Peng3d Documentation

Release 1.12.0

notna

Sep 12, 2022

Contents

Contents:

CHAPTER 1

Guides

This part of the documentation contains guides related to various parts of peng3d.

Contents:

1.1 Installation

There are several different ways to install `peng3d`. The most common ways are listed below.

1.1.1 Using `pip`

This is by far the simplest way to install `peng3d`. Simply run the following command:

```
$ pip install peng3d
```

You may also wish to add `peng3d` to your `requirements.txt` or similar file.

See also:

See the documentation of the Requirements File Format for more details regarding dependency specification.

1.1.2 From source

If you wish to install a development version that is not available on PyPI, you can also install `peng3d` directly from its source code.

First, you'll have to download the code somewhere. This can be done in any way you like, but here is how it can be done using `git`:

```
$ git clone https://github.com/not-na/peng3d.git
$ cd peng3d
```

After having downloaded the source code, you can now install it using the `setup.py` file:

```
$ python setup.py install
```

If there weren't any errors, `peng3d` should now be installed.

1.1.3 Where to go next

After having installed `peng3d`, you may wish to look at the *Quickstart* Guide for a simple and fast introduction to `peng3d` or the *Designing a basic 3D Application with peng3d* Guide for a more in-depth guide.

There are also some examples available in the `examples/` folder on the main repository.

1.2 Quickstart

In this guide, we'll learn how to create a simple GUI using `peng3d`.

See also:

For a more complex example, see *Designing a basic 3D Application with peng3d*.

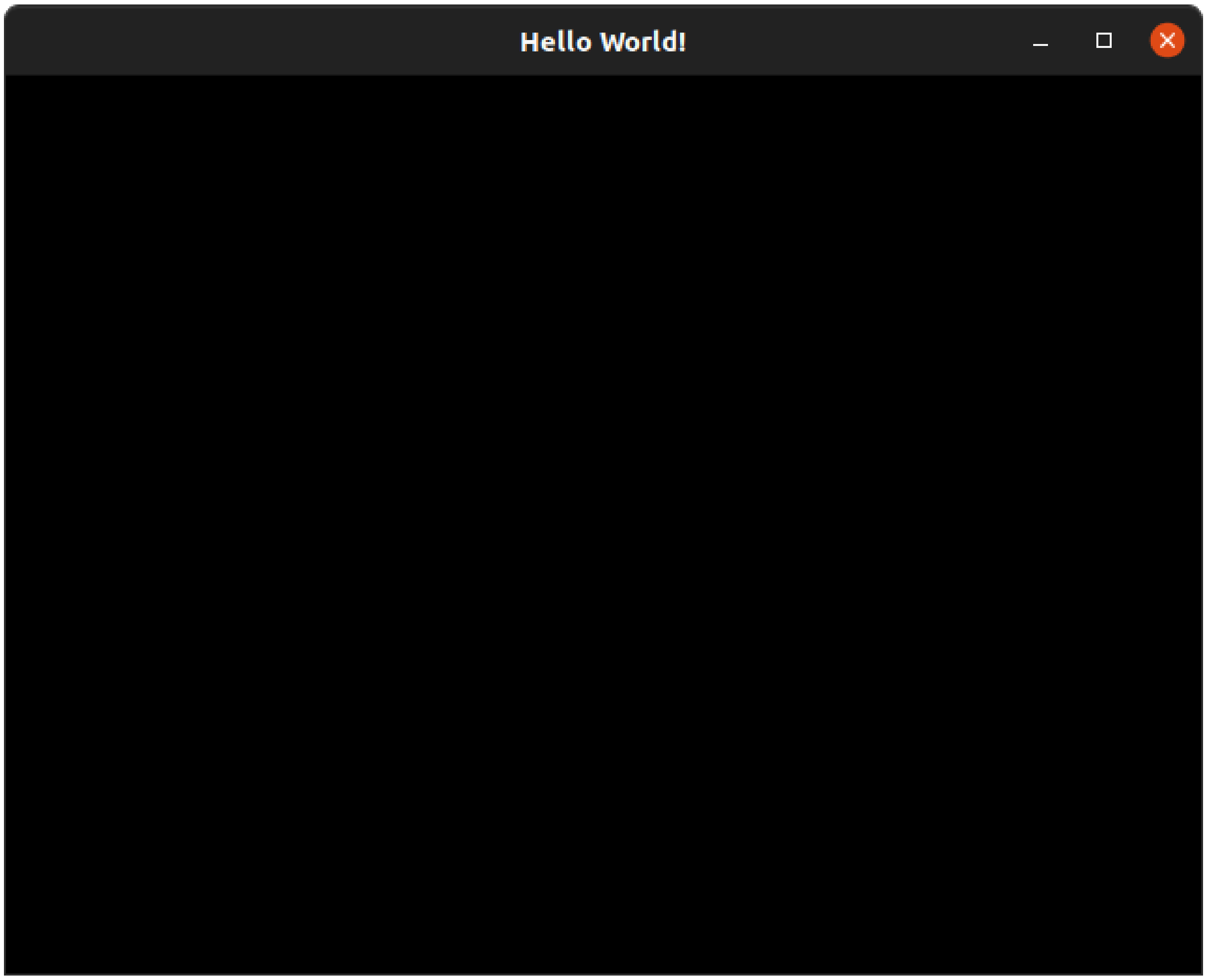
1.2.1 Basic Structure

In this guide, we will be writing our app using only a single Python file for simplicity. For more complex projects, it is recommended to split your application by menus or even submenus into different files.

First, here's the minimum example required to show anything with `peng3d`:

```
import peng3d

peng = peng3d.Peng()

peng.createWindow(caption="Hello World!", resizable=True)

main_menu = peng3d.Menu("main", peng.window)
peng.window.addMenu(main_menu)

peng.window.changeMenu("main")
peng.run()
```

If you run the app now, you should see a black window with a title of `Hello World!`:



While most of the lines should be fairly self-explanatory, let's go through them one by one.

First, we start by importing `peng3d` and creating an instance of the `peng3d.Peng()` class:

```
import peng3d

peng = peng3d.Peng()
```

There should only be one instance of this class per app, shared between all components. It manages the event system and some other globally shared resources.

Next, we create our window with the desired caption:

```
peng.createWindow(caption="Hello World!", resizable=True)
```

Since we want to keep this example very simple, we only pass a caption and activate resizing. All arguments to `Peng.createWindow()` are optional and should be passed as keyword arguments. Any arguments not recognized by `peng3d` are passed through to the underlying `PengWindow` class, which will in turn pass through unrecognized arguments to `pyglet`.

See also:

See the `pyglet.window` module docs for a list of all arguments.

Now that we have created our window, we'll create our first menu and register it:

```
main_menu = peng3d.Menu("main", peng.window)
peng.window.addMenu(main_menu)
```

The basic `Menu` class is designed for layer-based rendering. We will later change this, since we want to create a GUI with widgets.

Also, we always need to register menus, so it is a good practice to always register a menu right after creating it.

Lastly, we'll switch to our main menu and start the application:

```
peng.window.changeMenu("main")
peng.run()
```

The call to `changeMenu()` can be used to switch between different menus, here we use it to define which menu our app starts with. Note that we pass in the name of our menu, not the menu object itself.

The final call to `peng.run()` starts the internal event loop of pyglet and blocks until the application exits, usually by clicking the X button.

Note: The code described in this subsection can also be found in `examples/quickstart/quickstart_basic.py` here.

1.2.2 Creating our first widget

Now that we have a basic skeleton running, let's add some actual functionality. We'll modify the code from the previous subsection bit by bit.

First, lets switch to a more advanced `GUIMenu` instead of the simple `Menu` we used earlier.

Note: For most widget-oriented apps, this is what you'll use, although custom 3D canvases usually use a plain `Menu` with a `GUILayer` for overlayed widgets.

To do this, we'll first change the class name:

```
main_menu = peng3d.GUIMenu("main", peng.window)
```

Then, let's set the background to a more appealing color. For now, we'll use a light grey, although many more variants are possible. You could even use an image or a custom callback as a background. To set the background, simply call `GUIMenu.setBackground()` with the color you want:

```
main_menu.setBackground([240, 240, 240])
```

By setting the background in the menu instead of the submenu, all submenus of this menu will automatically inherit the background unless they overwrite it. This makes it easier to e.g. swap themes.

If you try to run the app now, you'll notice that it won't start. This is because `GUIMenu` menus require an active submenu at all times that they are active. So, let's add a submenu:

```
main_main = peng3d.SubMenu("main_sub", main_menu, peng.window, peng)
```

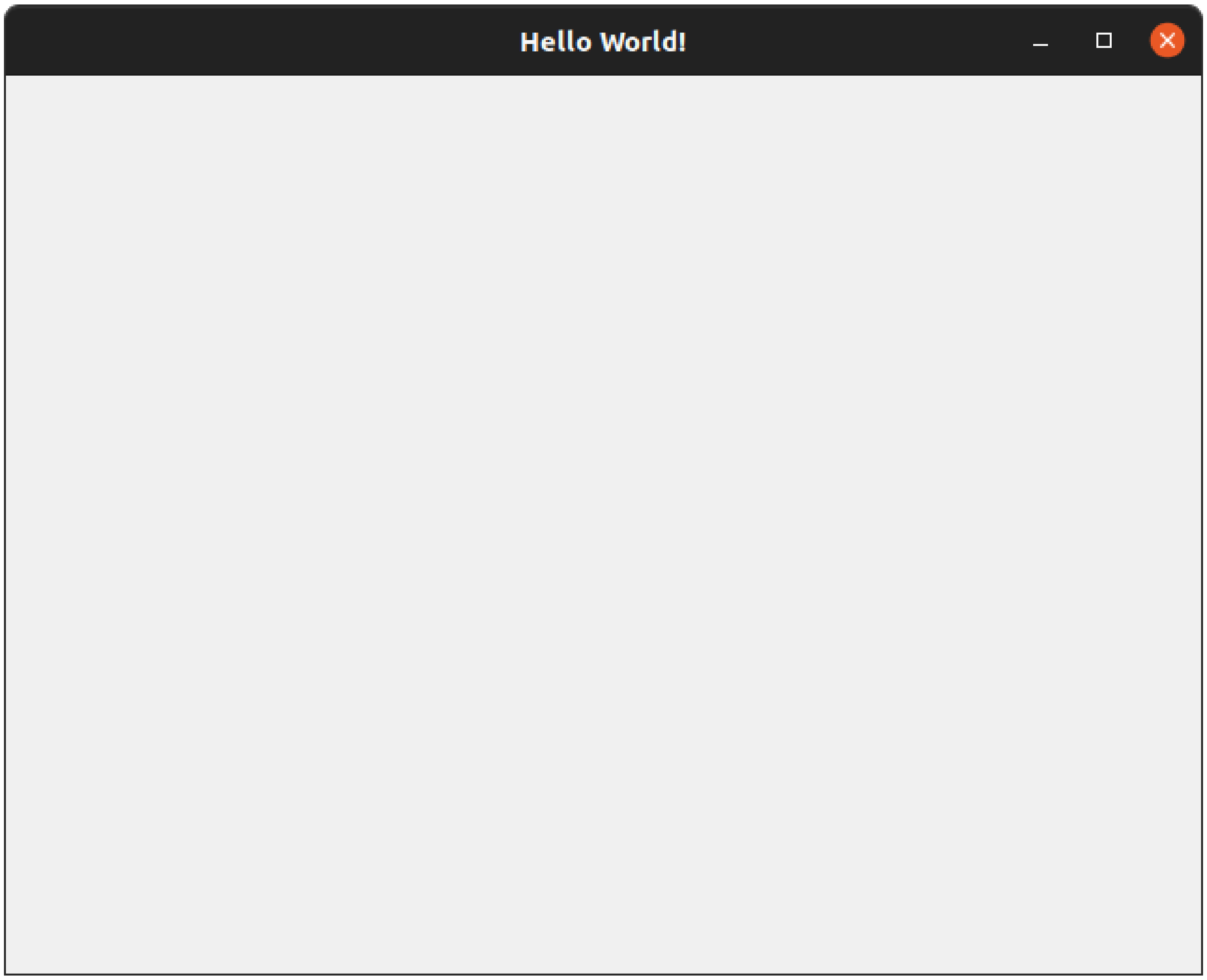
While it may not matter much in this simple app, we have chosen a name for this submenu that is different from the main menu. Any string can be used as a name, so feel free to create your own naming convention.

Note: While submenus of different menus could have identical names, this is strongly discouraged, as it can lead to confusion in larger projects. Ideally, each named object should have a unique name.

We'll also have to tell the main menu to use this submenu, just before it is activated itself:

```
main_menu.changeSubMenu("main_sub")
```

If you run the app now, you should see a grey window instead of a black window:



Now, this is a bit better than just a black window, but not by much. Let's go a bit further and add a single button that prints whenever it is clicked.

To do this, we'll have to first create the button and register it:

```
button = peng3d.Button(
    "btn",
    main_main,
    pos=[100, 100],
    size=[200, 100],
    label="Press me!",
    borderstyle="oldshadow",
```

(continues on next page)

(continued from previous page)

```
)
main_main.addWidget(button)
```

The Button class takes a lot of arguments, so let's go over them.

In the first line, we pass the name of the button. Here, the same caveats apply as with submenu and menu names. We also pass the submenu this widget belongs to, from which the window and Peng singleton references are gathered.

In the next two lines, we pass the position and size of the widget.

Note: Positions in peng3d widgets are always from the bottom-left corner of the screen. Both positions and sizes are in pixels.

Next, we pass the label. For now, we give it a static label, though peng3d also supports easy translation capabilities.

Lastly, we pass what style of border to use. There are several border styles available, further information is available in the documentation of the *Button* class.

See also:

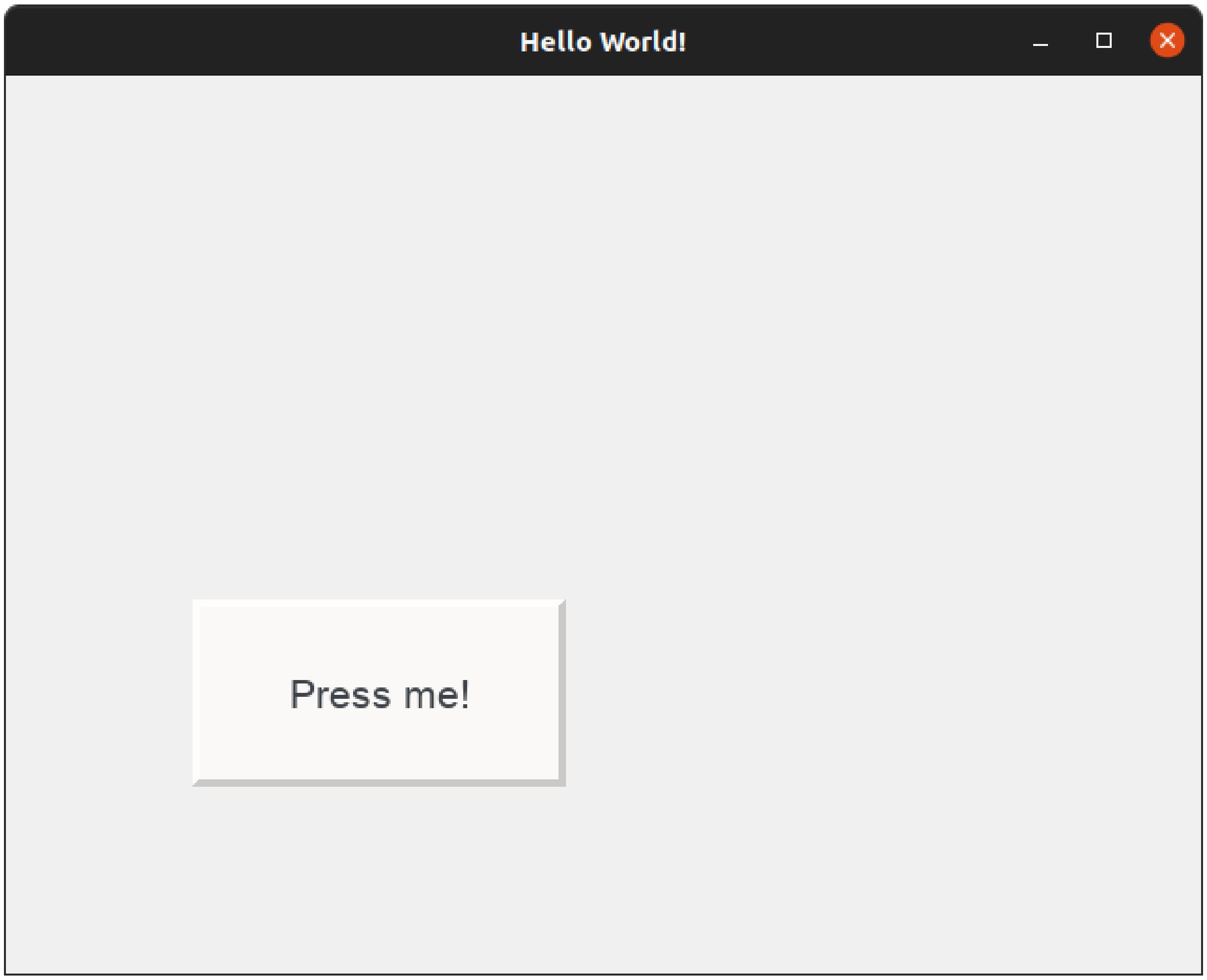
There are many more optional arguments that the *Button* class can take. See the API documentation for details.

Now, we have a button. But it does not do anything yet, so let's add an action that prints something whenever it is called:

```
button.addAction("click", print, "Clicked!")
```

The addAction() method is quite flexible. It takes the name of the action as the first parameter, a function to call as the second parameter and passes all other arguments to each call of the function. So while we could write a one-line function to print out our message, we can just pass the argument to print. Obviously, you'll still have to write proper functions or methods for more complicated handlers.

Now, let's take a look at our current app:



If you run the app yourself, try clicking on the button and watching the console output. You should see `Clicked!` every time you release the button.

If you want, try playing around with the parameters to the `Button` class and see how they effect the look or behaviour of the app.

Once you are done, move on to the next subsection, where we learn how to use and switch between multiple menus.

Note: The code described in this subsection can also be found in `examples/quickstart/quickstart_widget.py` here.

1.2.3 Switching between menus

Todo: Write this subsection

1.2.4 Dynamically adjusting our layout to the window size

Todo: Write this subsection

1.2.5 Further reading

There are other, more advanced guides available. For example, take a look at *Designing a basic 3D Application with peng3d*.

See also:

See the `examples/` folder on the main repository for more examples of various `peng3d` features.

1.3 Designing a basic 3D Application with `peng3d`

Contents:

1.3.1 Introduction

Todo: Write this section

Todo: Write this guide

Todo: Add a full game tutorial (maybe a simple minecraft clone)

CHAPTER 2

`peng3d` - Peng3D main module

This Module represents the root of the peng3d Engine.

Most classes contained in submodules are available under the same name, e.g. you can use `peng3d.Peng()` instead of *`peng3d.peng.Peng()`*. Note that for compatibility reasons, peng3d.window is not available by default and will need to be imported directly.

`*`- importing submodules should be safe as most modules define an `__all__` variable.

2.1 `peng3d.peng` - Main Engine class

class peng3d.peng.**Peng**(*cfg: Optional[peng3d.config.Config] = None*, ***, *style: Optional[Dict[str, Union[float, str, Type[Borderstyle]]]] = None*)

This Class should only be instantiated once per application, if you want to use multiple windows, see *`createWindow()`*.

An Instance of this class represents the whole Engine, with all accompanying state and window/world objects.

Be sure to keep your instance accessible, as it will be needed to create most other classes.

addEventListener(*event: str*, *func: Callable*, *raiseErrors: bool = False*)

Adds a handler to the given event.

A event may have an arbitrary amount of handlers, though assigning too many handlers may slow down event processing.

For the format of `event`, see *`sendEvent()`*.

