Pygame GUI

Release 0.6.9

Dan Lawrence

May 06, 2024

CONTENTS

Pygame GUI is a module to help you make graphical user interfaces for games written in pygame. The module is firmly forward looking and is designed to work on Pygame 2 and Python 3.

CHAPTER

ONE

FEATURES

- Theme-able UI elements/widgets - you can use JSON theme files to change the colours, fonts and other appearance related details of your UI without touching your code.
- A subset of HTML is supported for drawing word-wrapped text. Have bold styled words in the middle of a paragraph of text! Stick a link in there! Go absolutely hog wild, within the bounds of the defined subset of HTML the module supports.
- Buttons, text entry, scroll bars and drop down menus all supported, with more on the way.
- A window stack that will let you keep a bunch of moveable windows of 'stuff' around and correctly sorted.
- Support for localizing your GUI into different languages.
- As closely respecting of the pygame way of doing things as possible.

CHAPTER

TWO

INSTALLATION

Install the latest release from PyPi using pip with:

```
pip install pygame_gui -U
```

Or, you can build the latest version from GitHub here by downloading the source, navigating to the project's directory (the one with setup.py in it) and then building it with:

```
python setup.py install
pip install . -U
```

CHAPTER

THREE

WHY IS YOUR PACKAGE CALLED PYGAME-GUI ON PYPI?

PyPI converts all non-letter characters in package names to dashes for web search optimisation reasons. I can assure you that the pygame-gui package on PyPI is this library pygame_gui. Conversely, Python does not allow dashes in package names. So it is not possible to standardise around either convention unless you forgo any kind of non-lowercase letter, pygamegui is already taken as a name on PyPI - so here we are.

Please live with the inconsistency.

CHAPTER

FOUR

SOURCE CODE ON GITHUB

The source code is available from GitHub here .

CHAPTER

FIVE

GETTING STARTED

Try our *Quick Start Guide* here if you are new to Pygame GUI. Check out the *Theme Guide* if you want to learn how to style your GUI.

5.1 Examples

If you want to see Pygame GUI in action have a rifle through the examples project over on GitHub to see some of the stuff the library can do in action.

CHAPTER

SIX

GAME PROJECTS USING PYGAME GUI

- Tower Defence - A tower defence demo game.
- Christmas Adventure - A text adventure demo.

CHAPTER

SEVEN

TABLE OF CONTENTS

7.1 Quick Start Guide

To start making use of Pygame GUI, you first need to have at least the bare bones of a pygame CE project if you don't know much about pygame CE then there is some documentation here and many tutorials across the internet.

Assuming you have some idea what you are doing with pygame CE, I've created a basic, empty pygame CE project with the code below. You can just copy and paste it into an empty python script file.:

```
import pygame


pygame.init()

pygame.display.set_caption('Quick Start')
window_surface = pygame.display.set_mode((800, 600))

background = pygame.Surface((800, 600))
background.fill(pygame.Color('#000000'))

is_running = True

while is_running:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            is_running = False

    window_surface.blit(background, (0, 0))

    pygame.display.update()
```

That should open an empty window upon being run. If it doesn't you may need to install pygame CE.

Next, we need to make sure that we've installed the pygame_gui module. If you haven't, then the quickest way is to open a terminal or Command Prompt and type:

```
pip install pygame_gui
```

Assuming that all installed correctly, then the next step is to head back to our code, import the Pygame GUI module and create a UIManager:

```
import pygame
import pygame_gui


pygame.init()

pygame.display.set_caption('Quick Start')
window_surface = pygame.display.set_mode((800, 600))

background = pygame.Surface((800, 600))
background.fill(pygame.Color('#000000'))

manager = pygame_gui.UIManager((800, 600))

is_running = True

while is_running:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            is_running = False

    window_surface.blit(background, (0, 0))

    pygame.display.update()
```

As you can see the UIManager class, like pygame's display.set_mode() function also needs to know the current size of the screen.

The UI manager handles calling the update, draw and event handling functions of all the UI elements we create and assign to it. To make it do this we need to call these functions on the UIManager object we just created.

```
import pygame
import pygame_gui


pygame.init()

pygame.display.set_caption('Quick Start')
window_surface = pygame.display.set_mode((800, 600))

background = pygame.Surface((800, 600))
background.fill(pygame.Color('#000000'))

manager = pygame_gui.UIManager((800, 600))

clock = pygame.time.Clock()
is_running = True

while is_running:
    time_delta = clock.tick(60)/1000.0
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
```

(continues on next page)

(continued from previous page)

```
            is_running = False

        manager.process_events(event)

    manager.update(time_delta)

    window_surface.blit(background, (0, 0))
    manager.draw_ui(window_surface)

    pygame.display.update()
```

As you may have noticed we also had to create a pygame Clock to track the amount of time in seconds that passes between each loop of the program. We need this 'time_delta' value because several of the UI elements make use of timers and this is a convenient place to get it.

Using .tick() to fix the frame rate of your pygame program is a good idea anyway, otherwise your code will just run as fast as it can go unnecessarily, probably straining the circuits of any computer it runs on.

So, now the UI manager is all setup it's time to create a UI element so we can actually see something on the screen. Let's try and stick a UIButton in the middle of the screen that prints 'Hello World' to the console when we press it.

To start lets make the button.

```
import pygame
import pygame_gui


pygame.init()

pygame.display.set_caption('Quick Start')
window_surface = pygame.display.set_mode((800, 600))

background = pygame.Surface((800, 600))
background.fill(pygame.Color('#000000'))

manager = pygame_gui.UIManager((800, 600))

hello_button = pygame_gui.elements.UIButton(relative_rect=pygame.Rect((350, 275), (100,␣
↪50)),
                                            text='Say Hello',
                                            manager=manager)

clock = pygame.time.Clock()
is_running = True

while is_running:
    time_delta = clock.tick(60)/1000.0
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            is_running = False

        manager.process_events(event)

    manager.update(time_delta)
```

(continues on next page)

(continued from previous page)

```
    window_surface.blit(background, (0, 0))
    manager.draw_ui(window_surface)

    pygame.display.update()
```

Now if you try running the program again you should see a grey rectangle in the middle of the window with the text 'Say Hello' on it, and if you move the mouse over it or click on it the rectangle changes colour. That's what a basic UIButton looks like. If you load a theme file into the UIManager we can change these colours, the font of the text on the button and several other things about it's appearance.

