ 'congruent': 1, 'corrAns': 3}, {'text': 'blue', 'rgb': [1, -1, -1], 'congruent': 0,
↪'corrAns': 3}]" valType="str" updates="None"/>
      <Param name="trialListFile" val="/Users/jwp...troop/trialTypes.csv" valType="str"␣
↪updates="None"/>
    </LoopInitiator>
    <Routine name="trial"/>
    <LoopTerminator name="trials"/>
    <Routine name="thanks"/>
  </Flow>
</PsychoPy2experiment>
```

CHAPTER

NINETEEN

INDICES

- *Glossary*
- genindex

PYTHON MODULE INDEX