`func` is the handler which will be executed with two arguments, `event_type` and `data`, as supplied to *`sendEvent()`*.

If `raiseErrors` is True, exceptions caused by the handler will be re-raised. Defaults to `False`.

addPygletListener(*event_type: str*, *handler: Callable*)

Registers an event handler.

The specified callable handler will be called every time an event with the same `event_type` is encountered.

All event arguments are passed as positional arguments.

This method should be used to listen for pyglet events. For new code, it is recommended to use `addEventListener()` instead.

See `handleEvent()` for information about tunneled pyglet events.

For custom events, use `addEventListener()` instead.

`createWindow`(*cls=window.PengWindow*, **args*, ***kwargs*)
Creates a new window using the supplied `cls`.

If `cls` is not given, `peng3d.window.PengWindow()` will be used.

Any other positional or keyword arguments are passed to the class constructor.

Note that this method currently does not support using multiple windows.

Todo: Implement having multiple windows.

`delEventListener`(*event: str*, *func: Callable*)
Removes the given handler from the given event.

If the event does not exist, a `NameError` is thrown.

If the handler has not been registered previously, also a `NameError` will be thrown.

`handleEvent`(*event_type: str*, *args: Tuple*, *window: Optional[pyglet.window.Window] = None*)
Handles a pyglet event.

This method is called by `PengWindow.dispatch_event()` and handles all events.

See `registerEventHandler()` for how to listen to these events.

This method should be used to send pyglet events. For new code, it is recommended to use `sendEvent()` instead. For "tunneling" pyglet events, use event names of the format `pyglet:<event>` and for the data use `{"args":<args as list>,"window":<window object or none>, "src":<event source>,"event_type":<event type>}`

Note that you should send pyglet events only via this method, the above event will be sent automatically.

Do not use this method to send custom events, use `sendEvent()` instead.

`registerEventHandler`(*event_type: str*, *handler: Callable*)
Registers an event handler.

The specified callable handler will be called every time an event with the same `event_type` is encountered.

All event arguments are passed as positional arguments.

This method should be used to listen for pyglet events. For new code, it is recommended to use `addEventListener()` instead.

See `handleEvent()` for information about tunneled pyglet events.

For custom events, use `addEventListener()` instead.

`run`(*evloop: Optional[pyglet.app.EventLoop] = None*)
Runs the application main loop.

This method is blocking and needs to be called from the main thread to avoid OpenGL bugs that can occur.

`evloop` may optionally be a subclass of `pyglet.app.base.EventLoop` to replace the default event loop.

`sendEvent`(*event: str*, *data: Optional[dict] = None*)
Sends an event with attached data.

`event` should be a string of format `<namespace>:<category1>.<subcategory2>.<name>`. There may be an arbitrary amount of subcategories. Also note that this format is not strictly enforced, but rather recommended by convention.

`data` may be any Python Object, but it usually is a dictionary containing relevant parameters. For example, most built-in events use a dictionary containing at least the `peng` key set to an instance of this class.

If there are no handlers for the event, a corresponding message will be printed to the log file. To prevent spam, the maximum amount of ignored messages can be configured via *`events.maxignore`* and defaults to 3.

If the config value *`debug.events.dumpfile`* is a file path, the event type will be added to an internal list and be saved to the given file during program exit.

`sendPygletEvent`(*event_type: str*, *args: Tuple*, *window: Optional[pyglet.window.Window] = None*)
Handles a pyglet event.

This method is called by `PengWindow.dispatch_event()` and handles all events.

See *`registerEventHandler()`* for how to listen to these events.

This method should be used to send pyglet events. For new code, it is recommended to use *`sendEvent()`* instead. For "tunneling" pyglet events, use event names of the format `pyglet:<event>` and for the data use `{"args":<args as list>,"window":<window object or none>, "src":<event source>,"event_type":<event type>}`

Note that you should send pyglet events only via this method, the above event will be sent automatically.

Do not use this method to send custom events, use *`sendEvent()`* instead.

`class` `peng3d.peng.`**`HeadlessPeng`**(*cfg: Union[dict, peng3d.config.Config, None] = None*)
Variant of peng that should work without having pyglet installed.

This class is intended for use in servers as a drop-in replacement for the normal engine class.

Note that this class is a work in progress and should not yet be relied upon.

2.2 `peng3d.window` - Windowing with batteries included

`class` `peng3d.window.`**`PengWindow`**(*peng: peng3d.Peng*, **args*, ***kwargs*)
Main window class for peng3d and subclass of `pyglet.window.Window()`.

This class should not be instantiated directly, use the `Peng.createWindow()` method.

`addMenu`(*menu: peng3d.BasicMenu*) → peng3d.BasicMenu
Adds a menu to the list of menus.

`changeMenu`(*menu: str*) → None
Changes to the given menu.

`menu` must be a valid menu name that is currently known.

`dispatch_event`(*event_type: str*, **args*)
Internal event handling method.

This method extends the behavior inherited from `pyglet.window.Window.dispatch_event()` by calling the various `handleEvent()` methods.

By default, `Peng.handleEvent()`, `handleEvent()` and `Menu.handleEvent()` are called in this order to handle events.

Note that some events may not be handled by all handlers during early startup.

menu

Property for accessing the currently active menu.

Always equals `self.menus[self.activeMenu]`.

This property is read-only.

on_draw()

Clears the screen and draws the currently active menu.

run(*evloop: Optional[pyglet.app.EventLoop] = None*) → None

Runs the application in the current thread.

This method should not be called directly, especially when using multiple windows, use `Peng.run()` instead.

Note that this method is blocking as rendering needs to happen in the main thread. It is thus recommendable to run your game logic in another thread that should be started before calling this method.

`evloop` may optionally be a subclass of `pyglet.app.base.EventLoop` to replace the default event loop.

set2d()

Configures OpenGL to draw in 2D.

Note that wireframe mode is always disabled in 2D-Mode, but can be re-enabled by calling `glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)`.

set3d(*cam*)

Configures OpenGL to draw in 3D.

This method also applies the correct rotation and translation as set in the supplied camera `cam`. It is discouraged to use `glTranslatef()` or `glRotatef()` directly as this may cause visual glitches.

If you need to configure any of the standard parameters, see the docs about *Configuration Options for peng3d*.

The *`graphics.wireframe`* config value can be used to enable a wireframe mode, useful for debugging visual glitches.

set_fps(*fps: Optional[float]*)

Sets the new FPS limit.

This limit will be used until the application closes or this method is called again.

A value of `None` will cause the FPS limit to be disabled.

Note that this is only a limit, which may or may not be fulfilled depending on available resources.

Note: By default, pyglet only redraws the window when an event arrives. To force a certain redraw rate (which still respects system performance), call `pyglet.clock.schedule_interval()` once during initialization with a dummy function and your desired refresh rate in seconds.

Parameters **`fps`** –

Returns

`setup``()`
Sets up the OpenGL state.

This method should be called once after the config has been created and before the main loop is started. You should not need to manually call this method, as it is automatically called by `run()`.

Repeatedly calling this method has no effects.

`setupFog``()`
Sets the fog system up.

The specific options available are documented under *`graphics.fogSettings`*.

`setupLight``()`
Sets the light system up.

The specific options available are documented under *`graphics.lightSettings`*.

Note that this feature is currently not implemented.

`toggle_exclusivity``(`*override=None*`)`
Toggles mouse exclusivity via pyglet's `set_exclusive_mouse()` method.

If `override` is given, it will be used instead.

You may also read the current exclusivity state via `exclusive`.

2.3 `peng3d.layer` - Extensible 2D/3D Layering

`class` `peng3d.layer.`**`Layer`**`(`*menu: peng3d.menu.BasicMenu*, *window: Any = None*, *peng: Any = None*`)`
Base layer class.

A Layer can be used to separate background from foreground or the 3d world from a 2d HUD.

This class by itself does nothing, you will need to subclass it and override the `draw()` method.

`draw``()` → None
Called when this layer needs to be drawn.

Override this method in subclasses to redefine behavior.

`on_menu_enter``(`*old*`)`
Custom fake event handler called by `Menu.on_enter()` for every layer.

Useful for adding and removing event handlers per layer.

`on_menu_exit``(`*new*`)`
Custom fake event handler called by `Menu.on_exit()` for every layer.

Useful for adding and removing event handlers per layer.

`on_redraw``()` → None
Called whenever the Layer should redraw itself.

Note that this method should not be called manually, instead call `redraw()`.

Returns None

postdraw() → None

Called after the `draw()` method is called.

This method can be used to reset OpenGL state to avoid conflicts with other code.

Override this method in subclasses to redefine behavior.

predraw() → None

Called before the `draw()` method is called.

This method is used in the `Layer2D()` and `Layer3D()` subclasses for setting OpenGL state.

Override this method in subclasses to redefine behavior.

redraw() → None

Call this to redraw the layer.

Note that the redraw will not happen immediately, rather on the next frame that this layer is rendered. This massively improves performance.

Returns

class peng3d.layer.**Layer2D**(*menu: peng3d.menu.BasicMenu*, *window: Any = None*, *peng: Any = None*)

2D Variant of `Layer()` and a subclass of the former.

This class makes use of the `predraw()` method to configure OpenGL to draw 2-Dimensionally. This class uses `PengWindow.set2d()` to set the 2D mode.

When overriding the `predraw()` method, make sure to call the superclass.

predraw()

Uses `PengWindow.set2d()` to enable a 2D OpenGL state.

class peng3d.layer.**Layer3D**(*menu: peng3d.menu.BasicMenu*, *window: Any = None*, *peng: Any = None*, *cam: peng3d.camera.Camera = None*)

3D Variant of `Layer()` and a subclass of the former.

This class works the same as `Layer2D()`, only for 3D drawing instead. This class uses `PengWindow.set3d()` to set the 3D mode.

Also, the correct `glTranslatef()` and `glRotatef()` are applied to simplify drawing objects. When writing the `draw()` method of this class, you will only need to use world coordinates, not camera coordinates. This allows for easy building of Games using First-Person-Perspectives.

predraw()

Uses `PengWindow.set3d()` to enable a 3D OpenGL state.

class peng3d.layer.**LayerGroup**(*menu: peng3d.menu.BasicMenu*, *window: Any = None*, *peng: Any = None*, *group: pyglet.graphics.Group = None*)

Layer variant wrapping the supplied pyglet group.

`group` may only be an instance of `pyglet.graphics.Group`, else a `TypeError` will be raised.

Also note that both the `predraw()` and `postdraw()` methods are overwritten by this class.

See also:

For more information about pyglet groups, see the pyglet docs.

postdraw()

Re-sets the previous state.

predraw()

Sets the group state.

class `peng3d.layer.`**`LayerWorld`**(*menu: peng3d.menu.BasicMenu*, *window: Any = None*, *peng: Any = None*, *world=None*, *viewname: str = None*)

Subclass of *`Layer3D()`* implementing a 3D Layer showing a specific `WorldView`.

All arguments passed to the constructor should be self-explanatory.

Note that you may not set any of the attributes directly, or crashes and bugs may appear indirectly within a certain during future re-drawing of the screen.

`draw`()

Draws the view using the `World.render3d()` method.

`on_menu_enter`(*old*)

Passes the event through to the WorldView to allow for custom behavior.

`on_menu_exit`(*new*)

Same as *`on_menu_enter()`*.

`predraw`()

Sets up the attributes used by *`Layer3D()`* and calls *`Layer3D.predraw()`*.

`setView`(*name: str*) → None

Sets the view used to the specified `name`.

The name must be known to the world or else a `ValueError` is raised.

2.4 `peng3d.menu` - Flexible menu system

class `peng3d.menu.`**`BasicMenu`**(*name: str*, *window: peng3d.window.PengWindow*, *peng: Any = None*)

Menu base class without layer support.

Each menu is separated from the other menus and can be switched between at any time.

Actions supported by default:

`enter` is triggered everytime the *`on_enter()`* method has been called.

`exit` is triggered everytime the *`on_exit()`* method has been called.

See also:

See *`Menu()`* for more information.

`addWorld`(*world*)

Adds the given world to the internal list.

Worlds that are registered via this method will get all events that are given to this menu passed through.

This mechanic is mainly used to implement actor controllers.

`draw`()

This method is called if it is time to render the menu.

Override this method in subclasses to customize behavior and actually draw stuff.

`on_enter`(*old*)

This fake event handler will be called every time this menu is entered via the `PengWindow.changeMenu()` method.

This handler will not be called if this menu is already active.

on_exit(*new*)
This fake event handler will be called every time this menu is exited via the `PengWindow.changeMenu()` method.

This handler will not be called if this menu is the same as the new menu.

class `peng3d.menu.`**Menu**(*name: str*, *window: peng3d.window.PengWindow*, *peng: Any = None*)
Subclass of *`BasicMenu`* adding support for the `Layer` class.

This subclass overrides the draw and __init__ method, so be sure to call them if you override them.

addLayer(*layer: peng3d.layer.Layer*, *z: int = -1*) → None
Adds a new layer to the stack, optionally at the specified z-value.

`layer` must be an instance of Layer or subclasses.

`z` can be used to override the index of the layer in the stack. Defaults to `-1` for appending.

draw() → None
Draws the layers in the appropriate order.

Layers that have their `enabled` attribute set to `False` are skipped.

on_enter(*old*)
Same as *`BasicMenu.on_enter()`*, but also calls `Layer.on_menu_enter()` on every layer.

on_exit(*new*)
Same as *`BasicMenu.on_exit()`*, but also calls `Layer.on_menu_exit()` on every layer.

2.5 `peng3d.gui` - 2D Widget based GUI System

class `peng3d.gui.`**GUIMenu**(*name: str*, *window: peng3d.window.PengWindow*, *peng: Any = None*, *font: Optional[str] = None*, *font_size: Optional[float] = None*, *font_color: Optional[List[int]] = None*, *borderstyle: Union[Type[Borderstyle], str, None] = None*, *style: Optional[Dict[str, Union[float, str, Type[Borderstyle]]]] = None*)
`peng3d.menu.Menu` subclass adding 2D GUI Support.

Note that widgets are not managed directly by this class, but rather by each *`SubMenu`*.

addSubMenu(*submenu: peng3d.gui.SubMenu*) → None
Adds a *`SubMenu`* to this Menu.

Note that nothing will be displayed unless a submenu is activated.

Deprecated since version 1.12.0: This method is no longer needed in most cases, since submenus now register themselves.

changeSubMenu(*submenu: str*) → None
Changes the submenu that is displayed.

Raises `ValueError` – if the name was not previously registered

draw() → None
Draws each menu layer and the active submenu.

Note that the layers are drawn first and may be overridden by the submenu and widgets.

on_enter(*old*)
Same as `BasicMenu.on_enter()`, but also calls `Layer.on_menu_enter()` on every layer.

`submenu`
Property containing the `SubMenu` instance that is currently active.

class `peng3d.gui.`**`SubMenu`**(*name: str, menu: peng3d.gui.GUIMenu, window: Any = None, peng: Any = None, font: Optional[str] = None, font_size: Optional[float] = None, font_color: Optional[List[int]] = None, borderstyle: Union[Type[Borderstyle], str, None] = None, style: Optional[Dict[str, Union[float, str, Type[Borderstyle]]]] = None*)
Sub Menu of the GUI system.

Changed in version 1.12: Sub menus are automatically registered with their parent, using `addSubMenu()` is no longer necessary.

Actions supported by default:

`enter` is triggered everytime the `on_enter()` method has been called.

`exit` is triggered everytime the `on_exit()` method has been called.

`send_form` is triggered if the contained form is sent by either pressing enter or calling `send_form()`.

`addWidget`(*widget: peng3d.gui.widgets.BasicWidget, order_key: int = 0*) → None
Adds a widget to this submenu.

`order_key` optionally specifies the “layer” this widget will be on. Note that this does not work with batched widgets. All batched widgets will be drawn before widgets that use a custom draw() method.

Deprecated since version 1.12.0: This method is no longer needed in most cases, since widgets now register themselves by default.

`delWidget`(*widget: str*) → None
Deletes the widget by the given name.

Note that this feature is currently experimental as there seems to be a memory leak with this method.

`draw`() → None
Draws the submenu and its background.

Note that this leaves the OpenGL state set to 2d drawing.

`form_valid`(*ctx=None*) → bool
Called to pre-check if a form is valid.

Should be overridden by subclasses.

By default, this always returns true.

Parameters **`ctx`** – Arbitrary context

Returns If the form is valid

`getWidget`(*name: str*) → peng3d.gui.widgets.BasicWidget
Returns the widget with the given name.

`send_form`(*ctx=None*) → bool
Triggers whatever form data is entered to be sent.

Only causes action `send_form` to be sent if submenu is active and `form_valid()` returns true.

The given context is stored in `form_ctx`.

Parameters **`ctx`** – Arbitrary context

Returns If the form was actually sent

setBackground(*bg: Union[Layer, Callable, list, tuple, Background, str, Type[DEFER_BG]]*) → None

Sets the background of the submenu.

The background may be a RGB or RGBA color to fill the background with.

Alternatively, a `peng3d.layer.Layer` instance or other object with a `.draw()` method may be supplied. It is also possible to supply any other method or function that will get called.

Also, the strings `flat`, `gradient`, `oldshadow` and `material` may be given, resulting in a background that looks similar to buttons.

If the Background is `None`, the default background of the parent menu will be used.

Lastly, the string `"blank"` may be passed to skip background drawing.

class peng3d.gui.**GUILayer**(*name: str, menu: peng3d.menu.Menu, window: Optional[peng3d.window.PengWindow] = None, peng: Optional[peng3d.Peng] = None*)

Hybrid of `GUIMenu` and `peng3d.layer.Layer2D`.

This class allows you to create Head-Up Displays and other overlays easily.

draw() → None

Draws the Menu.

2.6 `peng3d.gui.widgets` - 2D GUI Widget Base classes

class peng3d.gui.widgets.**BasicWidget**(*name: Optional[str], submenu: SubMenu, window: Any = None, peng: Any = None, *, pos: Union[List[float], Callable[[float, float, float, float], Tuple[float, float]], layout.LayoutCell], size: Union[List[float], Callable[[float, float], Tuple[float, float]], None] = None, order_key: Optional[int] = 0, style: Optional[Dict[str, Union[float, str, Type[Borderstyle]]]] = None*)

Basic Widget class.

`pos` may be either a list or 2-tuple of `(x,y)` for static positions or a function with the signature `window_width,window_height,widget_width,widget_height` returning a tuple.

`size` is similar to `pos` but will only get `window_width,window_height` as its arguments.

Commonly, the lambda `lambda sw,sh,bw,bh:  (sw/2.-bw/2.,sh/2.-bh/2.)` is used to center the widget.

Additionally, an instance of a subclass of `LayoutCell` may be passed as `pos`. Note that this will automatically override `size` as well, unless `size` is also supplied.

The actions available may differ from widget to widget, by default these are used:

- `press` is called upon starting to click on the widget
- `click` is called if the mouse is released on the widget while also having been pressed on it before, recommended for typical button callbacks
- `context` is called upon right-clicking on the widget and may be used to display a context menu
- `hover_start` signals that the cursor is now hovering over the widget
- `hover` is called every time the cursor moves while still being over the widget

- `hover_end` is called after the cursor leaves the widget
- `statechanged` is called every time the visual state of the widget should change

Deprecated since version 1.12: The `window` and `peng` parameters are deprecated and will be removed in peng3d 2.0. They are no longer needed and should be removed from existing code.

Changed in version 1.12: It is no longer necessary to register widgets using *`addWidget()`*, widget registration is now automatically performed by widgets themselves.

`IS_CLICKABLE = False`

Class attribute used to signal if widgets of this class are usually clickable.

This attribute is used to fill the initial value of *`enabled`* and can therefore be overridden on a widget-by-widget basis.

Note that leaving this set to `False` will prevent most mouse-related actions from being triggered. This is due to internal optimization and the main benefit of leaving this option off.

`clickable`

Property used for determining if the widget should be clickable by the user.

This is only true if the submenu of this widget is active and this widget is enabled.

The widget may be either disabled by setting this property or the *`enabled`* attribute.