For now though, we won't worry about theming our button - we still need to make it print 'Hello World!' to the console When we click on it. To do that we need to check the pygame event queue:

```
import pygame
import pygame_gui


pygame.init()

pygame.display.set_caption('Quick Start')
window_surface = pygame.display.set_mode((800, 600))

background = pygame.Surface((800, 600))
background.fill(pygame.Color('#000000'))

manager = pygame_gui.UIManager((800, 600))

hello_button = pygame_gui.elements.UIButton(relative_rect=pygame.Rect((350, 275), (100,␣
↪50)),
                                            text='Say Hello',
                                            manager=manager)

clock = pygame.time.Clock()
is_running = True

while is_running:
    time_delta = clock.tick(60)/1000.0
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            is_running = False

        if event.type == pygame_gui.UI_BUTTON_PRESSED:
            if event.ui_element == hello_button:
                print('Hello World!')

        manager.process_events(event)

    manager.update(time_delta)

    window_surface.blit(background, (0, 0))
    manager.draw_ui(window_surface)

```

(continues on next page)

(continued from previous page)

```
    pygame.display.update()
```

Pygame GUI creates events of various types, in this case we are after UI_BUTTON_PRESSED. You can find more documentation on the different event types under *GUI events*. Finally, we do a check to see which specific button has been pressed, since we have a variable for our hello_button and the event also includes a reference to the ui_element that created it, we can just compare the event's ui_element attribute with our hello_button variable to confirm they are one and the same.

Try running the code again and clicking on the button. If it's all worked you should see 'Hello World!' printed to the python console each time you click the button.

Now that we've got the basics of the code up and running, let's try experimenting with a custom theme file in part 2 of this quick start guide:

7.1.1 Quick Start Guides

Quick Start Guide - Part 2: Theming

The first thing we need to do is create an empty theme file.

Open a new text file - call it what ever you like and save it with a .json extension. I use my IDE for this but a simple text editor like windows notepad will also work. JSON is the JavaScript Object Notation file format and commonly used for saving all types of data - in this case theming data - that we can then use across multiple UI Elements.

Start with a simple outline for theming all "button" type elements - see below - inside your new json file

Listing 1: quick_start.json

```
{
    "button":
    {
    }
}
```

Of course the file existing on it's own will not do anything - you will also need to load it into your UIManager. Paths to theme files can either be absolute (starting from the drive location) or relative to the current working directory of your script. Here's the modified quick_start.py file loading our 'quick_start.json' (or whatever you have called it) theme file into the UIManager:

```
import pygame
import pygame_gui


pygame.init()

pygame.display.set_caption('Quick Start')
window_surface = pygame.display.set_mode((800, 600))

background = pygame.Surface((800, 600))
background.fill(pygame.Color('#000000'))

manager = pygame_gui.UIManager((800, 600), theme_path="quick_start.json")

```

(continues on next page)

(continued from previous page)

```
hello_button = pygame_gui.elements.UIButton(relative_rect=pygame.Rect((350, 275), (100,
↪50)),
                                            text='Say Hello',
                                            manager=manager)

clock = pygame.time.Clock()
is_running = True

while is_running:
    time_delta = clock.tick(60)/1000.0
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            is_running = False

         if event.type == pygame_gui.UI_BUTTON_PRESSED:
             if event.ui_element == hello_button:
                 print('Hello World!')

        manager.process_events(event)

    manager.update(time_delta)

    window_surface.blit(background, (0, 0))
    manager.draw_ui(window_surface)

    pygame.display.update()
```

Now we have a custom theme file - but we haven't actually changed anything about our button's appearance with it yet. So let's do that. First up, let's change the colours, to do this you first need to add a colours block to your theme file and add some colours to it - like so:

Listing 2: quick_start.json

```
{
    "button":
    {
        "colours":
        {
            "normal_border": "White",
            "normal_bg": "SlateGray",
            "normal_text": "White"
        }
    }
}
```

You can specify colours in a variety of ways - as detailed in the *Theme Guide* - and you can find out which parts of an element can have their colour changed in the specific theming guide for each element. The guide for the UIButton we are theming here can be found at *UIButton Theming Parameters*.

If you run the quick_start.py script again with these new edits to the theme file, you should start to see some changes at last. Hopefully the button now looks like this:

It's not just colours you can mess around with though. The most versatile category of theming options is the 'misc' group. Next I'm going to add three of these parameters to make the button have a rounded rectangular shape, set the

radius of the four corners and set the border round the edge of the button to be 2 pixels thick.

Listing 3: quick_start.json

```
{
    "button":
    {
        "colours":
        {
            "normal_border": "White",
            "normal_bg": "SlateGray",
            "normal_text": "White"
        },
        "misc":
        {
            "shape": "rounded_rectangle",
            "shape_corner_radius": "10",
            "border_width": "2"
        }
    }
}
```

With those your parameters saved into your theme file, run the quick_start.py file again and hopefully you will see something like this:

And that's about covered the basics of theming.

Congratulations, you've learned most of the basics of using Pygame GUI! If you want to explore more, check out the API Reference and try creating some of the other UI Elements, or have a look at how layout works with the *Layout Guide* - otherwise you could head over to the *Theme Guide* to learn more about how to style your elements.

7.2 Layout Guide

Pygame GUI elements are positioned in three different axes - x (horizontal), y (vertical) and a layer.

Anchors were improved in Version 0.6.5, adding center anchors and removing the need to specify lots of anchors when you only want to alter one.

7.2.1 Horizontal & Vertical positioning

Just as in regular pygame-ce, the x and y axis used by pygame GUI run from 0 in the top left corner down to the pixel size of whatever surface/window you are positioning your elements on.

The standard way of positioning elements is through using a 'relative rectangle'; this rectangle's position is always relative to the container it is inside of. If you do not supply a container when creating your element, they will be assigned the default 'root container' which is created when you make the GUI's UI Manager and is the same size as the supplied window_resolution parameter.

Relative rectangles, by default also follow the pygame-ce style of being defined by four values - a top position, a left position, a width and a height. Unlike pygame-ce we also add a couple of extra features for some elements. For example, UIButtons & UILabels can also have a 'dynamic' width and/or height - which means their final width and height will be determined by the width and height of the text supplied when they are created. To specify the width or height as dynamic you just set the corresponding value in their relative rect parameter to '-1'.

As you will see later, we can also change what the 'top' and 'left' positioning values are relative to. By default they are relative to the 'top' and 'left' position values of their container - but adding an anchor can change this.

If you do supply a container when creating an element, by default it will normally be positioned relative to the top left corner of the container. For example, if we were to position a 'hello' UIButton element inside of a UIWindow container, and set it's relative_rect parameter like so:

```
button_layout_rect = pygame.Rect(30, 20, 100, 20)

UIButton(relative_rect=button_layout_rect,
         text='Hello',
         manager=manager,
         container=ui_window)
```

You would get a result something like this:

The button would maintain it's relative x and y position to the top left corner of the window it's contained inside of, no matter where the window is moved to.