`delete``()`

Deletes resources of this widget that require manual cleanup.

Currently removes all actions, event handlers and the background.

The background itself should automatically remove all vertex lists to avoid visual artifacts.

Note that this method is currently experimental, as it seems to have a memory leak.

`draw``()` → None

Draws all vertex lists associated with this widget.

`enabled`

Property used for storing whether or not this widget is enabled.

May influence rendering and behavior.

Note that the widget will be immediately redrawn if this property is changed.

`getState``()` → str

Returns the current state of the widget.

One of `"pressed"`, `"hover"`, `"disabled"` or `"idle"`. Note that some information may be lost by getting this state, for example it is not possible to know if the widget is hovered or not if `"pressed"` is returned. However, this should not be a problem for most intended uses of this method.

`on_redraw``()` → None

Callback to be overridden by subclasses called if redrawing the widget seems necessary.

Note that this method should not be called manually, see *`redraw()`* instead.

`pos`

Property that will always be a 2-tuple representing the position of the widget.

Note that this method may call the method given as `pos` in the initializer.

The returned object will actually be an instance of a helper class to allow for setting only the x/y coordinate.

This property also respects any `Container` set as its parent, any offset will be added automatically.

Note that setting this property will override any callable set permanently.

redraw() → None
Triggers a redraw of the widget.

Note that the redraw may not be executed instantly, but rather batched together on the next frame. If an instant and synchronous redraw is needed, use *on_redraw()* instead.

registerEventHandlers()
Registers event handlers used by this widget, e.g. mouse click/motion and window resize.

This will allow the widget to redraw itself upon resizing of the window in case the position needs to be adjusted.

size
Similar to *pos* but for the size instead.

visible
Property used for storing whether or not this widget is enabled.

May influence rendering and behavior.

Note that the widget will be immediately redrawn if this property is changed.

class peng3d.gui.widgets.**Background**(*widget: peng3d.gui.widgets.BasicWidget*)
Class representing the background of a widget.

Note that if a background is used as the background of a SubMenu, the SubMenu instance itself should be passed as the widget.

This base class does not do anything.

init_bg() → None
Called just before the background will be drawn the first time.

Commonly used to initialize vertex lists.

It is recommended to add all vertex lists to the `submenu.batch2d` Batch to speed up rendering and preventing glitches with grouping.

is_hovering
Read-only helper property for easier access.

Equivalent to `widget.is_hovering`.

peng
Property for accessing the parent widget's instance of *peng3d.peng.Peng*.

pressed
Read-only helper property for easier access.

Equivalent to `widget.pressed`.

redraw_bg() → None
Method called by the parent widget every time its `Widget.redraw()` method is called.

reg_vlist(*vlist: pyglet.graphics.vertexdomain.VertexList*) → None
Registers a vertex list to the internal list.

This allows the class to clean itself up properly during deletion, as the background would still be visible after deletion otherwise.

submenu
Property for accessing the parent widget's submenu.

window
Property for accessing the parent widget's window.

peng3d.gui.widgets.**DEFER_BG = <peng3d.gui.widgets._DeferBackgroundSentinel object>**

Sentinel object that may be passed instead of an actual background to signify that the background will be set later.

Differs from passing `None`, since `None` will cause an *EmptyBackground* to be unnecessarily created, while *DEFER_BG* simply does nothing.

Note that if the actual background is not set before the first render, a `TypeError` will be raised.

class peng3d.gui.widgets.**Widget**(*name: Optional[str], submenu: SubMenu, window: Any = None, peng: Any = None, *, pos: Union[List[float], Callable[[float, float, float, float], Tuple[float, float]], layout.LayoutCell], size: Union[List[float], Callable[[float, float], Tuple[float, float]], None] = None, bg: peng3d.gui.widgets.Background = None, min_size: Optional[List[float]] = None*)

Subclass of *BasicWidget* adding support for changing the *Background*.

If no background is given, an *EmptyBackground* will be used instead.

on_redraw()

Draws the background and the widget itself.

Subclasses should use `super()` to call this method, or rendering may glitch out.

setBackground(*bg: peng3d.gui.widgets.Background*) → peng3d.gui.widgets.Background

Sets the background of the widget.

This may cause the background to be initialized.

class peng3d.gui.widgets.**EmptyBackground**(*widget: peng3d.gui.widgets.BasicWidget*)

Background that draws simply nothing.

Can be used as a placeholder.

2.7 `peng3d.gui.button` - Button Widgets

class peng3d.gui.button.**Button**(*name: Optional[str], submenu: SubMenu, window: Any = None, peng: Any = None, *, pos: Union[List[float], Callable[[float, float, float, float], Tuple[float, float]], layout.LayoutCell], size: Union[List[float], Callable[[float, float], Tuple[float, float]], None] = None, bg: peng3d.gui.widgets.Background = None, border: Optional[Tuple[float, float]] = None, borderstyle: Union[Type[Borderstyle], str, None] = None, label: Union[str, _LazyTranslator] = 'Button', min_size=None, font_size: Optional[float] = None, font=None, font_color=None, label_layer=1*)

Button Widget allowing the user to trigger specific actions.

By default, this Widget uses *ButtonBackground* as its Background class.

The border given is in pixels from the left/right and top/bottom, respectively.

The borderstyle may be either `flat`, which has no border at all, `gradient`, which fades from the inner color to the background color, `oldshadow`, which uses a simple fake shadow with the light from the top-left corner and `material`, which imitates Google Material Design shadows.

Also, the label of the button may only be a single line of text, anything else may produce undocumented behavior.

If necessary, the font size of the Label may be changed via the global Constant `LABEL_FONT_SIZE`, changes will only apply to Buttons created after change. The text color used is `[62,67,73,255]` in RGBA and the font used is Arial, which should be available on most systems.

delete()
Deletes resources of this widget that require manual cleanup.

Currently removes all actions, event handlers and the background.

The background itself should automatically remove all vertex lists to avoid visual artifacts.

Note that this method is currently experimental, as it seems to have a memory leak.

label
Property for accessing the label of this Button.

on_redraw()
Draws the background and the widget itself.

Subclasses should use `super()` to call this method, or rendering may glitch out.

redraw_label()
Re-draws the label by calculating its position.

Currently, the label will always be centered on the Button.

class peng3d.gui.button.**ButtonBackground**(*widget*, *border=None*, *borderstyle='flat'*, *batch=None*, *change_on_press=None*)
Background for the *Button* Widget.

This background renders the button and its border, but not the label.

getColors()
Overrideable function that generates the colors to be used by various borderstyles.

Should return a 5-tuple of `(bg,o,i,s,h)`.

`bg` is the base color of the background.

`o` is the outer color, it is usually the same as the background color.

`i` is the inner color, it is usually lighter than the background color.

`s` is the shadow color, it is usually quite a bit darker than the background.

`h` is the highlight color, it is usually quite a bit lighter than the background.

getPosSize()
Helper function converting the actual widget position and size into a usable and offsetted form.

This function should return a 6-tuple of `(sx,sy,x,y,bx,by)` where sx and sy are the size, x and y the position and bx and by are the border size.

All values should be in pixels and already include all offsets, as they are used directly for generation of vertex data.

This method can also be overridden to limit the background to a specific part of its widget.

init_bg()
Called just before the background will be drawn the first time.

Commonly used to initialize vertex lists.

It is recommended to add all vertex lists to the `submenu.batch2d` Batch to speed up rendering and preventing glitches with grouping.

`is_hovering`
Read-only helper property to be used by borderstyles for determining if the widget should be rendered as hovered or not.

Note that this property may not represent the actual hovering state, it will always be False if `change_on_press` is disabled.

`pressed`
Read-only helper property to be used by borderstyles for determining if the widget should be rendered as pressed or not.

Note that this property may not represent the actual pressed state, it will always be False if `change_on_press` is disabled.

`redraw_bg``()`
Method called by the parent widget every time its `Widget.redraw()` method is called.

class `peng3d.gui.button.`**`ImageButton`**`(`*name: Optional[str], submenu: SubMenu, window: Any = None, peng: Any = None, *, pos: Union[List[float], Callable[[float, float, float, float], Tuple[float, float]], layout.LayoutCell], size: Union[List[float], Callable[[float, float], Tuple[float, float]], None] = None, bg=None, label='Button', font_size=None, font=None, font_color=None, bg_idle=None, bg_hover=None, bg_disabled=None, bg_pressed=None, label_layer=1*`)`
Subclass of *`Button`* using an image as a background instead.

By default, this Widget uses *`ImageBackground`* as its Background class.

There are no changes to any other mechanics of the Button, only visually.

class `peng3d.gui.button.`**`ImageBackground`**`(`*widget*, *bg_idle=None*, *bg_hover=None*, *bg_disabled=None*, *bg_pressed=None*`)`
Background for the *`ImageButton`* Widget.

This background renders a image given based on whether the widget is pressed, hovered over or disabled.

It should also be possible to use this class as a background for most other Widgets.

`init_bg``()`
Called just before the background will be drawn the first time.

Commonly used to initialize vertex lists.

It is recommended to add all vertex lists to the `submenu.batch2d` Batch to speed up rendering and preventing glitches with grouping.

`pressed`
Read-only helper property to be used by borderstyles for determining if the widget should be rendered as pressed or not.

Note that this property may not represent the actual pressed state, it will always be False if `change_on_press` is disabled.

`redraw_bg``()`
Method called by the parent widget every time its `Widget.redraw()` method is called.

class peng3d.gui.button.**FramedImageButton**(*name: Optional[str], submenu: SubMenu, window: Any = None, peng: Any = None, *, pos: Union[List[float], Callable[[float, float, float, float], Tuple[float, float]], layout.LayoutCell], size: Union[List[float], Callable[[float, float], Tuple[float, float]], None] = None, bg=None, label='Button', font_size=None, font=None, font_color=None, bg_idle=None, bg_hover=None, bg_disabled=None, bg_pressed=None, frame=[[2, 10, 2], [2, 10, 2]], scale=(1, 1), repeat_edge=False, repeat_center=False, tex_size=None, label_layer=1*)

Subclass of `ImageButton` adding smart scaling to the background.

By default, this Widget uses `FramedImageBackground` as its Background class.

`frame` defines the ratio between the borders and the center. The sum of each item must be greater than zero, else a ZeroDivisionError may be thrown. Note that up to two items of each frame may be left as `0`. This will cause the appropriate border or center to not be rendered at all.

`tex_size` may be left empty if a resource name is passed. It will then be automatically determined.

Todo: Document `scale`

class peng3d.gui.button.**FramedImageBackground**(*widget, bg_idle=None, bg_hover=None, bg_disabled=None, bg_pressed=None, frame=[[2, 10, 2], [2, 10, 2]], scale=(0, 0), repeat_edge=False, repeat_center=False, tex_size=None*)

Background for the `FramedImageButton` Widget.

This background is similar to `ImageBackground`, but it attempts to scale smarter with less artifacts.

init_bg()

Called just before the background will be drawn the first time.

Commonly used to initialize vertex lists.

It is recommended to add all vertex lists to the `submenu.batch2d` Batch to speed up rendering and preventing glitches with grouping.

redraw_bg()

Method called by the parent widget every time its `Widget.redraw()` method is called.

class peng3d.gui.button.**ToggleButton**(*name: Optional[str], submenu: SubMenu, window: Any = None, peng: Any = None, *, pos: Union[List[float], Callable[[float, float, float, float], Tuple[float, float]], layout.LayoutCell], size: Union[List[float], Callable[[float, float], Tuple[float, float]], None] = None, bg: peng3d.gui.widgets.Background = None, border: Optional[Tuple[float, float]] = None, borderstyle: Union[Type[Borderstyle], str, None] = None, label: Union[str, _LazyTranslator] = 'Button', min_size=None, font_size: Optional[float] = None, font=None, font_color=None, label_layer=1*)

Variant of `Button` that stays pressed until clicked again.

This widgets adds the following actions:

- `press_down` is called upon depressing the button
- `press_up` is called upon releasing the button
- `click` is changed to be called on every click on the button, e.g. like `press_down` and `press_up` combined

class `peng3d.gui.button.`**`Checkbox`**(*name: Optional[str], submenu: SubMenu, window: Any = None, peng: Any = None, *, pos: Union[List[float], Callable[[float, float, float, float], Tuple[float, float]], layout.LayoutCell], size: Union[List[float], Callable[[float, float], Tuple[float, float]], None] = None, bg=None, borderstyle=None, label='Checkbox', checkcolor=[240, 119, 70], font_size=None, font=None, font_color=None, label_layer=1*)

Variant of *`ToggleButton`* using a different visual indicator.

By default, this Widget uses *`CheckboxBackground`* as its Background class.

Note that the position and size given are for the indicator, the label will be bigger than the given size.

The label given will be displayed to the right of the Checkbox.

`redraw_label`()

Re-calculates the position of the Label.

class `peng3d.gui.button.`**`CheckboxBackground`**(*widget, borderstyle, checkcolor=[240, 119, 70], **kwargs*)

Background for the *`Checkbox`* Widget.

This background looks like a button, but adds a square in the middle if it is pressed.

The color of the square defaults to a tone of orange commonly found in GTK GUIs on Ubuntu.

`init_bg`()

Called just before the background will be drawn the first time.

Commonly used to initialize vertex lists.

It is recommended to add all vertex lists to the `submenu.batch2d` Batch to speed up rendering and preventing glitches with grouping.

`redraw_bg`()

Method called by the parent widget every time its `Widget.redraw()` method is called.

2.8 `peng3d.gui.menus` - Menus and Dialogs

Menus are special submenus that act like modal dialogs.

They include glue code that automatically switches back to the previous submenu after they are left. Note that this will cuase the `SubMenu.on_enter()` method to be called again.

Since these menus are internally implemented as submenus, they are specific to their `Menu`, which must be active to be able to use the dialog.

2.8.1 Customization

Menus are customizable via several different means.

If you just want to change the appearance or label of a part of the menu, you can use keyword arguments while initializing the class. For example, setting the `label_main` argument to the string `Hello World!` the main label or title of the dialog will now display `Hello World!` instead of its default value. What exact arguments are supported differs from dialog to dialog.

Note that sometimes specific labels are supported, but not used by default. Just setting these to anything may cause GUI components to be rendered that should not be there.

It is also possible for most of these values to be set on-the-fly via properties on the object they belong to.

For example, the *`DialogSubMenu.label_main`* property may be set to change the main label even while the dialog is active.

Note that the values accessible via keyword arguments and properties may differ. This depends on the dialog implementing them.

For clarity, these keyword arguments will from now on be called “labels”. This also includes labels that are not strictly text, like the maximum value of a progressbar.

class `peng3d.gui.menus.`**`DialogSubMenu`**(*name*, *menu*, *window=None*, *peng=None*, *borderstyle=None*, *font_size=None*, *font=None*, *font_color=None*, *multiline=False*, ***kwargs*)

Base Dialog Class.

This class acts as a base class for all other dialog submenus.

When the dialog is entered, the `prev_submenu` attribute will be set to the name of the previous submenu. This attribute is later used when exiting the dialog.

Dialog submenus also support the basic actions used by all submenus, e.g. `enter` and `exit`. Additionally, many dialogs also add actions for whenever a label is changed or the dialog is exited through a special means, e.g. pressing a specific button of multiple presented.

If used by itself, it will display a text centered on the screen with a button below it. Clicking the button will cause the dialog to exit and also the additional `click_ok` action to be fired.

The labels supported by default are `label_main`, which defaults to `Default Text` and is recommended to always be customized, and `label_ok`, which defaults to `OK` and may be left as-is.

Subclasses may override these defaults by setting the keys of the same name in the `DEFAULT_LABELS` class attribute. Note that any unchanged labels must also be declared when overwriting any labels, or they may not be displayed.

Widgets and their initializers are stored in the `WIDGETS` class attribute, see *`add_widgets()`* for more information.

`activate`()

Helper method to enter the dialog.

Calling this method will simply cause the dialog to become the active submenu.

Note that is not necessary to call this method over `changeSubMenu()`, as the storing of the previous submenu is done elsewhere.

`add_btn_ok`(*label_ok*)

Adds an OK button to allow the user to exit the dialog.

This widget can be triggered by setting the label `label_ok` to a string.

This widget will be mostly centered on the screen, but below the main label by the double of its height.

`add_label_main`(*label_main*)

Adds the main label of the dialog.

This widget can be triggered by setting the label `label_main` to a string.

This widget will be centered on the screen.

add_widgets(***kwargs*)

Called by the initializer to add all widgets.

Widgets are discovered by searching through the `WIDGETS` class attribute. If a key in `WIDGETS` is also found in the keyword arguments and not none, the function with the name given in the value of the key will be called with its only argument being the value of the keyword argument.

For more complex usage scenarios, it is also possible to override this method in a subclass, but the original method should always be called to ensure compatibility with classes relying on this feature.

exitDialog()

Helper method that exits the dialog.

This method will cause the previously active submenu to activate.

label_main

Property that proxies the `label_main` label.

Setting this property will cause the `label_main_change` action to trigger.

Note that trying to access this property if the widget is not used may cause an error.

label_ok

Property that proxies the `label_ok` label.

Setting this property will cause the `label_ok_change` action to trigger.

Note that trying to access this property if the widget is not used may cause an error.

class peng3d.gui.menus.**ConfirmSubMenu**(*name*, *menu*, *window=None*, *peng=None*, *borderstyle=None*, *font_size=None*, *font=None*, *font_color=None*, *multiline=False*, ***kwargs*)

Dialog that allows the user to confirm or cancel an action.

By default, the OK button will be hidden and the `label_main` will be set to `Are you sure?`.

Clicking the confirm button will cause the `confirm` action to trigger, while the cancel button will cause the `cancel` action to trigger.

add_btn_cancel(*label_cancel*)

Adds a cancel button to let the user cancel whatever choice they were given.

This widget can be triggered by setting the label `label_cancel` to a string.

This widget will be positioned slightly below the main label and to the right of the confirm button.

add_btn_confirm(*label_confirm*)

Adds a confirm button to let the user confirm whatever action they were presented with.

This widget can be triggered by setting the label `label_confirm` to a string.

This widget will be positioned slightly below the main label and to the left of the cancel button.

label_cancel

Property that proxies the `label_cancel` label.

Setting this property will cause the `label_cancel_change` action to trigger.

Note that trying to access this property if the widget is not used may cause an error.

label_confirm

Property that proxies the `label_confirm` label.

Setting this property will cause the `label_confirm_change` action to trigger.

Note that trying to access this property if the widget is not used may cause an error.

class `peng3d.gui.menus.`**`TextSubMenu`**(*name*, *menu*, *window*, *peng*, *timeout=10*, ***kwargs*)

Dialog without user interaction that can automatically exit after a certain amount of time.

This dialog accepts the `timeout` keyword argument, which may be set to any time in seconds to delay before exiting the dialog. A value of `-1` will cause the dialog to never exit on its own.

Note that the user will not be able to exit this dialog and may believe the program is hanging if not assured otherwise. It is thus recommended to use the *`ProgressSubMenu`* dialog instead, especially for long-running operations.

class `peng3d.gui.menus.`**`ProgressSubMenu`**(*name*, *menu*, *window=None*, *peng=None*, *borderstyle=None*, *font_size=None*, *font=None*, *font_color=None*, *multiline=False*, ***kwargs*)

Dialog without user interaction displaying a progressbar.

By default, the progressbar will range from 0-100, effectively a percentage.

The *`auto_exit`* attribute may be set to control whether or not the dialog will exit automatically when the maximum value is reached.

`add_progressbar`(*label_progressbar*)

Adds a progressbar and label displaying the progress within a certain task.

This widget can be triggered by setting the label `label_progressbar` to a string.

The progressbar will be displayed centered and below the main label. The progress label will be displayed within the progressbar.

The label of the progressbar may be a string containing formatting codes which will be resolved via the `format()` method.

Currently, there are six keys available:

`n` and `value` are the current progress rounded to 4 decimal places.

`nmin` is the minimum progress value rounded to 4 decimal places.