7.2.2 Layout Anchors

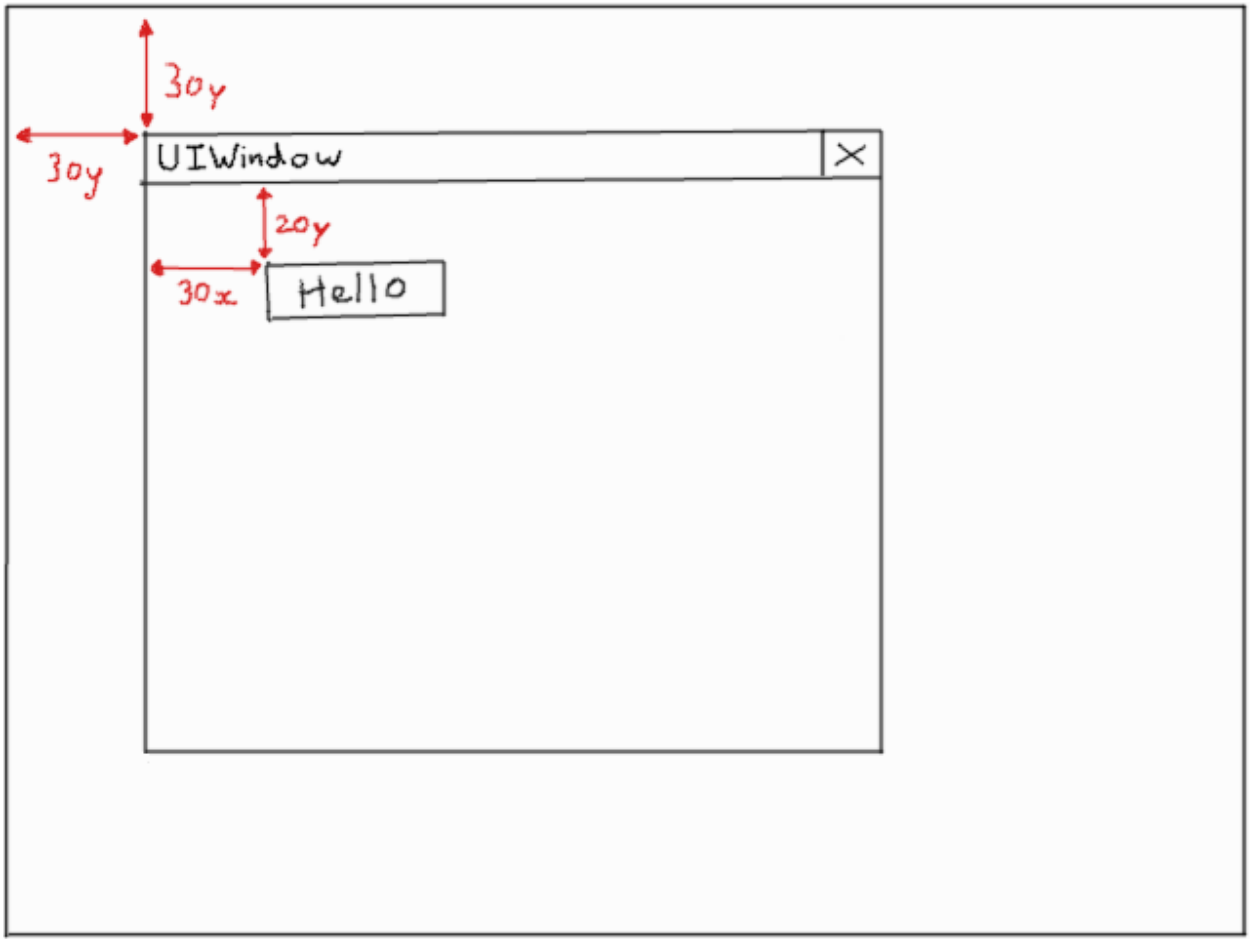
What if you don't want to position your element relative to the top left hand corner of a container? That's where layout anchors come in, by changing the anchors for an element you change what the relative layout rectangle is relative _to_.

The most straight forward use is to switch both layout axes to track different sides of the container. So instead of being relative to the top left we anchor to, say the bottom right. That would look something like this:

```
button_layout_rect = pygame.Rect(0, 0, 100, 20)
button_layout_rect.bottomright = (-30, -20)

UIButton(relative_rect=button_layout_rect,
         text='Hello', manager=manager,
```

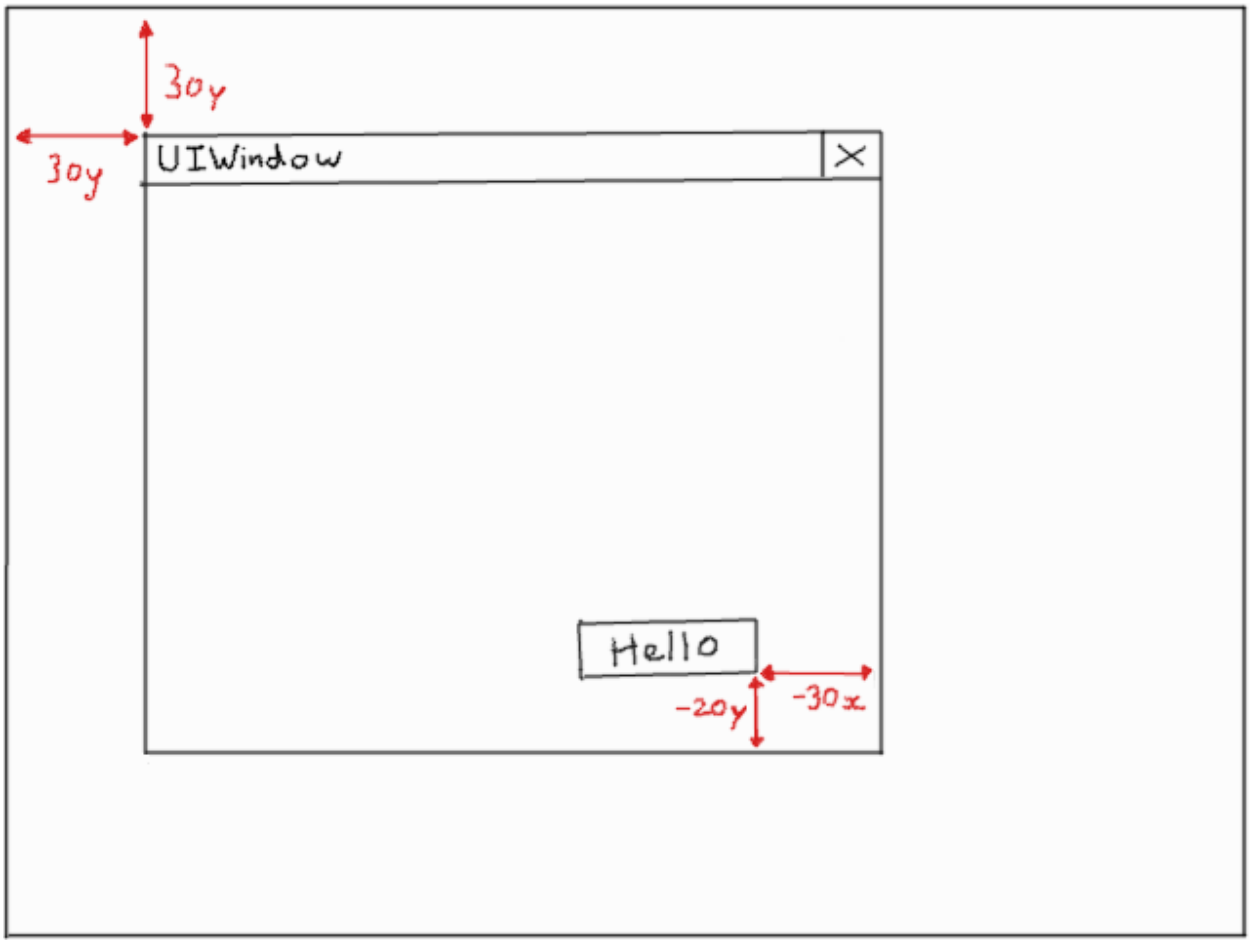
(continues on next page)