`nmax` is the maximum progress value rounded to 4 decimal places.

`p` and `percent` are the percentage value that the progressbar is completed rounded to 4 decimal places.

By default, the progressbar label will be `{percent}%` displaying the percentage the progressbar is complete.

`auto_exit = False`

Controls whether or not the dialog will exit automatically after the maximum value has been reached.

`label_progressbar`

Property that proxies the `label_progressbar` label.

Setting this property will cause the progressbar label to be recalculated.

Note that setting this property if the widget has not been initialized may cause various errors to occur.

`progress_n`

Property that proxies the `progress_n` label.

Setting this property will cause the progressbar label to be recalculated.

Additionally, if the supplied value is higher than the maximum value and *`auto_exit`* is true, the dialog will exit.

progress_nmax
Property that proxies the `progress_nmax` label.

Setting this property will cause the progressbar label to be recalculated.

Note that setting this property if the widget has not been initialized may cause various errors to occur.

progress_nmin
Property that proxies the `progress_nmin` label.

Setting this property will cause the progressbar label to be recalculated.

Note that setting this property if the widget has not been initialized may cause various errors to occur.

update_progressbar()
Updates the progressbar by re-calculating the label.

It is not required to manually call this method since setting any of the properties of this class will automatically trigger a re-calculation.

class peng3d.gui.menus.**AdvancedProgressSubMenu**(*name*, *menu*, *window=None*, *peng=None*, *borderstyle=None*, *font_size=None*, *font=None*, *font_color=None*, *multiline=False*, ***kwargs*)

addCategory(**args*, ***kwargs*)
Proxy for `addCategory()`.

add_progressbar(*label_progressbar*)
Adds a progressbar and label displaying the progress within a certain task.

This widget can be triggered by setting the label `label_progressbar` to a string.

The progressbar will be displayed centered and below the main label. The progress label will be displayed within the progressbar.

The label of the progressbar may be a string containing formatting codes which will be resolved via the `format()` method.

Currently, there are six keys available:

`n` and `value` are the current progress rounded to 4 decimal places.

`nmin` is the minimum progress value rounded to 4 decimal places.

`nmax` is the maximum progress value rounded to 4 decimal places.

`p` and `percent` are the percentage value that the progressbar is completed rounded to 4 decimal places.

By default, the progressbar label will be `{percent}%` displaying the percentage the progressbar is complete.

deleteCategory(**args*, ***kwargs*)
Proxy for `deleteCategory()`.

updateCategory(**args*, ***kwargs*)
Proxy for `updateCategory()`.

2.9 peng3d.gui.layout - Layout Helper Classes

class peng3d.gui.layout.**Layout**(*peng*, *parent*)
Base Layout class.

This class does not serve any purpose directly other than to be a common base class for all layouts.

Note that layouts can be nested, e.g. usually the first layouts parent is a SubMenu and sub-layouts get a LayoutCell of their parent layout as their parent.

class `peng3d.gui.layout.`**`GridLayout`**(*peng*, *parent*, *res*, *border*)

Grid-based layout helper class.

This class provides a grid-like layout to its sub-widgets. A border between widgets can be defined. Additionally, all widgets using this layout should automatically scale with screen size.

`cell_size`

Helper property defining the current size of cells in both x and y axis.

Returns 2-tuple of float

`get_cell`(*pos*, *size*, *anchor_x='left'*, *anchor_y='bottom'*, *border=1*)

Returns a grid cell suitable for use as the `pos` parameter of any widget.

The `size` parameter of the widget will automatically be overwritten.

Parameters

- **`pos`** – Grid position, in cell
- **`size`** – Size, in cells
- **`anchor_x`** – either `left`, `center` or `right`
- **`anchor_y`** – either `bottom`, `center` or `top`

Returns LayoutCell subclass

class `peng3d.gui.layout.`**`LayoutCell`**

Base Layout Cell.

Not to be used directly. Usually subclasses of this class are returned by layouts.

Instances can be passed to Widgets as the `pos` argument. The `size` argument will be automatically overridden.

Note that manually setting `size` will override the size set by the layout cell, though the position will be kept.

`pos`

Property accessing the position of the cell.

This usually refers to the bottom-left corner, but may change depending on arguments passed during creation.

Note that results can be floats.

Returns 2-tuple of `(x,y)`

`size`

Property accessing the size of the cell.

Note that results can be floats.

Returns 2-tuple of `(width, height)`

2.10 `peng3d.gui.layered` - Layered Widgets

class `peng3d.gui.layered.`**`LayeredWidget`**(*name: Optional[str], submenu: SubMenu, window: Any = None, peng: Any = None, *, pos: Union[List[float], Callable[[float, float, float, float], Tuple[float, float]], layout.LayoutCell], size: Union[List[float], Callable[[float, float], Tuple[float, float]], None] = None, bg=None, layers=[]*)

Layered Widget allowing for easy creation of custom widgets.

A Layered Widget consists of (nearly) any amount of layers in a specific order.

All Layers should be subclasses of *`BasicWidgetLayer`* or *`WidgetLayer`*.

`layers` must be a list of 2-tuples of `(layer, z_index)`.

`addLayer`(*layer, z_index=None*)

Adds the given layer at the given Z Index.

If `z_index` is not given, the Z Index specified by the layer will be used.

`delete`()

Deletes all layers within this LayeredWidget before deleting itself.

Recommended to call if you are removing the widget, but not yet exiting the interpreter.

`draw`()

Draws all layers of this LayeredWidget.

This should normally be unneccessary, since it is recommended that layers use Vertex Lists instead of OpenGL Immediate Mode.

`getLayer`(*name*)

Returns the layer corresponding to the given name.

Raises **`KeyError`** – If there is no Layer with the given name.

`on_redraw`()

Draws the background and the widget itself.

Subclasses should use `super()` to call this method, or rendering may glitch out.

`redraw_layer`(*name*)

Redraws the given layer.

Raises **`ValueError`** – If there is no Layer with the given name.

class `peng3d.gui.layered.`**`BasicWidgetLayer`**(*name, widget, z_index=None*)

Base class for all Layers to be used with *`LayeredWidget()`*.

Not to be confused with *`peng3d.layer.Layer()`*, these classes are not compatible.

It is recommended to use *`WidgetLayer()`* instead, since functionality is limited in this basic class.

Note that the `z_index` will default to a reasonable value for most subclasses and thus is not required to be given explicitly. The `z_index` for this Layer defaults to `0`.

`delete`()

Deletes this Layer.

Currently only deletes VertexLists registered with *`regVList()`*.

draw()
Called to draw the layer.

Note that using this function is discouraged, use Pyglet Vertex Lists instead.

If you want to call this method manually, call `_draw()` instead. This will make sure that *predraw()* and *postdraw()* are called.

on_redraw()
Called by the parent widget if this Layer should be redrawn.

Note that it is recommended to call the Baseclass Variant of this Method first when overwriting it. See *WidgetLayer.on_redraw()* for more information.

postdraw()
Called after calling the *draw()* Method.

Useful for unsetting OpenGL state.

predraw()
Called before calling the *draw()* Method.

Useful for setting up OpenGL state.

regVList(*vlist*)
Registers a vertex list for proper deletion once this Layer gets destroyed.

This prevents visual artifacts from forming during deletion of a layer.

class peng3d.gui.layered.**WidgetLayer**(*name, widget, z_index=None, border=[0, 0], offset=[0, 0]*)
Subclass of *WidgetLayer()* adding commonly used utility features.

This subclass adds a border and offset system.

The `border` is a 2-tuple of `(x_border,y_border)`. The border is applied to all sides, resulting in the size being decreased by two pixel per pixel border width.

`offset` is relative to the bottom left corner of the screen.

border
Property to be used for setting and getting the border of the layer.

Note that setting this property causes an immediate redraw.

getPos()
Returns the absolute position and size of the layer.

This method is intended for use in vertex position calculation, as the border and offset have already been applied.

The returned value is a 4-tuple of `(sx,sy,ex,ey)`. The two values starting with an s are the "start" position, or the lower-left corner. The second pair of values signify the "end" position, or upper-right corner.

getSize()
Returns the size of the layer, with the border size already subtracted.

initialize()
Called just before *on_redraw()* is called the first time.

offset
Property to be used for setting and getting the offset of the layer.

Note that setting this property causes an immediate redraw.

on_redraw()
Called when the Layer should be redrawn.

If a subclass uses the *initialize()* Method, it is very important to also call the Super Class Method to prevent crashes.

class peng3d.gui.layered.**GroupWidgetLayer**(*name, widget, group=None, z_index=None, border=[0, 0], offset=[0, 0]*)
Subclass of *WidgetLayer()* allowing for using a pyglet group to manage OpenGL state.

If no pyglet group is given, `pyglet.graphics.NullGroup()` will be used.

postdraw()
Called after calling the `draw()` Method.

Useful for unsetting OpenGL state.

predraw()
Called before calling the `draw()` Method.

Useful for setting up OpenGL state.

class peng3d.gui.layered.**ImageWidgetLayer**(*name, widget, z_index=None, border=[0, 0], offset=[0, 0], img=[None, None]*)
Subclass of *WidgetLayer()* implementing a simple static image view.

This layer can display any resource representable by the `ResourceManager()`.

`img` is a 2-tuple of `(resource_name,category)`.

The `z_index` for this Layer defaults to `1`.

initialize()
Called just before *on_redraw()* is called the first time.

on_redraw()
Called when the Layer should be redrawn.

If a subclass uses the *initialize()* Method, it is very important to also call the Super Class Method to prevent crashes.

class peng3d.gui.layered.**DynImageWidgetLayer**(*name, widget, z_index=None, border=[0, 0], offset=[0, 0], imgs={}, default=None*)
Subclass of *WidgetLayer* allowing for dynamic images.

`imgs` is a dictionary of names to 2-tuples of `(resource_name,category)`.

If no default image name is given, a semi-random one will be selected.

The `z_index` for this Layer defaults to `1`.

addImage(*name, rsrc*)
Adds an image to the internal registry.

`rsrc` should be a 2-tuple of `(resource_name,category)`.

initialize()
Called just before *on_redraw()* is called the first time.

on_redraw()
Called when the Layer should be redrawn.

If a subclass uses the *initialize()* Method, it is very important to also call the Super Class Method to prevent crashes.

switchImage(*name*)
Switches the active image to the given name.

Raises `ValueError` – If there is no such image

class peng3d.gui.layered.**FramedImageWidgetLayer**(*name, widget, z_index=None, border=[0, 0], offset=[0, 0], imgs={}, default=None, frame=[[2, 10, 2], [2, 10, 2]], scale=(0, 0), repeat_edge=False, repeat_center=False, tex_size=None*)
Subclass of `DynImageWidgetLayer` allowing for dynamically smart scaled images.

Similar to `FramedImageButton`. Allows for scaling and/or repeating the borders, corners and center independently.

Note that the `tex_size` parameter, if not given, will be derived from a random texture that has been given in `imgs`. Also note that the `frame`, `scale`, `repeat_edge` and `repeat_center` parameters are identical for all images.

initialize()
Called just before `on_redraw()` is called the first time.

on_redraw()
Called when the Layer should be redrawn.

If a subclass uses the `initialize()` Method, it is very important to also call the Super Class Method to prevent crashes.

class peng3d.gui.layered.**ImageButtonWidgetLayer**(*name, widget, z_index=None, border=[0, 0], offset=[0, 0], img_idle=None, img_pressed=None, img_hover=None, img_disabled=None*)
Subclass of `DynImageWidgetLayer()` that acts like an `ImageButton()`.

The `img_*` arguments are of the same format as in `DynImageWidgetLayer()`.

This class internally uses the `BasicWidget.getState()` method for getting the state of the widget.

class peng3d.gui.layered.**LabelWidgetLayer**(*name, widget, z_index=None, border=[0, 0], offset=[0, 0], label=", font_size=None, font=None, font_color=None, multiline=False, style=None*)
Subclass of `WidgetLayer()` displaying arbitrary plain text.

Note that this method internally uses a pyglet Label that is centered on the Layer.

The `z_index` for this Layer defaults to `2`.

label
Property for accessing the text of the label.

on_redraw(*dt=None*)
Called when the Layer should be redrawn.

If a subclass uses the `initialize()` Method, it is very important to also call the Super Class Method to prevent crashes.

redraw_label()
Re-draws the text by calculating its position.

Currently, the text will always be centered on the position of the layer.

class peng3d.gui.layered.**FormattedLabelWidgetLayer**(*name, widget, z_index=None, border=[0, 0], offset=[0, 0], label=”, font_size=None, font=None, font_color=None, multiline=False, style=None*)

Subclass of *WidgetLayer()* serving as a base class for other formatted label layers.

The Label Type can be set via the class attribute `cls`, it should be set to any class that is compatible with `pyglet.text.Label`.

It is recommended to use one of the subclasses of this class instead of this class directly.

The `z_index` for this Layer defaults to `2`.

label

Property for accessing the text of the label.

Note that depending on the type of format, this property may not exactly represent the original text as it is converted internally.

on_redraw(*dt=None*)

Called when the Layer should be redrawn.

If a subclass uses the `initialize()` Method, it is very important to also call the Super Class Method to prevent crashes.

redraw_label()

Re-draws the text by calculating its position.

Currently, the text will always be centered on the position of the layer.

class peng3d.gui.layered.**HTMLLabelWidgetLayer**(*name, widget, z_index=None, border=[0, 0], offset=[0, 0], label=”, font_size=None, font=None, font_color=None, multiline=False, style=None*)

Subclass of *FormattedLabelWidgetLayer* implementing a basic HTML Label.

Note that not all tags are supported, see the docs for `pyglet.text.HTMLLabel` for details.

class peng3d.gui.layered.**BaseBorderWidgetLayer**(*name, widget, z_index=None, base_border=[0, 0], base_offset=[0, 0], border=[4, 4, 4, 4, 4, 4, 4, 4], style='flat', batch=None, change_on_press=None*)

Subclass of *WidgetLayer* that displays a basic border around the layer.

Note that not all styles will look good with this class, see *ButtonBorderWidgetLayer()* for more information.

Note that the `border` and `offset` arguments have been renamed to `base_border` and `base_offset` to prevent naming conflicts.

Subclasses may set the `n_vertices` value to change the number of vertices or `change_on_press` to change the default value for the argument of the same name. By default, 36 vertices are used and `changed_on_press` is set to `True`.

The `z_index` for this Layer defaults to `0.5`.

addStyle(*name, func*)

Adds a style to the layer.

Note that styles must be registered seperately for each layer.

`name` is the (string) name of the style.

`func` will be called with its arguments as `(bg,o,i,s,h)`, see `getColors()` for more information.

`genVertices``()`
Called to generate the vertices used by this layer.

The length of the output of this method should be three times the `n_vertices` attribute.

See the source code of this method for more information about the order of the vertices.

`getColors``()`
Overrideable function that generates the colors to be used by various styles.

Should return a 5-tuple of `(bg,o,i,s,h)`.

`bg` is the base color of the background.

`o` is the outer color, it is usually the same as the background color.

`i` is the inner color, it is usually lighter than the background color.

`s` is the shadow color, it is usually quite a bit darker than the background.

`h` is the highlight color, it is usually quite a bit lighter than the background.

The returned values may also be statically overridden by setting the `color_` attribute to anything but `None`.

`initialize``()`
Called just before `on_redraw()` is called the first time.

`is_hovering`
Read-only helper property to be used by styles for determining if the layer should be rendered as hovered or not.

Note that this property may not represent the actual hovering state, it will always be False if `change_on_press` is disabled.

`on_redraw``()`
Called when the Layer should be redrawn.

If a subclass uses the `initialize()` Method, it is very important to also call the Super Class Method to prevent crashes.

`pressed`
Read-only helper property to be used by styles for determining if the layer should be rendered as pressed or not.

Note that this property may not represent the actual pressed state, it will always be False if `change_on_press` is disabled.

`stretchColors``(`*c*`)`
Method that is called to stretch the colors.

Note that this should be implemented by subclasses if plausible and reasonable.

`class` `peng3d.gui.layered.`**`ButtonBorderWidgetLayer`**`(`*name, widget, z_index=None, base_border=[0, 0], base_offset=[0, 0], border=[4, 4], style='flat', batch=None, change_on_press=None*`)`
Subclass of `BaseBorderWidgetLayer()` implementing Button-Style borders.

This class is based on the `ButtonBackground` class. This means that most styles are also available here and should look identical.

Note that this class uses only 20 vertices and is thus not compatible with styles created for use with `BaseBorderWidgetLayer`.

Also note that the `border` argument also only receives two values instead of eight.

`genVertices`()
Called to generate the vertices used by this layer.

The length of the output of this method should be three times the `n_vertices` attribute.

See the source code of this method for more information about the order of the vertices.

`stretchColors`(*c*)
Method that is called to stretch the colors.

Note that this should be implemented by subclasses if plausible and reasonable.

2.11 `peng3d.gui.container` - GUI Container and Scrolling system

class `peng3d.gui.container.`**`Container`**(*name: Optional[str], submenu: SubMenu, window: Any = None, peng: Any = None, *, pos: Union[List[float], Callable[[float, float, float, float], Tuple[float, float]], layout.LayoutCell], size: Union[List[float], Callable[[float, float], Tuple[float, float]], None] = None, _skip_draw=False, font=None, font_size=None, font_color=None, borderstyle=None*)
Main class of the container system.

This widget may contain other widgets, limiting the childs to only draw within the defined bounds. Additionally, the given position will also act as a offset, making the child coordinates relative to the parent.

The `visible` attribute may be set to control whether or not this container is visible.

This Class is a subclass of `peng3d.gui.widgets.Widget` but also exhibits part of the API of `peng3d.gui.SubMenu`.

`addWidget`(*widget*, *order_key=0*)
Adds a widget to this container.

Note that trying to add the Container to itself will be ignored.

`clickable`
Property used for determining if the widget should be clickable by the user.

This is only true if the submenu of this widget is active and this widget is enabled.

The widget may be either disabled by setting this property or the `enabled` attribute.

`draw`()
Draws the submenu and its background.

Note that this leaves the OpenGL state set to 2d drawing and may modify the scissor settings.

`getWidget`(*name*)
Returns the widget with the given name.

`on_enter`(*old*)
Dummy method defined for compatibility with `peng3d.gui.SubMenu`, simply does nothing.

`on_exit`(*new*)
Dummy method defined for compatibility with `peng3d.gui.SubMenu`, simply does nothing.

on_redraw()
Redraws the background and any child widgets.

redraw()
Triggers a redraw of the widget.

Note that the redraw may not be executed instantly, but rather batched together on the next frame. If an instant and synchronous redraw is needed, use `on_redraw()` instead.

setBackground(*bg*)
Sets the background of the Container.

Similar to `peng3d.gui.SubMenu.setBackground()`, but only effects the region covered by the Container.

class peng3d.gui.container.**ScrollableContainer**(*name: Optional[str], submenu: SubMenu, window: Any = None, peng: Any = None, *, pos: Union[List[float], Callable[[float, float, float, float], Tuple[float, float]], layout.LayoutCell], size: Union[List[float], Callable[[float, float], Tuple[float, float]], None] = None, scrollbar_width=12, font=None, font_size=None, font_color=None, borderstyle=None, content_height=100*)
Subclass of `Container` allowing for scrolling its content.

The scrollbar currently is always on the right side and simply consists of a `peng3d.gui.slider.VerticalSlider`.

`scrollbar_width` and `borderstyle` will be passed to the scrollbar.

`content_height` refers to the maximum offset the user can scroll to.

The content height may be changed, but manually calling `redraw()` will be necessary.

on_redraw()
Redraws the background and contents, including scrollbar.

This method will also check the scrollbar for any movement and will be automatically called on movement of the slider.

class peng3d.gui.container.**ContainerButtonBackground**(*widget, border=None, borderstyle='flat', batch=None, change_on_press=None*)
Background class used to render the background of containers using a button style.

Mostly identical with `ButtonBackground` with added compatibility for containers.

getColors()
Overrideable function that generates the colors to be used by various borderstyles.

Should return a 5-tuple of `(bg,o,i,s,h)`.

`bg` is the base color of the background.

`o` is the outer color, it is usually the same as the background color.

`i` is the inner color, it is usually lighter than the background color.

`s` is the shadow color, it is usually quite a bit darker than the background.

`h` is the highlight color, it is usually quite a bit lighter than the background.

2.12 `peng3d.gui.text` - Textual Widgets

class `peng3d.gui.text.`**`Label`**(*name: Optional[str], submenu: SubMenu, window: Any = None, peng: Any = None, *, pos: Union[List[float], Callable[[float, float, float, float], Tuple[float, float]], layout.LayoutCell], size: Union[List[float], Callable[[float, float], Tuple[float, float]], None] = None, bg=None, label='Label', font_size=None, font=None, font_color=None, multiline=False, label_cls=<Mock name='mock.Label' id='139691008166864'>, anchor_x='center', anchor_y='center', label_layer=1*)

Simple widget that can display any single-line non-formatted string.

This widget does not use any background by default.

The default font color is chosen to work on the default background color and may need to be changed if the background color is changed.

`label`

Property for accessing the text of the label.

`on_redraw`(*dt=None*)

Draws the background and the widget itself.

Subclasses should use `super()` to call this method, or rendering may glitch out.

`redraw_label`()

Re-draws the text by calculating its position.

Currently, the text will always be centered on the position of the label.

class `peng3d.gui.text.`**`TextInput`**(*name: Optional[str], submenu: SubMenu, window: Any = None, peng: Any = None, *args, pos: Union[List[float], Callable[[float, float, float, float], Tuple[float, float]], layout.LayoutCell], size: Union[List[float], Callable[[float, float], Tuple[float, float]], None] = None, bg=None, text='', default='', border=[4, 4], borderstyle=None, font_size=None, font=None, font_color=None, font_color_default=[62, 67, 73, 200], allow_overflow=False, allow_copypaste=True, min_size=None, parent_bgcls=None, allow_returnkey=False, **kwargs*)

Basic Textual Input widget.

By default, this widget uses *`TextInputBackground`* as its Background class.

The optional default text will only be displayed if the text is empty.

The `allow_overflow` flag determines if the text entered can be longer than the size of the *`TextInput`*.

The `allow_copypaste` flag controls whether or not the user can copy and paste the contents of the text box. By default, copying and pasting is allowed. This flag can also be set to `"force"` to force a crash with an appropriate error message if the `pyperclip` module is not available. Currently, only copying, pasting and cutting the whole text box is supported, as there is no mechanism for text selection yet.

The key combinations used by this widget can be configured in the config via the `controls.keybinds.common.*` config values.

`parent_bgcls` may be used to override the background used. Note that the cursor will still be rendered. Additional parameters required by the custom background should be passed as keyword arguments. Note that arguments already used by TextInput are not passed down. This may cause issues with ButtonBackground and some other classes.

`allow_returnkey` determines whether pressing the return key inserts a `\r` character or not. Note that the `send_form` action of the submenu may still be sent, even if this is set to true.

`default`
Property for accessing the default text.

`draw``()`
Draws all vertex lists associated with this widget.

`on_redraw``()`
Draws the background and the widget itself.

Subclasses should use `super()` to call this method, or rendering may glitch out.

`redraw_label``()`
Re-draws the label by calculating its position.

Currently, the label will always be centered on the position of the label.

`text`
Property for accessing the text.

`class` `peng3d.gui.text.`**`TextInputBackground`**`(`**args*, ***kwargs*`)`
Background for the *`TextInput`* Widget.

This background uses the button drawing routines and adds a cursor.

`init_bg``()`
Called just before the background will be drawn the first time.

Commonly used to initialize vertex lists.

It is recommended to add all vertex lists to the `submenu.batch2d` Batch to speed up rendering and preventing glitches with grouping.

`pressed`
Read-only helper property to be used by borderstyles for determining if the widget should be rendered as pressed or not.

Note that this property may not represent the actual pressed state, it will always be False if `change_on_press` is disabled.

`redraw_bg``()`
Method called by the parent widget every time its `Widget.redraw()` method is called.

`class` `peng3d.gui.text.`**`CustomTextInputBackground`**`(`*widget*, *cls=<class 'peng3d.gui.button.ButtonBackground'>*, **args*, ***kwargs*`)`
Background for the *`TextInput`* Widget.

This background adds a cursor on top of another background.

`init_bg``()`
Called just before the background will be drawn the first time.

Commonly used to initialize vertex lists.

It is recommended to add all vertex lists to the `submenu.batch2d` Batch to speed up rendering and preventing glitches with grouping.

`redraw_bg``()`
Method called by the parent widget every time its `Widget.redraw()` method is called.

`class` `peng3d.gui.text.`**`PasswordInput`**`(`**args*, *replacement_char='*'*, ***kwargs*`)`

password
Proxy for *text*.

Returns Current password

text
Property for accessing the text.

2.13 `peng3d.gui.slider` - Slider and Progressbar Widgets

class `peng3d.gui.slider.`**Progressbar**(*name: Optional[str], submenu: SubMenu, window: Any = None, peng: Any = None, *, pos: Union[List[float], Callable[[float, float, float, float], Tuple[float, float]], layout.LayoutCell], size: Union[List[float], Callable[[float, float], Tuple[float, float]], None] = None, bg=None, nmin=0, nmax=100, n=0, border=None, borderstyle=None, colors=[[240, 119, 70], [240, 119, 70]]*)

Progressbar displaying a progress of any action to the user.

By default, this Widget uses `ProgressbarBackground` as its Background class.

The border and borderstyle options are the same as for the `peng3d.gui.button.Button` Widget.

The two colors given are for left and right, respectively. This may be used to create gradients.

`nmin`, `nmax` and `n` represent the minimal value, maximal value and current value, respectively. Unexpected behavior may occur if the minimal value is bigger then the maximum value.

n
Property representing the current value of the progressbar.

Changing this property will activate the `progresschange` action.

nmax
Property representing the maximum value of the progressbar. Typically `100` to represent percentages easily.

nmin
Property representing the minimal value of the progressbar. Typically `0`.

value
Alias to the `n` property.

class `peng3d.gui.slider.`**ProgressbarBackground**(*widget*, *border*, *borderstyle*, *colors*)

Background for the `Progressbar` Widget.

This background displays a bar with a border similar to `ButtonBackground`. Note that two colors may be given, one for the left and one for the right.

init_bg()
Called just before the background will be drawn the first time.

Commonly used to initialize vertex lists.

It is recommended to add all vertex lists to the `submenu.batch2d` Batch to speed up rendering and preventing glitches with grouping.

redraw_bg()
Method called by the parent widget every time its `Widget.redraw()` method is called.

class peng3d.gui.slider.**AdvancedProgressbar**(*name: Optional[str], submenu: SubMenu, window: Any = None, peng: Any = None, *, pos: Union[List[float], Callable[[float, float, float, float], Tuple[float, float]], layout.LayoutCell], size: Union[List[float], Callable[[float, float], Tuple[float, float]], None] = None, bg=None, categories=None, offset_nmin=0, offset_nmax=0, offset_n=0, border=None, borderstyle=None, colors=[[240, 119, 70], [240, 119, 70]]*)

Advanced Progressbar displaying the combined progress through multiple actions.

Visually, this widget is identical to `Progressbar` with the only difference being the way the progress percentage is calculated.

The `offset_nmin`, `offset_n` and `offset_nmax` parameters are equivalent to the parameters of the same name minus the `offset_` prefix.

`categories` may be any dictionary mapping category names to 3-tuples of format `(nmin,n,nmax)`.

It is possible to read, write and delete categories through the `widget[cat]` syntax. Note however, that modifying categories in-place, e.g. like `widget[cat][1]=100`, requires a manual call to `redraw()`.

When setting the `nmin`, `n` or `nmax` properties, only an internal offset value will be modified. This may result in otherwise unexpected behavior if setting e.g. `n` to `nmax` because the categories may influence the total percentage calculation.

addCategory(*name*, *nmin=0*, *n=0*, *nmax=100*)

Adds a category with the given name.

If the category already exists, a `KeyError` will be thrown. Use `updateCategory()` instead if you want to update a category.

deleteCategory(*name*)

Deletes the category with the given name.

If the category does not exist, a `KeyError` will be thrown.

n

Property representing the current value of the progressbar.

Changing this property will activate the `progresschange` action.

nmax

Property representing the maximum value of the progressbar. Typically `100` to represent percentages easily.

nmin

Property representing the minimal value of the progressbar. Typically `0`.

updateCategory(*name*, *nmin=None*, *n=None*, *nmax=None*)

Smartly updates the given category.

Only values that are given will be updated, others will be left unchanged.

If the category does not exist, a `KeyError` will be thrown. Use `addCategory()` instead if you want to add a category.

class `peng3d.gui.slider.`**`Slider`**(*name: Optional[str], submenu: SubMenu, window: Any = None, peng: Any = None, *, pos: Union[List[float], Callable[[float, float, float, float], Tuple[float, float]], layout.LayoutCell], size: Union[List[float], Callable[[float, float], Tuple[float, float]], None] = None, bg=None, border=None, borderstyle=None, nmin=0, nmax=100, n=0, handlesize=None*)

Slider that can be used to get a number from the user.

By default, this Widget uses `SliderBackground` as its Background class.

Most options are the same as for `Progressbar`.

`handlesize` simply determines the size of the handle.

Note that scaling this widget on the y-axis will not do much, scale the handlesize instead.

`p`

Helper property containing the percentage this slider is "filled".

This property is read-only.

class `peng3d.gui.slider.`**`SliderBackground`**(*widget, border=None, borderstyle='flat', batch=None, change_on_press=None*)

Background for the `Slider` Widget.

This background displays a button-like handle on top of a bar representing the selectable range.

All given parameters will affect the handle.

`getPosSize`()

Helper function converting the actual widget position and size into a usable and offsetted form.

This function should return a 6-tuple of `(sx,sy,x,y,bx,by)` where sx and sy are the size, x and y the position and bx and by are the border size.

All values should be in pixels and already include all offsets, as they are used directly for generation of vertex data.

This method can also be overridden to limit the background to a specific part of its widget.

`init_bg`()

Called just before the background will be drawn the first time.

Commonly used to initialize vertex lists.

It is recommended to add all vertex lists to the `submenu.batch2d` Batch to speed up rendering and preventing glitches with grouping.

`redraw_bg`()

Method called by the parent widget every time its `Widget.redraw()` method is called.

class `peng3d.gui.slider.`**`VerticalSlider`**(*name: Optional[str], submenu: SubMenu, window: Any = None, peng: Any = None, *, pos: Union[List[float], Callable[[float, float, float, float], Tuple[float, float]], layout.LayoutCell], size: Union[List[float], Callable[[float, float], Tuple[float, float]], None] = None, bg=None, border=None, borderstyle=None, **kwargs*)

Vertical slider that can be used as a scrollbar or getting other input.

By default, this Widget uses `VerticalSliderBackground` as its Background class.

This widget is essentially the same as `Slider`, only vertical.

Note that you may need to flip the x and y values of `size`, `handlesize` and `border` compared to `Slider`.

class `peng3d.gui.slider.`**`VerticalSliderBackground`**(*widget*, *border=None*, *borderstyle='flat'*, *batch=None*, *change_on_press=None*)

Background for the *`VerticalSlider`* Widget.

This background uses the same technique as *`SliderBackground`*, simply turned by 90 Degrees.

`getPosSize`()

Helper function converting the actual widget position and size into a usable and offsetted form.

This function should return a 6-tuple of `(sx,sy,x,y,bx,by)` where sx and sy are the size, x and y the position and bx and by are the border size.

All values should be in pixels and already include all offsets, as they are used directly for generation of vertex data.

This method can also be overridden to limit the background to a specific part of its widget.

2.14 `peng3d.gui.style` - Generic Styles for Widgets

class `peng3d.gui.style.`**`Borderstyle`**

Base class for all border styles.

Each border style class serves a single style, although some parameters may be adjustable.

Currently, border style classes are not instantiated per widget, but rather rely on class methods. This reduces memory overhead but may be changed in the future.

For backwards compatibility, border style classes themselves compare equal with their old-style string equivalents. This is accomplished with a metaclass and not necessary to emulate for new styles and may be removed in a future version.

classmethod get_colormap(*widget: widgets.BasicWidget, bg: List[int], o: List[int], i: List[int], s: List[int], h: List[int], state: Optional[str] = None*)

Gets the color map for a button-style widget.

TODO: document return format

Parameters

- **widget** (`BasicWidget`) – Widget that the color map belongs to
- **bg** (*`ColorRGB`*) – Background color this widget is placed on
- **o** (*`ColorRGB`*) – Outer color, usually same as the background
- **i** (*`ColorRGB`*) – Inner color, usually lighter than the background
- **s** (*`ColorRGB`*) – Shadow color, usually quite a bit darker than the background
- **h** (*`ColorRGB`*) – Highlight color, usually quite a bit lighter than the background
- **state** (*`str`*) – Optional widget state override

Returns

Return type

class `peng3d.gui.style.`**`FlatBorder`**

classmethod get_colormap(*widget: widgets.BasicWidget, bg: List[int], o: List[int], i: List[int], s: List[int], h: List[int], state: Optional[str] = None*)

Gets the color map for a button-style widget.

TODO: document return format

Parameters

- **widget** (BasicWidget) – Widget that the color map belongs to
- **bg** (*ColorRGB*) – Background color this widget is placed on
- **o** (*ColorRGB*) – Outer color, usually same as the background
- **i** (*ColorRGB*) – Inner color, usually lighter than the background
- **s** (*ColorRGB*) – Shadow color, usually quite a bit darker than the background
- **h** (*ColorRGB*) – Highlight color, usually quite a bit lighter than the background
- **state** (*str*) – Optional widget state override

Returns

Return type

class peng3d.gui.style.**GradientBorder**

classmethod get_colormap(*widget: widgets.BasicWidget, bg: List[int], o: List[int], i: List[int], s: List[int], h: List[int], state: Optional[str] = None*)

Gets the color map for a button-style widget.

TODO: document return format

Parameters

- **widget** (BasicWidget) – Widget that the color map belongs to
- **bg** (*ColorRGB*) – Background color this widget is placed on
- **o** (*ColorRGB*) – Outer color, usually same as the background
- **i** (*ColorRGB*) – Inner color, usually lighter than the background
- **s** (*ColorRGB*) – Shadow color, usually quite a bit darker than the background
- **h** (*ColorRGB*) – Highlight color, usually quite a bit lighter than the background
- **state** (*str*) – Optional widget state override

Returns

Return type

class peng3d.gui.style.**OldshadowBorder**

classmethod get_colormap(*widget: widgets.BasicWidget, bg: List[int], o: List[int], i: List[int], s: List[int], h: List[int], state: Optional[str] = None*)

Gets the color map for a button-style widget.

TODO: document return format

Parameters

- **widget** (BasicWidget) – Widget that the color map belongs to
- **bg** (*ColorRGB*) – Background color this widget is placed on
- **o** (*ColorRGB*) – Outer color, usually same as the background
- **i** (*ColorRGB*) – Inner color, usually lighter than the background

- **s** (*ColorRGB*) – Shadow color, usually quite a bit darker than the background
- **h** (*ColorRGB*) – Highlight color, usually quite a bit lighter than the background
- **state** (*str*) – Optional widget state override

Returns

Return type

class peng3d.gui.style.**MaterialBorder**

classmethod get_colormap(*widget: widgets.BasicWidget, bg: List[int], o: List[int], i: List[int], s: List[int], h: List[int], state: Optional[str] = None*)

Gets the color map for a button-style widget.

TODO: document return format

Parameters

- **widget** (BasicWidget) – Widget that the color map belongs to
- **bg** (*ColorRGB*) – Background color this widget is placed on
- **o** (*ColorRGB*) – Outer color, usually same as the background
- **i** (*ColorRGB*) – Inner color, usually lighter than the background
- **s** (*ColorRGB*) – Shadow color, usually quite a bit darker than the background
- **h** (*ColorRGB*) – Highlight color, usually quite a bit lighter than the background
- **state** (*str*) – Optional widget state override

Returns

Return type

peng3d.gui.style.**BORDERSTYLES = {'flat': <class 'peng3d.gui.style.FlatBorder'>, 'gradient**

Map of border style names to classes that implement them.

See the documentation of each class for descriptions.

peng3d.gui.style.**norm_borderstyle**(*borderstyle: Union[str, Type[peng3d.gui.style.Borderstyle]]*) → Type[peng3d.gui.style.Borderstyle]

Normalizes border styles to *Borderstyle* subclasses.

Parameters borderstyle (*Either str or Borderstyle subclass*) – Value to normalize

Returns Normalized value

Return type Borderstyle subclass

Raises TypeError – if an unexpected value was given

class peng3d.gui.style.**Style**(*parent: Union[Style, Dict[str, Union[float, str, Type[Borderstyle]]], None] = None, overrides: Optional[Dict[str, Union[float, str, Type[Borderstyle]]]] = None*)

Core of the hierarchical style system.

This class allows for easily inheriting styles from a parent (e.g. submenu or menu) while allowing dynamic overwriting at any level in the hierarchy. For example, a specific submenu could have a different font that would then be automatically applied to all widgets within it, unless the font was overridden for the widget locally.

When reading a style attribute, this class first checks if it has a local override for that attribute. If so, it will be returned. If the attribute wasn't overridden locally, the parent is queried and its result returned. The root of this hierarchy is the `style` attribute of the *`Peng()`* singleton, which uses the styles defined in *`DEFAULT_STYLE`* as a default. If a style attribute is not found anywhere, a `KeyError` will be raised.

When writing a style attribute, a local override is created. This causes all subsequent accesses to the style attribute within this instance and all children (e.g. widgets within a submenu) to read back the new value. Deleting the style attribute will reset this override and thus reset the read value back to the parent value.

Note that changes in an attribute usually require a redraw of the affected widgets. If a redraw is not performed, weird graphical glitches may happen.

This class is very flexible and allows several different modes of access.

First, it is possible to use it like a dict, e.g. `style["font"]`. It is possible to read, write and delete using this method. All styles are accessible in this manner and arbitrary strings are allowed as keys, though it is recommended to stick to valid Python identifiers.

For convenience, it is also possible to access style attributes as literal attributes of a *`Style`* instance, e.g. `style.font`. Note that this only allows accesses to style attributes whose name is a valid python identifier and that are not in the list of reserved attributes, stored in the class attribute *`Style.ATTRIBUTES`*. It is also not possible to access style attributes that start with an underscore or are methods of *`Style`* this way. This access mode also supports read, write and delete accesses.

Note that unlike the helpers *`default_property`* and *`default`*, *`Style`* does not reset an override if a write with a value of `None` is performed.

`ATTRIBUTES = ['parent', 'ATTRIBUTES']`

Internal list of attributes that are reserved and cannot be used for styles via attribute access.

This list may be extended in the future. Note that all attributes that start with an underscore are also implicitly reserved.

`add_watcher`(*watch_sel*, *callback=None*)

Adds a watcher for specific changes in local styles.

Watchers can be used to automatically update widgets or other visual elements whenever the effective value of a style attribute changes. This includes scenarios where the (not locally overridden) style attribute of the parent changes, causing a change in the effective local value.

The watcher system tries its best to remove unnecessary triggers and double-triggers, but they may still occur under some circumstances. Thus, it is recommended to only use (semi-)idempotent functions as callbacks. A popular example for a suitable callback would be the *redraw()* method of widgets, since it will only queue the actual redraw and thus prevents extraneous redraws.

This method accepts either an instance of `StyleWatcher` or a selector string followed by a callback function.

Selectors are strings that describe what changes to listen to. Currently, selectors are quite rudimentary, but it is planned to add a more sophisticated system later.

The special `*` selector matches all changes and will thus be triggered on any change of any local attribute.

Alternatively, all other strings will trigger on the change of a style attribute with their exact name.

Callback functions can either take no arguments or the old value of the style attribute as a single argument.