30y
30y
UIWindow
20y
30x
Hello

(continued from previous page)

```
            container=ui_window,
            anchors={'right': 'right',
                     'bottom': 'bottom'})
```

Note that both the left and right sides of the button are anchored to the right of our container, and both the top and bottom are anchored to its bottom. This will keep the button the same size whatever size the container is and will produce a layout looking a bit like this:



Another common use case of anchors is centering an element inside a container, in one dimension or both.

```
button_layout_rect = pygame.Rect(0, 0, 100, 20)
UIButton(relative_rect=button_layout_rect,
         text='Hello', manager=manager,
         container=ui_window,
         anchors={'center': 'center'})
```

When centering with an anchor, the normal left & top positions supplied to the element's relative rectangle are adjusted to instead be an offset from center to center. This just makes it a little bit easier to handle these common positions. Thus a rectangle position of (0, 0) as above will place the centre of the element in the center of the container.

If you just want to center in the x dimension, or the y dimension - then the 'centerx' and 'centery' anchors are what you need:

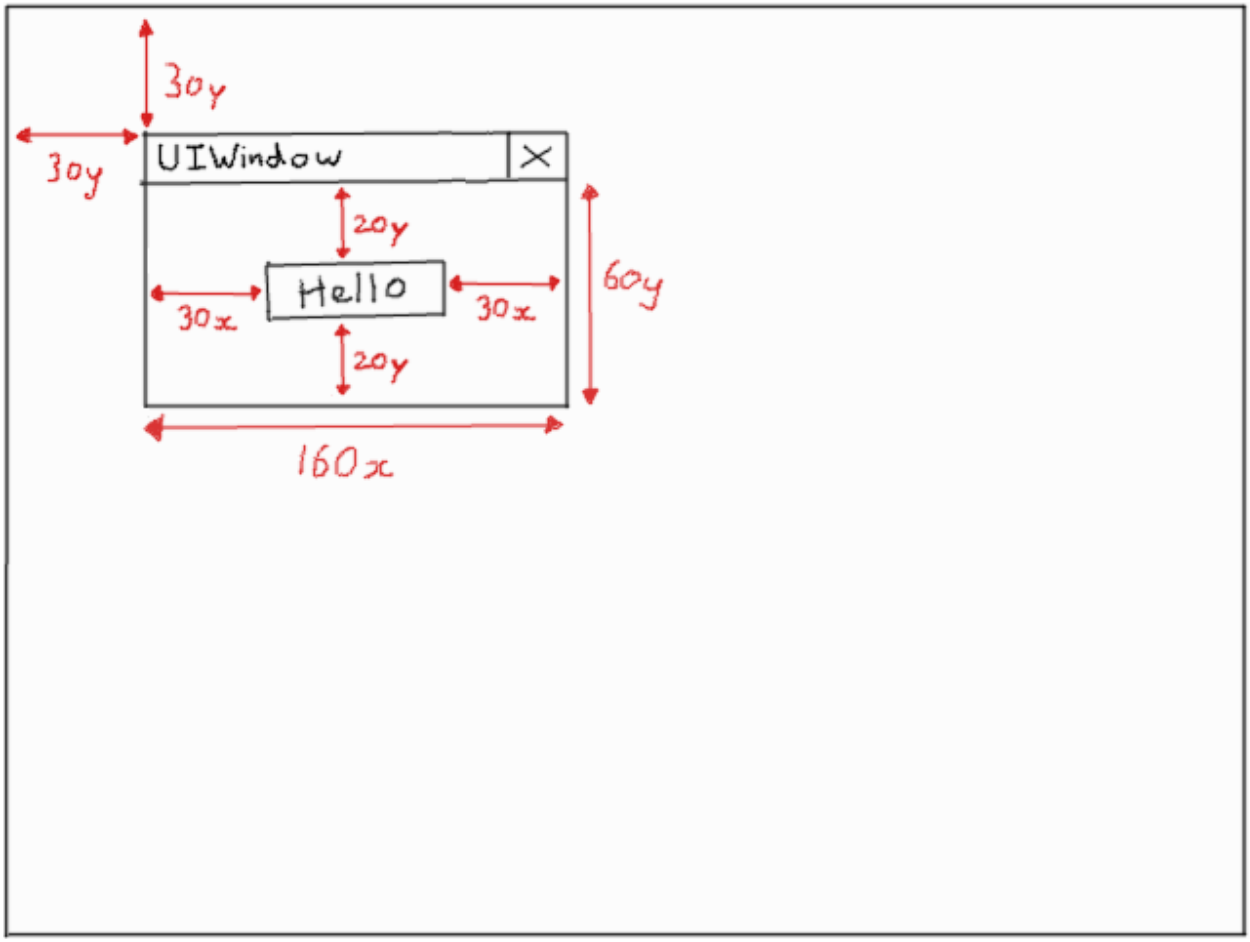
```
button_layout_rect = pygame.Rect(0, -30, 100, 20)
UIButton(relative_rect=button_layout_rect,
         text='Hello', manager=manager,
         container=ui_window,
         anchors={'centerx': 'centerx',
                  'bottom': 'bottom'})
```

Sometimes, you want a layout to change size with it's container so we make maximum use of the available space. In those cases we can simply set the appropriate axis anchors of our button to their counterparts on the window. So to stretch in the x axis (horizontal) set 'left' to 'left' & 'right' to 'right'. To stretch in the y axis (vertical) set 'top' to 'top' & 'bottom' to 'bottom'. For example, here is a hello button with a stretch (both x & y axes) anchor setup:

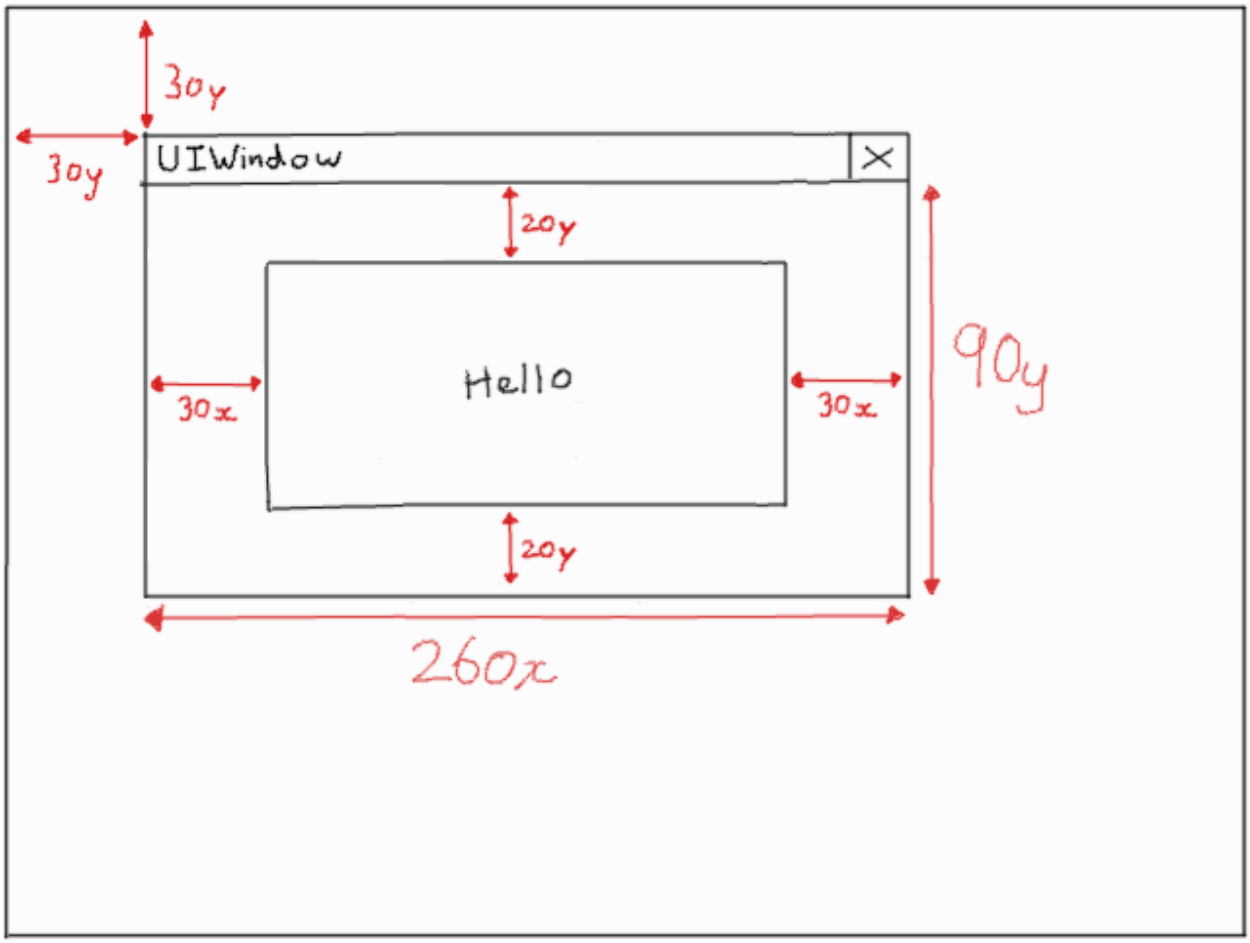
```
button_layout_rect = pygame.Rect(30, 20, 100, 20)

UIButton(relative_rect=button_layout_rect,
         text='Hello', manager=manager,
         container=ui_window,
         anchors={'left': 'left',
                  'right': 'right',
                  'top': 'top',
                  'bottom': 'bottom'})
```

Here's what it might look like placed in a small UIWindow:



And here's what happens to it when we resize the UIWindow to be a bit larger:



You'll note the gaps between the edges of the window have been maintained.

7.2.3 Invalid anchors

Some anchor combinations are currently invalid. For example, you can't set:

```
button_layout_rect = pygame.Rect(0, 0, 100, 20)
UIButton(relative_rect=button_layout_rect,
         text='Hello', manager=manager,
         container=ui_window,
         anchors={'left': 'right',
                  'right': 'left'})
```

Nor, the similar inversion for 'top' and 'bottom'. Currently the 'center' anchor can only be set to anchor to 'center', and likewise with 'centerx' and 'centery'. These restrictions were made cleared in Version 0.6.5.

7.2.4 Anchor targets

What if you don't want to position elements relative to container edges, but relative to other elements inside the container? For that we have anchor targets.

The first thing to appreciate about anchor targets is that because they are inside the container with the element being positioned, rather than the element being inside of them, the sides we are anchoring to are reversed. For example, anchoring the right hand side of our element to be positioned, to the right of the container is straight forward, but if we add an element as a 'right_target' we will actually anchor to the left hand side of this element, for the purposes of positioning.

You might get some strange results if you mix up anchoring direction schemes and anchor targets, generally it is a good idea to match the direction scheme of the element you are anchoring to.

Anchor targets are most useful when you have dynamically sized elements where you don't know how large the button next to you is going to be but you know you want your right hand side to be ten pixels away from it's left hand side.

Anchor targets are optional, just don't add them if you don't want to use them and positioning will default to the container edges. They are specified like this:

```
button_3 = pygame_gui.elements.UIButton(relative_rect=pygame.Rect((-10, -40), (-1, 30)),
                                        text='Anchored', manager=manager,
                                        container=dynamic_dimensions_window,
                                        anchors={'bottom': 'bottom',
                                                 'right': 'right',
                                                 'bottom_target': button_1,
                                                 'right_target': button_2})
```

7.2.5 Dynamically sized elements

Certain elements, such as UIButtons, can have dynamic sizes where their size is determined by the contents (Usually by passing in -1). In these cases positioning the top left in the relative rectangle should be done as if the dynamic dimension was 0 length. When the dimension is eventually calculated it will be added into the positioning formula.

7.2.6 UI Layers

UI Layers start at 0, which represents the lowest level, and progress upwards as they are needed. Things in higher layers will be drawn on top of things in lower layers if they overlap.

Most of the time using Pygame GUI you do not have to interact too much with the layer system. UIs don't tend to be designed with their interactive bits overlapping that often, and when they do they tend to be in windows. The main exception is for groups of elements, used for things like, HUDs which may overlap UI elements that appear 'in' game worlds; such as monster health bars. For these occasions there is the UIPanel element which works as a container, much like a UI Window, except you specify what layer of the UI it will draw on (and thus what it will appear on top of).

So how do you know where to position your UI Panels? Well for that purpose and for any other time you might need to interrogate the layer system there is a layer debug mode that you can activate by calling a function on the UI manager. This should let you figure out how many layers are being used for your 'in game' UI stuff and thus where to position your Panel. Here's how to turn it on:

```
ui_manager.set_visual_debug_mode(True)
```

It gives you information as a snapshot of the current state of the UI, so I recommend temporarily binding it to a keypress - then you can toggle it on and off at different times in your game. It will not keep track of any changes in the UI after being turned on.

7.3 Theme Guide

Pygame UI Elements can pretty much all be themed in some manner. A theme is created by loading a theme file in JSON format. To load one, simply pass the path to the theme file into the UIManager when you create it. Like so:

```
manager = pygame_gui.UIManager((800, 600), 'theme.json')
```

Paths can be absolute or relative. Relative paths are relative to the current working directory of your python application at runtime - not to the file location where you create the manager.

You can also theme your elements 'on the fly', while your pygame application is running. Just edit your theme file and save it, and you should see the UI update to reflect your changes. This is particularly helpful when trying to fiddle with colours, or visualise what the theming parameters do. You can also turn off live theming if you want to save a few CPU cycles by setting that option when you create your UI manager.