`get`(*key: str*, *default: Union[float, str, Type[Borderstyle], None] = None*) → Union[float, str, Type[peng3d.gui.style.Borderstyle], None]

Returns the effective value of the given key or the given default if it couldn't be found.

`is_overridden`(*key: str*) → bool

Checks whether a given key is currently being overridden.

If this returns true, any change in parent styles will not affect the value of the given style attribute.

Parameters **`key`** (*`str`*) – Key to check

Returns whether key is currently overridden

Return type bool

`override_if_not_none`(*key: str, value: Union[float, str, Type[Borderstyle], None]*) → None

Overrides the given key if the provided value is not `None`.

This helper allows for easy style overriding via keyword arguments. Simply create a keyword argument in the constructor of an object that uses styles and set the default of that keyword argument to `None`. In the constructor, you can then call this function like so:

```
self.style.override_if_not_none("font", font)
```

Note that this method is unsuitable for style attributes that may actually want to have a value of `None`.

Parameters

- **`key`** (*`str`*) – Key to override
- **`value`** (*`Optional[StyleValue]`*) – value used to override if it is not `None`

Returns None

Return type None

`parent = None`

Attribute that stores the parent of this style.

May be changed during runtime, though most widgets will require a redraw to fully respect changed effective style values.

It is usually not required to write to this attribute, since widgets do not currently support being moved between different submenus or even menus.

`update`(*_overrides: Optional[Dict[str, Union[float, str, Type[Borderstyle]]]] = None, **kwargs*) → None

Updates several style attributes at the same time.

Note that this method only supports creating and modifying overrides. Keys not present in the given data will be kept as is.

This method supports both passing in a dictionary and passing in keyword attributes. Note that in the case of a style attribute being present in both the dictionary and a keyword argument, the keyword argument takes precedence.

Parameters

- **`_overrides`** (*`Optional[Dict[str, StyleValue]]`*) – Optional dictionary to add/modify overrides from
- **`kwargs`** (*`StyleValue`*) – Optional keyword arguments to add/modify overrides from

Returns None

Return type None

`peng3d.gui.style.`**`DEFAULT_STYLE = {'border': (4, 4), 'borderstyle': <class 'peng3d.gui.sty`**

Default styles for all parts of peng3d.

These styles represent a sensible default for common use cases.

For application-wide changes, it is recommended to override the styles in question using the `peng3d.peng.Peng.style` attribute.

2.15 `peng3d.resource` - Resource loading system

class `peng3d.resource.`**`ResourceManager`**(*peng: peng3d.Peng*, *basepath: str*)

Manager that allows for efficient and simple loading and management of different kinds of resources.

Currently supports textures and models out of the box, but extension is possible.

Textures can be queried by any part of the application, they are only loaded on the first request and then cached for every request following it.

The same caching and lazy-loading principle applies to models loaded via this system.

`addCategory`(*name: str*, *size: Optional[int] = None*) → int

Adds a new texture category with the given name.

If the category already exists, it will be overridden.

`addFromTex`(*name: str*, *img: <Mock name='mock.AbstractImage' id='139691013134096'>*, *category: str*) → Tuple[int, int, Tuple]

Adds a new texture from the given image.

`img` may be any object that supports Pyglet-style copying in form of the `blit_to_texture()` method.

This can be used to add textures that come from non-file sources, e.g. Render-to-texture.

`getMissingTexture`() → <Mock name='mock.AbstractImage' id='139691013134096'>

Returns a texture to be used as a placeholder for missing textures.

A default missing texture file is provided in the assets folder of the source distribution. It consists of a simple checkerboard pattern of purple and black, this image may be copied to any project using peng3d for similar behavior.

If this texture cannot be found, a pattern is created in-memory, simply a solid square of purple.

This texture will also be cached separately from other textures.

`getModel`(*name: str*) → peng3d.model.Model

Gets the model object by the given name.

If it was loaded previously, a cached version will be returned. If it was not loaded, it will be loaded and inserted into the cache.

`getModelData`(*name: str*) → Dict[KT, VT]

Gets the model data associated with the given name.

If it was loaded, a cached copy will be returned. It it was not loaded, it will be loaded and cached.

`getTex`(*name: str*, *category: str*) → Tuple[int, int, Tuple]

Gets the texture associated with the given name and category.

`category` must have been created using *`addCategory()`* before.

If it was loaded previously, a cached version will be returned. If it was not loaded, it will be loaded and inserted into the cache.

See *`loadTex()`* for more information.

`loadModel`(*name: str*) → peng3d.model.Model

Loads the model of the given name.

The model will also be inserted into the cache.

`loadModelData`(*name: str*) → Dict[KT, VT]

Loads the model data of the given name.

The model file must always be a .json file.

loadTex(*name: str*, *category: str*) → Tuple[int, int, Tuple]
Loads the texture of the given name and category.

All textures currently must be PNG files, although support for more formats may be added soon.

If the texture cannot be found, a missing texture will instead be returned. See `getMissingTexture()` for more information.

Currently, all texture mipmaps will be generated and the filters will be set to `GL_NEAREST` for the magnification filter and `GL_NEAREST_MIPMAP_LINEAR` for the minification filter. This results in a pixelated texture and not a blurry one.

resourceExists(*name: str*, *ext: str = "*) → bool
Returns whether or not the resource with the given name and extension exists.

This must not mean that the resource is meaningful, it simply signals that the file exists.

resourceNameToPath(*name: str*, *ext: str = "*) → str
Converts the given resource name to a file path.

A resource path is of the format `<app>:<cat1>.<cat2>.<name>` where cat1 and cat2 can be repeated as often as desired.

`ext` is the file extension to use, e.g. `.png` or similar.

As an example, the resource name `peng3d:some.category.foo` with the extension `.png` results in the path `<basepath>/assets/peng3d/some/category/foo.png`.

This resource naming scheme is used by most other methods of this class.

Note that it is currently not possible to define multiple base paths to search through.

2.16 `peng3d.i18n` - Lightweight Translation Manager

class peng3d.i18n.**TranslationManager**(*peng: peng3d.Peng*)
Manages sets of translation files in multiple languages.

This Translation System uses language codes to identify languages, there is no requirement to follow a specific standard, but it is recommended to use simple 2-digit codes like `en` and `de`, adding an underscore to define sub-languages like `en_gb` and `en_us`.

Whenever a new translation file is needed, it will be parsed and then cached. This speeds up access times and also practically eliminates load times when switching languages.

Several events are sent by this class, see *peng3d:i18n.* Events Category*.

Most of these events are also sent as actions, these actions are described in the methods that cause them.

There are also severale config options that determine the behaviour of this class. See *Translation Options* for more information.

This Manager requires the `ResourceManager()` to be already initialized.

discoverLangs(*domain: str = '*'*) → List[str]
Generates a list of languages based on files found on disk.

The optional `domain` argument may specify a domain to use when checking for files. By default, all domains are checked.

This internally uses the `glob` built-in module and the `i18n.lang.format` config option to find suitable filenames. It then applies the regex in `i18n.discover_regex` to extract the language code.

loadDomain(*domain: str*, *lang: Optional[str] = None*, *encoding: str = 'utf-8'*) → bool
Loads the translation data of a single domain for a specific language from disk into the cache.

If no language was given, the current language is used.

If the translation file could not be found or any errors occur while reading it, these errors will be silently discarded, only recognizable by a return value of `False`.

If the load was successful, the action `loaddomain` will be executed and this method will return `True`.

setLang(*lang: str*) → None
Sets the default language for all domains.

For recommendations regarding the format of the language code, see *`TranslationManager`*.

Note that the `lang` parameter of both *`translate()`* and *`translate_lazy()`* will override this setting.

Also note that the code won't be checked for existence or plausibility. This may cause the fallback strings to be displayed instead if the language does not exist.

Calling this method will cause the `setlang` action and the :peng3d:event'peng3d:i18n.set_lang' event to be triggered. Note that both action and event will be triggered even if the language did not actually change.

This method also automatically updates the *`i18n.lang`* config value.

t(*key: str*, *translate: bool = True*, *lang: Optional[str] = None*) → str
Translates the given key.

If no language was given, the language last passed to *`setLang()`* will be used.

If the translation key could not be found (e.g. because the language code is invalid), the key itself will be returned.

Note that this method returns a string and thus does not have any way to modify the returned value if the language is changed by the user. If dynamic translation is required, *`translate_lazy()`* should be used instead.

tl(*key: str*, *data: Optional[Dict[KT*, *VT]] = None*, *translate: bool = True*, *lang: Optional[str] = None*)
→ peng3d.i18n._LazyTranslator
Lazily translates a given translation key.

This method is similar to *`translate()`*, but returns a special object rather than a string. This allows for on-the-fly changing of the language without having to re-set all the places where translated strings are used.

Whenever the returned object is converted to a string by `str()` or `repr()` or is formatted using either the old `%`-notation or the newer `str.format()`, the translation key will be looked up again, in case the language has changed.

Note that this requires support from the widgets (or other consumers of the returned value), namely that they only convert to string just prior to rendering and re-render either regularly or whenever either the `setlang` action or the `peng3d:i18n.set_lang` event is called.

Most built-in widgets support this, but some special cases are not supported yet. For example, setting the window title dynamically requires using the `caption_t` parameter instead of the raw `caption` parameter.

translate(*key: str*, *translate: bool = True*, *lang: Optional[str] = None*) → str
Translates the given key.

If no language was given, the language last passed to *`setLang()`* will be used.

If the translation key could not be found (e.g. because the language code is invalid), the key itself will be returned.

Note that this method returns a string and thus does not have any way to modify the returned value if the language is changed by the user. If dynamic translation is required, `translate_lazy()` should be used instead.

`translate_lazy`(*key: str*, *data: Optional[Dict[KT, VT]] = None*, *translate: bool = True*, *lang: Optional[str] = None*) → peng3d.i18n._LazyTranslator

Lazily translates a given translation key.

This method is similar to `translate()`, but returns a special object rather than a string. This allows for on-the-fly changing of the language without having to re-set all the places where translated strings are used.

Whenever the returned object is converted to a string by `str()` or `repr()` or is formatted using either the old `%`-notation or the newer `str.format()`, the translation key will be looked up again, in case the language has changed.

Note that this requires support from the widgets (or other consumers of the returned value), namely that they only convert to string just prior to rendering and re-render either regularly or whenever either the `setlang` action or the `peng3d:i18n.set_lang` event is called.

Most built-in widgets support this, but some special cases are not supported yet. For example, setting the window title dynamically requires using the `caption_t` parameter instead of the raw `caption` parameter.

2.17 `peng3d.model` - Model and Animation system

`peng3d.model.`**`grouper`**(*iterable*, *n*, *fillvalue=None*)

Allows for iteration over multiple elements of a iterable at once.

`iterable` may be any iterable, its values will be returned. Note that this may be iterated over more than once.

`n` is the size of each group. May be any positive integer.

`fillvalue` is optionally used to fill any groups that do not have enough items, for example if the length of the iterable is not divisible by n.

Example:

```
>>> for i in grouper("foobarbaz",2,fillvalue=" "):
...     print(i)
fo
ob
ar
ba
z  # Note the extra space after the z
```

`peng3d.model.`**`calcSphereCoordinates`**(*pos*, *radius*, *rot*)

Calculates the Cartesian coordinates from spherical coordinates.

`pos` is a simple offset to offset the result with.

`radius` is the radius of the input.

`rot` is a 2-tuple of `(azimuth,polar)` angles.

Angles are given in degrees. Most directions in this game use the same convention.

The azimuth ranges from 0 to 360 degrees with 0 degrees pointing directly to the x-axis.

The polar angle ranges from -90 to 90 with -90 degrees pointing straight down and 90 degrees straight up.

A visualization of the angles required is given in the source code of this function.

`peng3d.model.`**`v_magnitude`**`(`*v*`)`

Simple vector helper function returning the length of a vector.

`v` may be any vector, with any number of dimensions

`peng3d.model.`**`v_normalize`**`(`*v*`)`

Normalizes the given vector.

The vector given may have any number of dimensions.

`class` `peng3d.model.`**`Material`**`(`*rsrcMgr*, *name*, *matdata*`)`

Object that describes a single material of a model.

This object stores all relevant data and caches. Note that this object is only created once for each model and shared between all rendered instances of it.

See *`Model`* for more information about the model system.

`id`

Read-only property storing the numerical ID of the texture of this material.

Used to manipulate the texture behind this material.

Commonly used in binding the texture: `glBindTexture(material.target,material.id)`.

`target`

Read-only property storing the OpenGL constant representing the target of the texture of this material.

Commonly `GL_TEXTURE_2D` or `GL_TEXTURE_3D`.

Used in texture manipulation and activation, e.g. `glEnable(material.target)`.

`tex_coords`

Read-only property storing the texture coordinates to use when drawing with this texture.

Should not be used directly, see *`transformTexCoords()`*.

Enables substitution of pyglet `pyglet.graphics.Texture` objects with Materials in many places, e.g. in `pyglet.graphics.TextureGroup`.

`texdata`

Read-only property equivalent to a 3-tuple containing *`target`*, *`id`* and *`tex_coords`*.

Should be faster than getting each value directly. Useful if all of these values are needed.

`transformTexCoords``(`*data*, *texcoords*, *dims=2*`)`

Transforms the given texture coordinates using the internal texture coordinates.

Currently, the dimensionality of the input texture coordinates must always be 2 and the output is 3-dimensional with the last coordinate always being zero.

The given texture coordinates are fitted to the internal texture coordinates. Note that values higher than 1 or lower than 0 may result in unexpected visual glitches.

The length of the given texture coordinates should be divisible by the dimensionality.

`class` `peng3d.model.`**`Bone`**`(`*rsrcMgr*, *name*, *bonedata*`)`

Object that represents a single bone of a model.

This object stores all relevant data and caches. Note that this object is only created once for each model and shared between all rendered instances of it.

Actual bone rotation and length is stored per entity and not per model allowing for different bone rotations for multiple entities using the same model.

See `Model` for more information about the model system.

`addRegion`(*region*)
Register a vertex Region as a dependent of this bone.

`region` must be an instance of `Region`.

`ensureBones`(*data*)
Helper method ensuring per-entity bone data has been properly initialized.

Should be called at the start of every method accessing per-entity data.

`data` is the entity to check in dictionary form.

`getLength`(*data*)
Returns the length of this bone in the given entity.

`data` is the entity to query in dictionary form.

`getPivotPoint`(*data*)
Returns the point this bone pivots around on the given entity.

This method works recursively by calling its parent and then adding its own offset.

The resulting coordinate is relative to the entity, not the world.

`getRot`(*data*)
Returns the rotation of this bone in the given entity.

`data` is the entity to query in dictionary form.

`setLength`(*data*, *blength*)
Sets the length of this bone on the given entity.

`data` is the entity to modify in dictionary form.

`blength` is the new length of the bone.

`setParent`(*parent*)
Sets the parent of this bone for all entities.

Note that this method must be called before many other methods to ensure internal state has been initialized.

This method also registers this bone as a child of its parent.

`setRot`(*data*, *rot*)
Sets the rotation of this bone on the given entity.

`data` is the entity to modify in dictionary form.

`rot` is the rotation of the bone in the format used in `calcSphereCoordinates()`.

`setRotate`(*data*)
Sets the OpenGL state required for proper drawing of the model.

Mostly rotates and translates the camera.

It is important to call `unsetRotate()` after calling this method to properly unset state and avoid OpenGL errors.

`transformVertices`(*data*, *vertices*, *dims=3*)
Currently unused method that transforms the given vertices according to the rotation of the bone.

Currently just returns the vertices unmodified, will be implemented in the future.

unsetRotate(*data*)

Unsets the OpenGL state that was set before calling `setRotate()`.

Note that this method may cause various OpenGL errors if called without `setRotate()` having been called.

class `peng3d.model.`**RootBone**(*rsrcMgr*, *name*, *bonedata*)

Special bone that represents the root of a entity.

This bone is immutable and cannot be rotated or otherwise modified.

getLength(*data*)

Returns the length of this bone in the given entity.

`data` is the entity to query in dictionary form.

getPivotPoint(*data*)

Returns the point this bone pivots around on the given entity.

This method works recursively by calling its parent and then adding its own offset.

The resulting coordinate is relative to the entity, not the world.

setRotate(*data*)

Sets the OpenGL state required for proper drawing of the model.

Mostly rotates and translates the camera.

It is important to call `unsetRotate()` after calling this method to properly unset state and avoid OpenGL errors.

unsetRotate(*data*)

Unsets the OpenGL state that was set before calling `setRotate()`.

Note that this method may cause various OpenGL errors if called without `setRotate()` having been called.

class `peng3d.model.`**Region**(*rsrcMgr*, *name*, *regdata*)

Object that represents a vertex region of a model.

A vertex region is associated with a specific bone of the same model it is associated with. It has a list of vertices and optionally texture coordinates. The texture coordinates are transformed using the material it is associated with.

Most regions will use quads as their primitive type, but it is also possible to use triangles, lines and points.

To use quads as the geometry type, specify either `quads`, `quad` or `GL_QUADS` as its `geometry_type`.

To use triangles as the geometry type, specify either `tris`, `triangles`, `triangle` or `GL_TRIANGLES` as its `geometry_type`.

To use lines as the geometry type, specify either `lines`, `line` or `GL_LINES` as its `geometry_type`.

To use points as the geometry type, specify either `points`, `point`, `dots`, `dot` or `GL_POINTS` as its `geometry_type`.

Note that the number of vertices must be divisible by the number of vertices required per primitive, e.g. 4 for quads, 3 for triangles, 2 for lines and 1 for points.

Additionally, the number of vertices and texture coordinate pairs must also match.

If any of these conditions are not fulfilled, a `ValueError` will be raised.

getGeometryType(*data*)

Returns the OpenGL constant representing the type of primitives used by this region.

May be one of `GL_QUADS`, `GL_TRIANGLES`, `GL_LINES` or `GL_POINTS`.

`getTexCoords`(*data*)
Returns the texture coordinates, if any, to accompany the vertices of this region already transformed.

Note that it is recommended to check the `enable_tex` flag first.

Internally uses *`Material.transformTexCoords()`*.

`getTexInfo`(*data*)
Returns informations about the texture of this region.

Internally uses *`Material.texdata`*, exact specification available there.

`getVertices`(*data*)
Returns the vertices of this region already transformed and ready-to-use.

Internally uses *`Bone.transformVertices()`*.

`class` `peng3d.model.`**`Animation`**(*rsrcMgr*, *name*, *anidata*)
Object that represents an animation of a model.

Animations can be either static or animated using keyframes.

See *`Model`* for more information.

`setBones`(*bones*)
Sets the internal dictionary of bones in the parent model.

Must be a dictionary, else errors may appear later on.

`startAnimation`(*data*, *jumptype*)
Callback that is called to initialize this animation on a specific actor.

Internally sets the `_anidata` key of the given dict `data`.

`jumptype` is either `jump` or `animate` to define how to switch to this animation.

`tickEntity`(*data*)
Callback that should be called regularly to update the animation.

It is recommended to call this method about 60 times a second for smooth animations. Irregular calling of this method will be automatically adjusted.

This method sets all the bones in the given actor to the next state of the animation.

Note that *`startAnimation()`* must have been called before calling this method.

`class` `peng3d.model.`**`JSONModelGroup`**(*model*, *data*, *obj*, *parent=None*)
Pyglet group that sets the state required by a specific actor.

This group should always be set during any draw operations for the assigned actor. This can either be done by setting it as the group of a vertex list, the parent group of a group of a vertex list or manually calling *`set_state()`* and *`unset_state()`*.

`set_state`()
Sets the state required for this actor.

Currently translates the matrix to the position of the actor.

`unset_state`()
Resets the state required for this actor to the default state.

Currently resets the matrix to its previous translation.

class `peng3d.model.`**`JSONRegionGroup`**(*model*, *data*, *region*, *parent=None*)

Pyglet group that manages the state required by a specific vertex region of an actor.

This group and the associated *`JSONModelGroup`* should always be set during any draw operation for the assigned region.

See *`JSONModelGroup`* for more information about how to do this.

`set_state`()

Sets the state required for this vertex region.

Currently binds and enables the texture of the material of the region.

`unset_state`()

Resets the state required for this actor to the default state.

Currently only disables the target of the texture of the material, it may still be bound.

class `peng3d.model.`**`Model`**(*peng*, *rsrcMgr*, *name*)

Object that represents the model of an actor.

Note that this object is not bound to an actor but rather to a collection of materials, bones, vertex regions and animations.

A single instance of this class may be used by multiple actors at the same time. See `Actor.setModel()` for more information.

A test model is available at `assets/peng3d/model/test.json` and a demo program using it under `test_model.py`.

Todo: Document the format of .json model files.

`cleanup`(*obj*)

Cleans up any left over data structures, including vertex lists that reside in GPU memory.

Behaviour is undefined if it is attempted to use this model with the same object without calling *`create()`* first.

It is very important to call this method manually during deletion as this will delete references to data objects stored in global variables of third-party modules.

`create`(*obj*, *cache=False*)

Initializes per-actor data on the given object for this model.

If `cache` is set to True, the entity will not be redrawn after initialization.

Note that this method may set several attributes on the given object, most of them starting with underscores.

During initialization of vertex regions, several vertex lists will be created. If the given object has an attribute called `batch3d` it will be used, else it will be created.

If the batch already existed, the *`draw()`* method will do nothing, else it will draw the batch.

Memory leaks may occur if this is called more than once on the same object without calling *`cleanup()`* first.

`draw`(*obj*)

Actually draws the model of the given object to the render target.

Note that if the batch used for this object already existed, drawing will be skipped as the batch should be drawn by the owner of it.

ensureModelData(*obj*)
Ensures that the given `obj` has been initialized to be used with this model.

If the object is found to not be initialized, it will be initialized.

redraw(*obj*)
Redraws the model of the given object.

Note that currently this method probably won't change any data since all movement and animation is done through pyglet groups.

remove(*obj*)
Called if the actor is removed from the world.

Can be extended for custom features, currently calls *`cleanup()`*.

setAnimation(*obj*, *animation*, *transition=None*, *force=False*)
Sets the animation to be used by the object.

See `Actor.setAnimation()` for more information.

2.18 `peng3d.camera` - Camera System

class peng3d.camera.**Camera**(*world*, *name*, *pos=None*, *rot=None*)
Camera object representing a location to draw from.

Each *`Camera`* object is bound to a world and has three properties: a name, *`pos`* and *`rot`*.

The name of the camera can be any string and is used to identify the camera and thus should be unique.

on_activate(*old*)
Fake event handler called when this camera is made current by a `WorldView()` object.

on_move(*old*, *new*)
Fake event handler called when this camera moves.

The `old` and `new` parameters are both 3D Locations and are not equal. Each parameter is a 3-tuple of `(x,y,z)` in world coordinates.

on_rotate(*old*, *new*)
Fake event handler called when this camera is rotated.

The `old` and `new` parameters are both rotations and are not equal. Each parameter is a 2-tuple of `(yaw, pitch)`.

pos
Property for accessing the position of the camera.

This property uses a setter to call the *`on_move()`* method if set and the new location is not equal to the old location.

rot
Property for accessing the rotation of the camera.

This property uses a setter to call the *`on_rotate()`* method if set and the new location is not equal to the old location.

class peng3d.camera.**CameraActorFollower**(*world*, *name*, *actor*)
Special Camera that follows the specified `Actor()`.

Note that neither the *`on_move()`* nor the *`on_rotate()`* event handlers are called due to the way the updating works.

pos
This property always equals the value of `self.actor.pos`.

This property may also be written to.

rot
This property always equals the value of `self.actor.rot`.

This property may also be written to.

2.19 `peng3d.world` - World, Terrain and Actor management

class `peng3d.world.`**`World`**`(`*peng*`)`
World containing terrain, actors, cameras and views.

See the docs about `Camera()`, *`WorldView()`*, `Actor()` for more information about each class.

This class does not draw anything, see *`StaticWorld()`* for drawing simple terrain.

`addActor``(`*actor*`)`
Adds the given actor to the internal registry.

Note that this actors `uuid` attribute must be unique, else it will override any actors previously registered with its UUID.

`addCamera``(`*camera*`)`
Add the camera to the internal registry.

Each camera name must be unique, or else only the most recent version will be used. This behavior should not be relied on because some objects may cache objects.

Additionally, only instances of *`Camera()`* may be used, everything else raises a `TypeError`.

`addView``(`*view*`)`
Adds the supplied *`WorldView()`* object to the internal registry.

The same restrictions as for cameras apply, e.g. no duplicate names.

Additionally, only instances of *`WorldView()`* may be used, everything else raises a `TypeError`.

`getView``(`*name*`)`
Returns the view with name `name`.

Raises a `ValueError` if the view does not exist.

`render3d``(`*view=None*`)`
Renders the world in 3d-mode.

If you want to render custom terrain, you may override this method. Be careful that you still call the original method or else actors may not be rendered.

class `peng3d.world.`**`StaticWorld`**`(`*peng*, *quads*, *colors*`)`
Subclass of *`StaticWorld()`*, allows for semi-static terrain to be rendered.

This class is not suitable for highly complex or user-modifiable terrain.

`quads` is a list of 3d vertices, e.g. a single quad may be `[-1,-1,-1, 1,-1,-1, 1,-1,1, -1,-1,1]`, which represents a rectangle of size 2x2 centered around 0,0. It should also be noted that all quads have to be in a single list.

`colors` is a list of RGB Colors in a similar format to `quads` but with colors instead. Note that there must be a color for every vertex in the vertex list. Every color is an integer between 0 and 255 using the internal pyglet scheme `c3B/static`.

You can modify the terrain via the `terrain` attribute, note that it is a pyglet vertex list, and not a python list.

render3d(*view=None*)
Renders the world.

class peng3d.world.**WorldView**(*world*, *name*, *cam*)
Object representing a view on the world.

A `WorldView()` object references a camera and has a name.

`cam` is a valid camera name known to the world object supplied.

cam
Property for getting the currently active camera.

Always equals `self.cameras[self.activeCamera]`.

on_menu_enter(*old*)
Fake event handler called by `Layer.on_menu_enter()` when the containing menu is entered.

on_menu_exit(*new*)
Fake event handler, same as `on_menu_enter()` but for exiting menus instead.

pos
Property for accessing the current position of the active camera.

The value of this property will always be a list of length 3.

This property can also be written to.

rot
Property for accessing the current rotation of the active camera.

This property can also be written to.

setActiveCamera(*name*)
Sets the active camera.

This method also calls the `Camera.on_activate()` event handler if the camera is not already active.

class peng3d.world.**WorldViewMouseRotatable**(*world*, *name*, *cam*)
Subclass of `WorldView()` that is rotatable using the user.

Moving the mouse cursor left or right will rotate the attached camera horizontally and moving the mouse cursor up or down will rotate the camera vertically.

By default, each pixel traveled changes the angle in degrees by 0.15, though this can be changed via the `controls.mouse.sensitivity` config value.

on_key_press(*symbol*, *modifiers*)
Keyboard event handler handling only the escape key.

If an escape key press is detected, mouse exclusivity is toggled via `PengWindow.toggle_exclusivity()`.

on_menu_enter(*old*)
Fake event handler, same as `WorldView.on_menu_enter()` but forces mouse exclusivity.

on_menu_exit(*new*)
Fake event handler, same as `WorldView.on_menu_exit()` but force-disables mouse exclusivity.

on_mouse_drag(*x*, *y*, *dx*, *dy*, *buttons*, *modifiers*)
Handler used to still enable mouse movement while a button is pressed.

on_mouse_motion(*x*, *y*, *dx*, *dy*)

Handles mouse motion and rotates the attached camera accordingly.

For more information about how to customize mouse movement, see the class documentation here `WorldViewMouseRotatable()`.

2.20 `peng3d.actor` - Extendable Actor System

class peng3d.actor.**Actor**(*peng, world, uuid=None, pos=[0, 0, 0]*)

Actor object, base class for all other Actors in the world.

An actor represents an object in the world, for example the player, an animal, enemy or dropped item.

Everything that is not part of the terrain should be an actor.

The default actor does not do anything, you should look at the subclasses for more information.

addController(*controller*)

Adds a controller to the actor.

A controller can control its actor and can act as a bridge between actor and user inputs.

Controllers may be added anytime during the lifetime of an actor.

on_move(*old*)

Fake event handler called if the location of this actor changes.

This handler is called after the location has changed.

Parameters **old** (`list`) – The previous position

pos

Property allowing access to the position of this actor.

This actor is read-write but calls `on_move()` if it is set.

render(*view=None*)

Called by `World.render3d()` to render this actor.

By default, this method calls the draw method of its model, if any.

For custom render behavior, it is recommended to extend this method or modify the model.

setAnimation(*animation*, *transition=None*, *force=False*)

Sets the animation the model of this actor should show.

`animation` is the name of the animation to switch to.

`transition` can be used to override the transition between the animations.

`force` can be used to force reset the animation even if it is already running.

If there is no model set for this actor, a `RuntimeError` will be raised.

setModel(*model*)

Sets the model this actor should use when drawing.

This method also automatically initializes the new model and removes the old, if any.

class peng3d.actor.**RotatableActor**(*peng, world, uuid=None, pos=[0, 0, 0], rot=[0, 0]*)

Actor that can also be rotated.

This subclass adds a rotational value to the actor and a method to move the actor along the current rotation.

move (*dist*)

Moves the actor using standard trigonometry along the current rotational vector.

Parameters **dist** (*float*) – Distance to move

Todo: Test this method, also with negative distances

on_rotate (*old*)

Fake event handler called if the rotation of this actor changes.

This handler is called after the rotation has been made.

Parameters **old** (*tuple*) – Old rotation before rotating

rot

Property for accessing the rotation of this actor.

Rotation is a tuple of `(x,y)` where y is clamped to -90 and 90. x rolls over at 360, resulting in a seamless experience for players.

This property may also be written to, this calls `on_rotate()`.

`peng3d.actor.`**RotateableActor**

alias of `peng3d.actor.RotatableActor`

class `peng3d.actor.`**Controller** (*actor*)

Base class for all controllers.

Controllers define behavior of Actors and can be used to control them via e.g. the keyboard or an AI.

Every controller is bound to its actor and can be enabled and disabled individually. You may also deactivate all controllers of an Actor by setting the `enabled` key of `Actor.controlleroptions` to False.

enabled

Property allowing to get and set if this controller should be active.

When getting this property, the result of ANDing the internal flag and the actor flag is returned.

When setting, only the local internal flag is set, allowing other controllers to still work.

Raises **AssertionError** – when the supplied value is not of type bool

registerEventHandlers ()

Method to be overridden by subclasses for registering event handlers.

Automatically called upon object creation.

2.21 `peng3d.actor.player` - Player Actors

class `peng3d.actor.player.`**BasicPlayer** (*peng, world, uuid=None, pos=[0, 0, 0], rot=[0, 0]*)

Basic Player class, subclass of `RotatableActor()`.

This class adds no features currently, it can be used to identify player actors via `isinstance()`.

class `peng3d.actor.player.`**FirstPersonPlayer** (*peng, world, uuid=None, pos=[0, 0, 0], rot=[0, 0]*)

Old class allowing to create standard first-person players easily.

Deprecated See `EgoMouseRotationalController()` and `FourDirectionalMoveController()` instead

get_motion_vector()
Returns the movement vector according to held buttons and the rotation.

Returns 3-Tuple of `(dx,dy,dz)`

Return type tuple

update(*dt*)
Internal method used for moving the player.

Parameters **dt** (*float*) – Time delta since the last call to this method

class `peng3d.actor.player.`**FourDirectionalMoveController**(**args*, ***kwargs*)
Controller allowing the user to control the actor with the keyboard.

You can configure the used keybinds with the *controls.controls.forward* etc. The keybinds can also be changed with their `keybindname`, e.g. `peng3d:actor.<actor uuid>.player.controls.forward` for forward.

The movement speed may also be changed via the `movespeed` instance attribute, which defaults to *controls.controls.movespeed*.

You may also access the currently held keys via `move`, which is a list with 2 items, forwards/backwards and left/right.

get_motion_vector()
Returns the movement vector according to held buttons and the rotation.

Returns 3-Tuple of `(dx,dy,dz)`

Return type tuple

registerEventHandlers()
Registers needed keybinds and schedules the *update()* Method.

You can control what keybinds are used via the *controls.controls.forward* etc. Configuration Values.

update(*dt*)
Should be called regularly to move the actor.

This method does nothing if the `enabled` property is set to false.

Note that this method is called automatically and should not be manually called.

class `peng3d.actor.player.`**EgoMouseRotationalController**(**args*, ***kwargs*)
Controller allowing the user to rotate the actor with the mouse.

registerEventHandlers()
Registers the motion and drag handlers.

Note that because of the way pyglet treats mouse dragging, there is also an handler registered to the on_mouse_drag event.

class `peng3d.actor.player.`**BasicFlightController**(**args*, ***kwargs*)
Controller allowing the user to move up and down with the jump and crouch controls.

The used keybinds may be configured via *controls.controls.crouch* and *controls.controls.jump*.

The vertical speed used when flying may be configured via *controls.controls.verticalspeed* or the `speed` attribute.

registerEventHandlers()
Registers the up and down handlers.

Also registers a scheduled function every 60th of a second, causing pyglet to redraw your window with 60fps.

update(*dt*)
Should be called regularly to move the actor.

This method does nothing if the `enabled` property is set to False.

This method is called automatically and should not be called manually.

2.22 `peng3d.keybind` - Dynamic Keybinding System

class `peng3d.keybind.`**KeybindHandler**(*peng: peng3d.Peng*)
Handler class that automatically converts incoming key events to key combo events.

A keybinding always is of format `[MOD1-[MOD2-]]KEY` with potentially more modifiers.

See *MODNAME2MODIFIER* for more information about existing modifiers.

Note that the order in which modifiers are listed also is the order of the above listing.

Keybindings are matched exactly, and optionally a second time without the modifiers listed in *OPTIONAL_MODNAMES* if *controls.keybinds.strict* is set to False.

add(*keybind: str, kbname: str, handler: Callable[[int, int, bool], Any], mod: bool = True*) → None
Adds a keybind to the internal registry.

Keybind names should be of the format `namespace:category.subcategory.name`e.g. `peng3d:actor.player.controls.forward` for the forward key combo for the player actor.

Parameters

- **keybind** (*str*) – Keybind string, as described above
- **kbname** (*str*) – Name of the keybind, may be used to later change the keybinding without re-registering
- **handler** (*function*) – Function or any other callable called with the positional arguments `(symbol,modifiers,release)` if the keybind is pressed or released
- **mod** (*int*) – If the keybind should respect modifiers

changeKeybind(*kbname: str, combo: str*) → None
Changes a keybind of a specific keybindname.

Parameters

- **kbname** (*str*) – Same as kbname of *add()*
- **combo** (*str*) – New key combination

handle_combo(*combo: str, symbol: int, modifiers: int, release: bool = False, mod: bool = True*) → None
Handles a key combination and dispatches associated events.

First, all keybind handlers registered via *add()* will be handled, then the pyglet event `on_key_combo` with params `(combo,symbol,modifiers,release,mod)` is sent to the `Peng()` instance.

Also sends the events *peng3d:keybind.combo*, *peng3d:keybind.combo.press* and :peng3d:event‘peng3d:keybind.combo.release‘.

Params str combo Key combination pressed

Params int symbol Key pressed, passed from the same argument within pyglet

Params int modifiers Modifiers held while the key was pressed

Params bool release If the combo was released

Params bool mod If the combo was sent without mods

`mod_is_held`(*modname: str*, *modifiers: int*) → bool

Helper method to simplify checking if a modifier is held.

Parameters

- **`modname`** (*`str`*) – Name of the modifier, see *`MODNAME2MODIFIER`*
- **`modifiers`** (*`int`*) – Bitmask to check in, same as the modifiers argument of the on_key_press etc. handlers

`peng3d.keybind.`**`KeybindHandlerFunc = typing.Callable[[int, int, bool], typing.Any]`**

Custom type for a keybind handler function.

See *`KeybindHandler.add()`* for more details regarding the signature.

`peng3d.keybind.`**`MODNAME2MODIFIER`**

Ordered Bidict that maps between user-friendly names and internal constants.

Note that since this is a bidict, you can query the reverse mapping by accessing `MODNAME2MODIFIER.inv`. The non-inverse mapping maps from user-friendly name to internal constant.

This mapping is used by the Keybind system to convert the modifier constants to names.

The Mapping is as follows:

Name	Pyglet constant	Notes
ctrl	`key.MOD_ACCEL`	
alt	`key.MOD_ALT`	1
shift	`key.MOD_SHIFT`	
option	`key.MOD_OPTION`	
capslock	`key.MOD_CAPSLOCK`	
numlock	`key.MOD_NUMLOCK`	
scrollock	`key.MOD_SCROLLOCK`	

1: automatically replaced by MOD_CTRL on Darwin/OSX

`peng3d.keybind.`**`MOD_RELEASE = 32768`**

Fake modifier applied when a key is released instead of pressed.

This modifier internally has the value of `1<<15` and should thus be safe from any added modifiers in the future.

Note that this modifier is only applied within keybinds, not in regular `on_key_down` and `on_key_up` handlers.

`peng3d.keybind.`**`OPTIONAL_MODNAMES = ['capslock', 'numlock', 'scrollock']`**

List of modifiers that are not substantial to a key combo.

If the *`controls.keybinds.strict`* option is disabled, every key combo is emitted with and without the modifiers in this list. Else, only the combo with these modifiers is emitted.

This may cause no more combos to get through if numlock or capslock are activated.

2.23 `peng3d.config` - Configuration system

peng3d.config.**CFG_FOG_DEFAULT = {'color': None, 'enable': False, 'end': 160, 'start':**

Default fog configuration.

This configuration simply disables fog.

peng3d.config.**CFG_LIGHT_DEFAULT = {'enable': False}**

Default lighting configuration.

This configuration simply disables lighting.

peng3d.config.**DEFAULT_CONFIG**

Default configuration values.

All default configuration values are stored here, for more information about specific config values, see *Configuration Options for peng3d*.

class peng3d.config.**Config**(*config=None*, *defaults=None*)

Configuration object imitating a dictionary.

`config` can be any dictionary-style object and is used to store the configuration set by the user. This object only needs to implement the `__getitem__`, `__setitem__` and `__contains__` special methods.

`defaults` can be any dictionary-style object and is only read from in case the `config` object does not contain the key. Every config object is stackable, e.g. you can pass another *`Config`* object as the `defaults` object.

Example for stacking configs:

```
>>> myconf = Config()
>>> myconf2 = Config(defaults=myconf)
>>> myconf["foo"] = "bar"
>>> print(myconf2["foo"])
bar
>>> myconf2["bar"] = "foo"
>>> print(myconf2["bar"])
foo
>>> print(myconf["bar"])
Traceback (most recent call last):
...
KeyError: Key "bar" does not exist
```

There is no limit in stacking configurations, though higher-stacked configs may get slow when defaulting due to propagating through the whole chain.

2.24 `peng3d.util` - Utility Functions and Classes

class peng3d.util.**WatchingList**(*l*, *callback=None*)

Subclass of `list()` implementing a watched list.

A WatchingList will call the given callback with a reference to itself whenever it is modified. Internally, the callback is stored as a weak reference, meaning that the creator should keep a reference around.

This class is used in *`peng3d.gui.widgets.BasicWidget()`* to allow for modifying single coordinates of the pos and size properties.

peng3d.util.**register_pyglet_handler**(*peng*, *func*, *event*, *raiseErrors=False*)

Registers the given pyglet-style event handler for the given pyglet event.

This function allows pyglet-style event handlers to receive events bridged through the peng3d event system. Internally, this function creates a lambda function that decodes the arguments and then calls the pyglet-style event handler.