The most basic theming you can do is to set the default colours for the UI, which are the colours used if no more element specific colour is specified in the theme.

To set these you need to create a 'defaults' block in your JSON theme file, and a 'colours' sub-block within that. Then within the colours block you can start to set individual colours by their IDs. It should look something like this:

Listing 4: theme.json

```
{
    "defaults":
    {
        "colours":
        {
            "normal_bg":"#45494e",
            "hovered_bg":"#35393e",
            "disabled_bg":"#25292e",
            "selected_bg":"#193754",
            "dark_bg":"#15191e",
            "normal_text":"#c5cbd8",
            "hovered_text":"#FFFFFF",
            "selected_text":"#FFFFFF",
            "disabled_text":"#6d736f",
            "link_text": "#0000EE",
            "link_hover": "#2020FF",
            "link_selected": "#551A8B",
            "text_shadow": "#777777",
            "normal_border": "#DDDDDD",
            "hovered_border": "#B0B0B0",
            "disabled_border": "#808080",
            "selected_border": "#8080B0",
            "active_border": "#8080B0",
            "filled_bar":"#f4251b",
            "unfilled_bar":"#CCCCCC"
        }
    }
}
```

To add on, you don't only have to use hex values if you prefer, but you also can use the different supported colour syntax. It's all up to preference

Table 1: Supported Colour Inputs

Name	Accepted Value Types	Example	Notes
Colour Name (**recomme**	A valid string representation of a colour	blue	A List of valid colour names can be found here
Hex	6 Hexadecimal Digits	#A2F3BB	Native Pygame Color
Hex (With Alpha)	8 Hexadecimal Digits	#C2558F9F	Native Pygame Color
Shorthand Hex	3 Hexadecimal Digits	#FAB	Expands doubly (e.g. would be #FFAABB)
Shorthand Hex (With Alpha)	4 Hexadecimal Digits	#AF2F	Expands doubly (e.g. would be #AAFF22FF)
RGB	3 numbers between 0 and 255	rgb(20, 230, 43)	Red, Green, Blue
RGBA	4 numbers between 0 and 255	rgba(60, 30, 97, 255)	Red, Green, Blue, Alpha
HSL	Degree, 2 Percentage Values	hsl(190deg, 40%, .5)	Hue, Saturation, Luminance
HSLA	Degree, 3 Percentage Values	hsla(10deg, 40%, .5, 0.7)	Hue, Saturation, Luminance, Alpha
HSV	Degree, 2 Percentage Values	hsv(282deg, 63%, 92%)	Hue, Saturation, Value
HSVA	Degree, 3 Percentage Values	hsva(20deg, 40%, .5, 70%)	Hue, Saturation, Value, Alpha
CMY	3 percentage values	cmy(.3, .5, .7)	Cyan, Magenta, Yellow

Table 2: Value Types

Name	Description	Examples	Invalid Examples	Used In
Degree	**An Integer Value Bounded from 0 to 360** • Can optionally have the "deg" unit appended to the end	270, 30deg, 45deg	37.5, 32.8deg, -15deg, -22	HSL, HSLA, HSV, HSVA
Percentage	**Can accept data in two formats:** • A float value bounded from 0 to 1 • A percentage value with an integer value bounded from 0 to 100	40%, 0.67, 27%	0.4%, 2, 200%	HSL, HSLA, HSV, HSVA, CMY
U8	An integer value bounded from 0 to 255	12, 55, 154, 243, 255, 0	12.4, -23, -27.4, 20.0	RGB, RGBA

Note: If you'd like to easily find a custom colour that is not provided, colorpicker.me by QvCool is a great tool to choose a colour value to use and paste in your theme file

Listing 5: theme.json with different valid colour expressions

```
{
    "defaults":
    {
        "colours":
        {
            "normal_bg": "#f2f",
            "hovered_bg": "#ff7a",
            "disabled_bg": "rgb(200, 150, 60)",
            "selected_bg": "rgba(20, 50, 89, 225)",
            "dark_bg": "hsl(30, 0.6, 0.7)",
            "normal_text": "hsla(3deg, 0.5, 0.7, 90%)",
            "hovered_text": "#FFFFFF",
            "selected_text": "purple",
            "disabled_text": "RGB(40, 70, 90)",
            "link_text": "HSV(50deg, 30%, 40%)",
            "link_hover": "cmy(50%, 30%, 0.7)",
            "link_selected": "teal",
            "text_shadow": "skyblue",
            "normal_border": "gold",
        }
    }
}
```

Note: Some More Notes About Colours

- In shorthand hex values like **#fff** and **#ffff**, each value represents itself twice. for example: **#123** == **#112233**

and **#1234** == **#11223344**

- Color Model names are **not** case sensitive, you can write *RGB*, *rGb*, or *Rgb* anyway you like, it will not affect the validity of the colour string
- Hex values are also **not** case sensitive, **#FFF** and **#fff** are exactly the same
- Colour names are **not** case sensitive, RED, Red, and red are all the same colour

---

Of course, colours are not just colours - they can also be gradients, which have a very similar syntax. Like so:

Listing 6: theme.json

```
{
    "defaults":
    {
        "colours":
        {
           "normal_bg":"#45494e,#65696e,90",
           "hovered_bg":"rgb(30, 40, 60),#f3f,rgb(50, 60, 70),90deg"
        }
    }
}
```

Where the first two (or three) parameters indicate the colours used in the gradient, separated by commas, and the last parameter indicates the direction of the gradient as an angle in degrees (from 0 to 360).

To add theming for specific UI elements you then need to add additional blocks at the same level as the 'defaults' block. These blocks require an ID that references which elements that they apply to. IDs have a hierarchy allowing us to reference elements that are part of other elements. To address sub-elements we join them with a full stop For example, if we wanted to theme the vertical scroll bars that are part of a text box we could use 'text_box.vertical_scroll_bar' as the theme ID.

The parts of a theming block ID can be made up either of their element IDs or an 'ObjectID' which is passed to the element when it is created. ObjectID objects contain two string IDs - one called 'object_id' intended for identifying this specific object and another called 'class_id' for identifying this object as part of a class of objects that share theming parameters.

As a rule the entries in more specific theming ID blocks are preferred to more general ones. This means entries under your ObjectID's 'object_id' are preferred over it's 'class_id' and the 'class_id' is preferred over it's 'element_id'. It also means that if your object is part of a hierarchy, then IDs that specify more of the hierarchy will be preferred over those that only specify the final part of theme ID.

e.g. for the buttons on a scroll bar:

- 'vertical_scroll_bar.button' preferred over 'button'
- 'vertical_scroll_bar.@arrow_button' preferred over 'vertical_scroll_bar.button'
- 'vertical_scroll_bar.#bottom_button' preferred over 'vertical_scroll_bar.@arrow_button'

7.3.1 Object IDs - in depth

By convention pygame_gui starts an 'element_id' with no prefix, a 'class_id' with a prefix of '@' and an 'object_id' with a prefix of '#'. These are just conventions and not enforced by the code, but I find it helps make it easier to remember what type of ID each block refers to in larger theme files. Your ids must match between the code and the json theme file (including any prefixes in both).