The `raiseErrors` flag is passed through to the peng3d event system and will cause any errors raised by this handler to be ignored.

See also:

See *`addEventListener()`* for more information.

class `peng3d.util.`**`ActionDispatcher`**

Helper Class to be used to enable action support.

Actions are simple callbacks that are specific to the instance they are registered with.

To be able to use actions, a class must be a subclass of *`ActionDispatcher()`*.

Creation of required data structures is handled automatically when the first action is added.

Internally, this object uses the `actions` attribute to store a map of action names to a list of callbacks.

`addAction`(*action: str*, *func: Callable*, **args*, ***kwargs*)

Adds a callback to the specified action.

All other positional and keyword arguments will be stored and passed to the function upon activation.

`doAction`(*action: str*)

Helper method that calls all callbacks registered for the given action.

class `peng3d.util.`**`SmartRegistry`**(*data: Optional[Dict[str, Any]] = None*, *reuse_ids: bool = False*, *start_id: int = 0*, *max_id: Optional[int] = None*, *default_reg: Optional[Dict[KT, VT]] = None*)

Smart registry allowing easy management of mappings from int to str and vice versa.

Note that bidict is required to be able to use this class.

`data` may be a dictionary to initialize the registry with. Only dictionaries gotten from the *`data`* property should be used.

`reuse_ids` specifies whether or not the automatic ID generator should re-use old, now unused IDs. See *`genNewID()`* for more information.

`start_id` is the lowest ID that will be generated by the automatic ID generator.

`max_id` is the highest ID that will be generated by the automatic ID generator. Should this limit by reached, an `AssertionError` will be raised.

`default_reg` may be a dictionary mapping IDs to names. It will only be used if `data` did not already contain a registry.

It is possible to access the registry via the dict-style `reg[key]` notation. This will return the name of whatever object was used as the key.

Registering is also possible in a similar manner, like `reg[name]=id`. `id` may be `None` to automatically generate one.

This class also supports the `in` operator, note that both IDs and names are checked.

`data`

Read-only property to access the internal data.

This is a dictionary containing all information necessary to re-create the registry via the `data` argument.

The returned object is fully JSON/YAML/MessagePack serializable, as it only contains basic python data types.

genNewID() → int
Generates a new ID.

If `reuse_ids` was false, the new ID will be read from an internal counter which is also automatically increased. This means that the newly generated ID is already reserved.

If `reuse_ids` was true, this method starts counting up from `start_id` until it finds an ID that is not currently known. Note that the ID is not reserved, this means that calling this method simultaneously from multiple threads may cause the same ID to be returned twice.

Additionally, if the ID is greater or equal to `max_id`, an `AssertionError` is raised.

normalizeID(*in_id: Union[int, str]*) → int
Takes in an object and normalizes it to its ID/integer representation.

Currently, only integers and strings may be passed in, else a `TypeError` will be thrown.

normalizeName(*in_name: Union[int, str]*) → str
Takes in an object and normalizes it to its name/string.

Currently, only integers and strings may be passed in, else a `TypeError` will be thrown.

register(*name: str*, *force_id: Optional[int] = None*) → int
Registers a name to the registry.

`name` is the name of the object and must be a string.

`force_id` can be optionally set to override the automatic ID generation and force a specific ID.

Note that using `force_id` is discouraged, since it may cause problems when `reuse_ids` is false.

peng3d.util.**default**(*arg: Optional[T]*, *_default: T*) → T
Small helper function that replaces the given argument with a default if the argument is `None`.

This can also be written as a ternary expression in-line, but using this function makes the purpose clearer and easier to read.

class peng3d.util.**default_property**(*parent: Optional[str] = None*, *name: Optional[str] = None*, ***, *parent_attr: Optional[str] = None*)
Special property decorator/class that allows for easy defaulting of attributes to the parents' attributes.

This class can either be used as a decorator or as a class attribute.

For decorator usage, simply decorate an empty method with the name of the attribute to default, passing the name of the attribute the parent is stored in:

```
class A:
    @default_property("parent")
    def my_attr(self): ...
```

Accessing `my_attr` will then return the value of the `my_attr` attribute of the `parent` attribute of the class instance. Setting `my_attr` will not touch the attribute of the same name of the parent, but rather set an internal attribute, causing all subsequent accesses to return this local attribute.

Setting the property to `None` will reset the whole mechanism, causing all accesses until the next write to return the defaulted value.

Internally, all this is handled by a shadow attribute with the same name as the actual property, but prefixed by an underscore. This internal attribute may also be written to directly, which is especially useful in constructors.

Deleting the property will also just reset it, until it is next written.

Alternatively, this class can also be used as a class attribute, for the same effect as described above:

```
class A:
    my_attr = default_property("parent")
```

Note that this only works if the property is defined in the class body. Later assignment to the class object is possible, but requires providing the `name` argument, since auto-detecting the attribute name is then not possible.

To simplify creation, it is possible to set the `PARENT_ATTR` class attribute to provide a default first argument to *`default_property`*. This is usually worthwhile if multiple `default_properties` are used within the same class hierarchy, especially since the class attribute value can be inherited.

The `parent_attr` keyword-only argument may be passed to override what attribute of the parent is used as a default.

2.25 `peng3d.util.gui` - GUI Utility Functions and Classes

`peng3d.util.gui.`**`mouse_aabb`**(*mpos: List[float], size: List[float], pos: List[float]*) → bool
AABB Collision checker that can be used for most axis-aligned collisions.

Intended for use in widgets to check if the mouse is within the bounds of a particular widget.

`peng3d.util.gui.`**`points2htmlfontsize`**(*points: float*) → float
Approximate font size converter, converts from Points to HTML `<font>` tag font sizes.

Note that this method is very inaccurate, since there are only seven possible output values that represent at least 25 input values. When in doubt, this function always rounds down, e.g. every input value less than eight is converted to HTML size 1.

`class` `peng3d.util.gui.`**`ResourceGroup`**(*data*, *parent=None*)
Pyglet Group that represents a Resource as returned by the *`ResourceManager()`*.

This Group should automatically merge different groups with different resources that are on the same texture atlas.

2.26 `peng3d.util.types` - Custom Types

This module contains various helper types used throughout `peng3d`.

Most of these types are aliases, which ensure compatibility with existing applications that for example pass in literal values (which wouldn't be of a custom type without casting).

`peng3d.util.types.`**`ColorRGB = typing.List[int]`**
RGB Color represented as three integers from zero to 255.

`peng3d.util.types.`**`ColorRGBA = typing.List[int]`**
RGBA Color represented as four integers from zero to 255.

`peng3d.util.types.`**`ColorRGBFloat = typing.List[float]`**
RGB Color represented as three floats from zero to one.

`peng3d.util.types.`**`ColorRGBAFloat = typing.List[float]`**
RGBA Color represented as four floats from zero to one.

`peng3d.util.types.`**`BorderStyle = typing.Union[typing.Type[ForwardRef('Borderstyle')], str]`**
Border style identifier.

Either a subclass of *`Borderstyle()`* or a string identifier.

`peng3d.util.types.`**`BackgroundType = typing.Union[ForwardRef('Layer'), typing.Callable, list,`**

Type encompassing all allowed types for backgrounds of menus and submenus.

See also:

See *`setBackground()`* for further details.

`peng3d.util.types.`**`DynPosition = typing.Union[typing.List[float], typing.Callable[[float, fl`**

Dynamic position of a widget or container.

Can be either a static position as a list, a callback function or a `LayoutCell`.

`peng3d.util.types.`**`DynSize = typing.Union[typing.List[float], typing.Callable[[float, float]`**

Dynamic size of a widget or container.

Can be either a static size as a list or a callback function. May be `None` if the position is a `LayoutCell`.

`peng3d.util.types.`**`DynTranslateable = typing.Union[str, ForwardRef('_LazyTranslator')]`**

A string that may be either fixed, translated or even dynamically translated.

If a method or class accepts this type, it means that it supports dynamic translation of this attribute, unless indicated otherwise in its documentation.

2.27 `peng3d.version` - Version information

`peng3d.version.`**`VERSION`**

Full version number, compliant with semantic versioning

Used to display the version in the title of the documentation.

Also used for the version in `setup.py`.

Changed in version 1.10.0: Before 1.10.0, `peng3d` did not quite comply with semantic versioning. This is mainly due to the `a1` suffix on most version names.

`peng3d.version.`**`RELEASE`**

Currently the same as *`VERSION`*.

Used to display the version on the top-right of the documentation.

CHAPTER 3

Events used by Peng3d

See also:

This document describes the events used by peng3d, see *`peng3d.peng.Peng.sendEvent()`* for information about the event system itself.

Note that there is no completely safe way to get a list of all events used by an application, but you should get most events by setting the config value `debug.events.dumpfile` to a valid file name and running the application in question. Make sure to trigger all events, or else they may not appear in the list.

This document is sectioned after the categories of events used.

Note that many applications will add their own events, which should be listed in their documentation.

3.1 Peng3d Events using `sendEvent()`

Events listed here can be sent via the *`sendEvent()`* method and be received via *`addEventListener()`*.

If possible, this system should be used, as it is better and has many improvements over previous systems.

Most of these events use a dictionary containing at least the `peng` key as their data parameter.

3.1.1 Special events

These events are special and should not be sent manually, they are mostly for backwards-compatibility.

`peng3d:pyglet`

Special event sent by *`sendPygletEvent()`* for compatibility.

Additional parameters:

`args` is a list of the given parameters.

`window` is the window this event originated from.

`src` is the object this event was sent via.

`event_type` is the pyglet event type.

See also:

See *`pyglet:*`* for another way of accessing pyglet events.

`pyglet:*`
Special event sent by *`sendPygletEvent()`* for compatibility.

See *`peng3d:pyglet`* for more information on the given parameters.

3.1.2 `peng3d:peng.*` Events Category

These events are typically sent by the main *`Peng()`* instance.

`peng3d:peng.run`
Triggered once when calling *`run()`* just before starting the event loop.

Additional parameters are `window` set to the main window object and `evloop` set to the argument of the same name.

`peng3d:peng.exit`
Triggered once the pyglet event loop exits.

Note that the calling method may cause the program to continue running.

This event has no additional parameters.

3.1.3 `peng3d:window.*` Events Category

These events are sent to mark changes to an instance of *`PengWindow()`*.

Note that some of these events are not sent by the window itself and do not require a window to exist.

`peng3d:window.create.pre`
`peng3d:window.create`
`peng3d:window.create.post`
These events are sent when the main window is created.

The event *`peng3d:window.create.pre`* has the additional parameter `cls` containing the class used to create the window.

The events *`peng3d:window.create`* and *`peng3d:window.create.post`* both have the additional parameter `window` set to the window object.

Note that the `window` attribute of *`Peng()`* is only available after the handling of *`peng3d:window.create`* has finished.

`peng3d:window.menu.add`
Triggered whenever a menu is added to the window.

Additional parameters are `window` set to the window object and `menu` set to the menu object.

`peng3d:window.menu.change`
Triggered whenever the active menu is changed.

This event is sent after other event handlers have finished processing.

Additional parameters:

`window` is the current window object.

`old` is the name of the old menu. This may be `None` if there was no active menu.

`menu` is the name of the new menu.

`peng3d:window.toggle_exclusive`

Triggered whenever the mouse exclusivity is changed via *`toggle_exclusivity()`*.

Additional parameters are `window` set to the window object and `exclusive` set to the current exclusivity state.

3.1.4 `peng3d:rsrc.*` Events Category

These events are sent by the *`ResourceManager()`* to signal that either the manager itself was modified or a resource was changed.

`peng3d:rsrc.init.pre`
`peng3d:rsrc.init`
`peng3d:rsrc.init.post`

These events are sent when the resource manager is first initialized.

The event *`peng3d:rsrc.init.pre`* has the additional parameter `basepath` containing the base path of the new resource manager.

The events *`peng3d:rsrc.init`* and *`peng3d:rsrc.init.post`* both have the additional parameter `rsrcMgr` set to the newly created resource manager.

Note that the `resourceMgr` attribute of *`Peng()`* is only available after the handling of *`peng3d:rsrc.init`* has finished.

`peng3d:rsrc.category.add`

Sent when a new resource category is added.

The additional parameter `category` is set to the name of the new category.

`peng3d:rsrc.tex.load`

Sent when a texture resource is first loaded.

Additional parameters are `name` and `category` set to their corresponding arguments given to *`loadTex()`*.

`peng3d:rsrc.model.load`

Sent when a model resource is first loaded.

Additional parameters are `name` set to the name of the model.

3.1.5 `peng3d:i18n.*` Events Category

See also:

See *`TranslationManager()`* for more information about the translation system.

`peng3d.i18n.set_lang`

Sent whenever the default language is set.

Note that this event is sent regardless of whether or not the language actually changed.

Additional parameters are `i18n`, set to the translation manager, and `lang` set to the new language.

3.1.6 `peng3d:keybind.*` Events Category

These events usually mark an event related to a specific key combination.

See also:

See *KeybindHandler()* for more information on the keybind system.

peng3d:keybind.add

Triggered when a keybind is added to the system.

Additional parameters are all arguments given to *add()*.

peng3d:keybind.change

Triggered when a keybind is changed.

Additional parameters are all arguments given to *changeKeybind()*.

peng3d:keybind.combo
peng3d:keybind.combo.press
peng3d:keybind.combo.release

These events are triggered whenever a key combination is detected.

Note that this event will be sent regardless of whether or not there are any handlers registered for the keybind in question.

peng3d:keybind.combo is always sent, and depending on the `release` flag, either *peng3d:keybind.combo.press* or *peng3d:keybind.combo.release* is also sent.

Additional parameters are the same as the arguments given to *handle_combo()*.

3.2 Pyglet Events using `sendPygletEvent()`

Events listed here can be sent via the *sendPygletEvent()* method and be received via *addPygletListener()*.

There are also several events sent by pyglet itself, see the Pyglet Docs for more information.

Todo: Add docs for custom pyglet events.

CHAPTER 4

Configuration Options for peng3d

Almost all important settings can be configured per-window or globally via the `Peng.cfg` or `Window.cfg` attributes.

4.1 Graphic Settings/OpenGL Base State

For most of these graphical settings, it is important to actually use the exact type specified. For example, you should only pass floats and not integers if the specified type is float.

`graphics.clearColor`

A 4-tuple of RGBA colors used to clear the window before drawing.

Each Color part should be a float between `0` and `1`.

By default, this option is set to `(0.,0.,0.,1.)`.

Be sure to verify that each value is a float, not an integer.

`graphics.wireframe`

A Boolean value determining the polygon-fill-mode used by OpenGL.

`True` results in `GL_LINE` being used, while `False` will result in `GL_FILL` being used.

This option can be used to create a wireframe-like mode.

The default value for this option is `False`.

Note: This option is always turned off by `PengWindow.set2d()` but re-enabled by `PengWindow.set3d()` if necessary.

`graphics.fieldofview`

An float value passed to `gluPerspective()` as the first argument.

For more information about this config option, see the GL/GLU documentation.

By default, this option is set to `65.0`.

`graphics.nearclip`
`graphics.farclip`

An float value specifying the near and far clipping plane, respectively.

These clipping planes determine at what point vertices are cut off to save GPU cycles.

By default, *`graphics.nearclip`* equals `0.1` and *`graphics.farclip`* equals `10000`.

4.1.1 Fog settings

`graphics.fogSettings`

`Config()` object storing the fog-specific settings.

To access fog settings, use `peng.cfg["graphics.fogSettings"]["<configoption>"]` as appropriate.

`graphics.fogSettings["enable"]`

A boolean value activating or deactivating the OpenGL fog.

By default disabled.

`graphics.fogSettings["color"]`

A 4-Tuple representing an RGB Color.

Note that the values should be 0<=n<=1, not in range(0,256).

For most cases, this value should be set to the clear color, else, visual artifacts may occur.

`graphics.fogSettings["start"]`
`graphics.fogSettings["end"]`

Defines start and end of the fog zone.

The end value should be nearer than the far clipping plane to avoid cut-off vertices.

Each value should be a float and is measured in standard OpenGL units.

By default, the fog starts at 128 units and ends 32 units further out.

4.1.2 Light settings

`graphics.lightSettings`

`Config()` object storing the light settings.

To access light settings, use `peng.cfg["graphics.lightSettings"]["<configoption>"]` as appropriate.

`graphics.lightSettings["enable"]`

A boolean value activating or deactivating the light config.

By default disabled.

Todo: Implement light settings with shader system

4.2 Controls

Note that most of these config values are read when the appropriate objects are initialized, this means that you should consult the objects documentation for how to change the option at runtime.

4.2.1 Mouse

`controls.mouse.sensitivity`

Degrees to move per pixel traveled by the mouse.

This applies to both horizontal and vertical movement.

Defaults to `0.15`.

4.2.2 Keyboard

`controls.controls.movespeed`

Speed multiplier for most movements.

Defaults to `10.0`.

`controls.controls.verticalspeed`

Speed multiplier for vertical movement.

Defaults to `5.0`.

These keys are all registered with the `mod` flag set to False, thus they will ignore any modifiers.

`controls.controls.forward`
`controls.controls.backward`
`controls.controls.strafeleft`
`controls.controls.straferight`

Four basic movement keys.

Each of these keys can be changed individually.

Defaults are `w`, `s`, `a` and `d`, respectively.

`controls.controls.jump`

Jump key.

Defaults to `space`.

`controls.controls.crouch`

Crouch key.

Defaults to `lshift`.

4.2.3 Commonly used Key Combination Configuration Values

`controls.keybinds.common.copy`
`controls.keybinds.common.paste`
`controls.keybinds.common.cut`

Key Combinations used to be used by various parts of the GUI.

Currently used by the *`peng3d.gui.text.TextInput()`* Widget for basic clipboard operations.

By default, these are set to the commonly used values of `Ctrl-C` for Copy, `Ctrl-V` for Paste and `Ctrl-X` for Cutting.

4.2.4 General Controls Configuration Values

`controls.keybinds.strict`

Whether or not keybindings should be strict.

See *`peng3d.keybind.KeybindHandler()`* for more information.

4.3 Debug Options

All of these options are disabled by default.

`controls.keybinds.debug`

If enabled, all pressed keybinds will be printed.

`debug.events.dump`

If enabled, all events are printed including their arguments.

Note that `on_draw` and `on_mouse_motion` are never printed to avoid excessive outputs.

`debug.events.logerr`

If enabled, Exceptions catched during event handling are printed.

Note that only `AttributeError` exceptions are catched and printed, other exceptions will propagate further.

`debug.events.register`

If enabled, all event handler registrations are printed.

`debug.events.dumpfile`

If not an empty string, this should point to a valid file path for dumping all event names.

If enabled, all event handler registrations and event sends will be logged to this file. Note that only the name of the event without data is stored and automatically deduplicated.

Defaults to `""`.

4.4 Resource Options

`rsrc.enable`

Enables or Disables the resource module.

By default enabled.

`rsrc.basepath`

Base directory of the Resource Manager.

By default determined via `pyglet.resource.get_script_home()`.

`rsrc.maxtexsize`

Maximum Texture size per bin.

Limits the texture in size, useful if the graphics card has big textures (16kx16k) but only few textures will be needed.

By default set to 1024.

4.5 Translation Options

`i18n.enable`
Enables or Disables the i18n module.

By default enabled.

`i18n.lang`
Determines the default language selected upon startup.

Note that setting this config option after creating the first window will have no effect. Use *`setLang()`* instead.

Currently defaults to `en`, but may be changed to operating system language in the future.

4.6 Event Options

`events.removeonerror`
If True, automatically removes erroring event handlers. Note that the `raiseErrors` parameter takes precedent over this setting.

Defaults to `True`.

`events.maxignore`
An integer number defining the maximum amount of ignored event messages to write to the log file.

This setting is per event, not globally.

Defaults to 3.

4.7 Other Options

`pyglet.patch.patch_float2int`
Enables the float2int patch for pyglet.

See `peng3d.pyglet_patch.patch_float2int()` for more information.

Enabled by default.

Todo: Implement more config options

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index