To create an ObjectID for one of your elements, you first need to import the ObjectID class from the core submodule, then you can create on and pass it into your element when you create it. See the example below:

Listing 7: object_id.py

```
from pygame_gui.core import ObjectID
from pygame_gui.elements import UIButton

...  # other code omitted here -
     # see quick start guide for how to get up and running with a single button

hello_button = UIButton(relative_rect=pygame.Rect((350, 280), (-1, -1)),
                        text='Hello',
                        manager=manager,
                        object_id=ObjectID(class_id='@friendly_buttons',
                                           object_id='#hello_button'))
```

Once the ObjectID is in place in the code you can refer to it in a block in your loaded theme file, like so:

Listing 8: theme.json

```
{
    "button":
    {
        "misc":
        {
            "border_width": "1",
            "shadow_width": "2"
        }
    },
    "@friendly_buttons":
    {
        "misc":
        {
            "shadow_width": "5",
            "shape": "rounded_rectangle"
        }
    },
    "#hello_button":
    {
        "misc":
        {
            "text_horiz_alignment": "left"
        }
    }
}
```

If you want to change a created element's Object ID later on, just use the *.change_object_id()* function. E.g.:

Listing 9: object_id.py

```
from pygame_gui.core import ObjectID
from pygame_gui.elements import UIButton

...  # other code omitted here -
     # see quick start guide for how to get up and running with a single button

hello_button = UIButton(relative_rect=pygame.Rect((350, 280), (-1, -1)),
                        text='Hello',
                        manager=manager,
                        object_id=ObjectID(class_id='@friendly_buttons',
                                           object_id='#hello_button'))

hello_button.change_object_id(ObjectID(class_id='@unfriendly_buttons',
                                       object_id='#hello_button'))
```

7.3.2 Theme block categories

There are four general categories of theming which each have their own sub-blocks under the theme block IDs:

- 'colours'
- 'font'
- 'misc'
- 'images'

Here's an example of adding a 'button' theme block to the JSON file above:

Listing 10: theme.json

```
{
    "defaults":
    {
        "colours":
        {
            "normal_bg":"#45494e",
            "hovered_bg":"#35393e",
            "disabled_bg":"#25292e",
            "selected_bg":"#193754",
            "dark_bg":"#15191e",
            "normal_text":"#c5cbd8",
            "hovered_text":"#FFFFFF",
            "selected_text":"#FFFFFF",
            "disabled_text":"#6d736f",
            "link_text": "#0000EE",
            "link_hover": "#2020FF",
            "link_selected": "#551A8B",
            "text_shadow": "#777777",
            "normal_border": "#DDDDDD",
            "hovered_border": "#B0B0B0",
            "disabled_border": "#808080",
```

(continues on next page)

(continued from previous page)

```
            "selected_border": "#8080B0",
            "active_border": "#8080B0",
            "filled_bar":"#f4251b",
            "unfilled_bar":"#CCCCCC"
        }
    },

    "button":
    {
        "colours":
        {
            "normal_bg":"#45494e",
            "hovered_bg":"#35393e",
            "disabled_bg":"#25292e",
            "selected_bg":"#193754",
            "active_bg":"#193754",
            "dark_bg":"#15191e",
            "normal_text":"#c5cbd8",
            "hovered_text":"#FFFFFF",
            "selected_text":"#FFFFFF",
            "disabled_text":"#6d736f",
            "active_text":"#FFFFFF",
            "normal_border": "#DDDDDD",
            "hovered_border": "#B0B0B0",
            "disabled_border": "#808080",
            "selected_border": "#8080B0",
            "active_border": "#8080B0"
        },

        "misc":
        {
            "tool_tip_delay": "1.5"
        }
    }
}
```

Each of the UI Elements supports different theme options, what exactly these are is detailed in the individual theming guide for each element shown below.

7.3.3 Theme Prototypes

As well as creating theming blocks that address specific elements, or classes of elements, you can also create theming blocks that don't have an ID that matches any element in your UI.

Why would you do this?

To save yourself a bunch of typing by taking advantage of theme prototypes, that's why! In every element theme block you create you can also specify a 'prototype' theme block which will be loaded as that block parameters first before any changes are applied. This lets us quickly apply a bunch of identical theming changes to multiple different parts of our UI theme.

Prototype theme blocks must be defined higher in the theme file than where they are used, otherwise they won't exist to be imported.

If you are using multiple locale specifiers in font blocks, you will need to specify these in every font block in your prototype hierarchy so that the theming can determine which locale's font to apply the theming to.

Here's a quick example of using a simple prototype:

Listing 11: theme.json

```
{
   "#new_shape_style":
   {
      "misc":
      {
         "shape": "rectangle",
         "border_width": "2",
         "shadow_width": "1"
      }
   },

   "button"
   {
      "prototype": "#new_shape_style"
   },

   "horizontal_slider"
   {
      "prototype": "#new_shape_style",
      "misc":
      {
         "enable_arrow_buttons": 0
      }
   }
}
```

7.3.4 Multiple Theme Files

Because of the way that pygame_gui loads theme files you can load multiple theme files with different stuff defined in each one into a single UI Manager:

```
manager = pygame_gui.UIManager((800, 600), 'base_theme.json')
manager.get_theme().load_theme('menu_theme.json')
manager.get_theme().load_theme('hud_theme.json')
```

As long as you keep your IDs distinct you can divide your theming up into lots of different files.

Alternatively, if memory is tight and you are using lots of data in your themes, you could also use different UI Managers with different loaded themes for different states of your game.

7.3.5 Theme Options Per Element

UIButton Theming Parameters

The *UIButton* theming block id is 'button'.

Colours

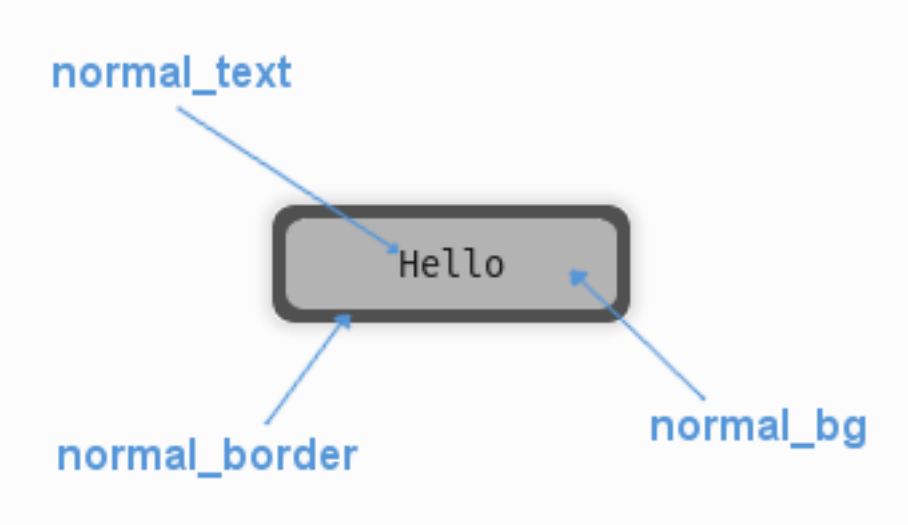


Fig. 1: A diagram of which part of the element is themed by which colour parameter. This correspondence is the same for the parameters named for the different button states e.g. the parameter 'hovered_bg' themes the same area as 'normal_bg' when the button is in the hovered state.

UIButton makes use of these colour parameters in a 'colours' block. Most of these colours can also be a colour gradient:

- "**normal_bg**" - The background colour/gradient of the button in the default state.
- "**hovered_bg**" - The background colour/gradient of the button when the mouse pointer is over it.
- "**disabled_bg**" - The background colour/gradient of the button when the button has been disabled (so users cannot interact with it)
- "**selected_bg**" - The background colour/gradient of the button when the button has select focus.
- "**active_bg**" - The background colour/gradient of the button 'mid-click', you will mostly see it while dragging things around via buttons.
- "**normal_text**" - The colour/gradient of the button's text in the default state.
- "**hovered_text**" - The colour/gradient of the button's text when the mouse pointer is over it.
- "**disabled_text**" - The colour/gradient of the button's text when the button has been disabled (so users cannot interact with it)
- "**selected_text**" - The colour/gradient of the button's text when the button has select focus.
- "**active_text**" - The colour/gradient of the button's text (if any) 'mid-click', you will mostly see it while dragging things around via buttons.
- "**normal_border**" - The colour/gradient of the border around the button (if it has one), in the default state.
- "**hovered_border**" - The colour/gradient of the border around the button (if it has one), in the hovered state.

- "**disabled_border**" - The colour/gradient of the border around the button (if it has one), in the disabled state.
- "**selected_border**" - The colour/gradient of the border around the button (if it has one), in the selected state.
- "**active_border**" - The colour/gradient of the border around the button (if it has one), in the active state.
- "**normal_text_shadow**" - The colour of the shadow behind the button's text (if it has one), in the default state.
- "**hovered_text_shadow**" - The colour of the shadow behind the button's text (if it has one), in the hovered state.
- "**disabled_text_shadow**" - The colour of the shadow behind the button's text (if it has one), in the disabled state.
- "**selected_text_shadow**" - The colour of the shadow behind the button's text (if it has one), in the selected state.
- "**active_text_shadow**" - The colour of the shadow behind the button's text (if it has one), in the active state.

Font

UIButton accepts a font specified in the theme via a 'font' block. A 'font' block has these parameters:

- "**name**" - Necessary to make a valid block. This is the name that this font goes by in the UI, if this is a new font then subsequent font instances with different styles or sizes should use the same name.
- "**locale**" - Optional parameter to set this font as belonging to a particular locale only. See the *Localization* guide. You will need to keep repeating the locale specifier if using prototypes to make a hierarchy.
- "**size**" - Necessary to make a valid block. This is the point size of the font to use on the button.
- "**bold**" - Optional parameter. Set it to "1" to make this font bold.
- "**italic**" - Optional parameter. Set it to "1" to make this font italic.

There are two methods to refer to font resource locations. First, using packaged resources:

- **"regular_resource - The location of this font's file with no particular style applied.**
  - **package** - The name of the python package containing this resource - e.g. 'data.fonts'
  - **resource** - The file name of the resource - e.g. 'FiraCode-Regular.ttf'
- **"bold_resource" - The location of this font's file with bold style applied.**
  - **package** - The name of the python package containing this resource - e.g. 'data.fonts'
  - **resource** - The file name of the resource - e.g. 'FiraCode-Bold.ttf'
- **"italic_resource" - The location of this font's file with italic style applied.**
  - **package** - The name of the python package containing this resource - e.g. 'data.fonts'
  - **resource** - The file name of the resource - e.g. 'FiraMono-Italic.ttf'
- **"bold_italic_resource" - The location of this font's file with bold and italic style applied.**
  - **package** - The name of the python package containing this resource - e.g. 'data.fonts'
  - **resource** - The file name of the resource - e.g. 'FiraMono-BoldItalic.ttf'

Second using paths:

- "**regular_path**" - The path to this font's file with no particular style applied.
- "**bold_path**" - The path to this font's file with bold style applied.
- "**italic_path**" - The path to this font's file with italic style applied.
- "**bold_italic_path**" - The path to this font's file with bold and italic style applied.

You only need to specify locations if this is the first use of this font name in the GUI.

Images

UIButton accepts images specified in the theme via an 'images' block. An 'images' block has these parameters:

- "**normal_image**" - The image displayed in the buttons default state. It has the following block of sub-parameters:
  - "**path**" - The string path to the image to be displayed. OR
  - "**package** - The name of the python package containing this resource - e.g. 'data.images'
  - "**resource** - The file name of the resource in the python package - e.g. 'splat.png' - Use a 'package' and 'resource' or a 'path' not both.
  - "**sub_surface_rect**" - An optional rectangle (described like "x,y,width,height") that will be used to grab a smaller portion of the image specified. This allows us to create many image surfaces from one image file.
- "**hovered_image**" - The image displayed in the buttons hovered state. It has the following block of sub-parameters:
  - "**path**" - The string path to the image to be displayed. OR
  - "**package** - The name of the python package containing this resource - e.g. 'data.images'
  - "**resource** - The file name of the resource in the python package - e.g. 'splat.png' - Use a 'package' and 'resource' or a 'path' not both.
  - "**sub_surface_rect**" - An optional rectangle (described like "x,y,width,height") that will be used to grab a smaller portion of the image specified. This allows us to create many image surfaces from one image file.
- "**selected_image**" - The image displayed in the buttons select focused state. It has the following block of sub-parameters:
  - "**path**" - The string path to the image to be displayed. OR
  - "**package** - The name of the python package containing this resource - e.g. 'data.images'
  - "**resource** - The file name of the resource in the python package - e.g. 'splat.png' - Use a 'package' and 'resource' or a 'path' not both.
  - "**sub_surface_rect**" - An optional rectangle (described like "x,y,width,height") that will be used to grab a smaller portion of the image specified. This allows us to create many image surfaces from one image file.
- "**disabled_image**" - The image displayed in the buttons disabled state. It has the following block of sub-parameters:
  - "**path**" - The string path to the image to be displayed. OR
  - "**package** - The name of the python package containing this resource - e.g. 'data.images'
  - "**resource** - The file name of the resource in the python package - e.g. 'splat.png' - Use a 'package' and 'resource' or a 'path' not both.
  - "**sub_surface_rect**" - An optional rectangle (described like "x,y,width,height") that will be used to grab a smaller portion of the image specified. This allows us to create many image surfaces from one image file.

Misc

UIButton accepts the following miscellaneous parameters in a 'misc' block:

- "**shape**" - Can be one of 'rectangle', 'rounded_rectangle' or 'ellipse'. Different shapes for this UI element.
- "**shape_corner_radius**" - Only used if our shape is 'rounded_rectangle'. It sets the radius, or radii, used for the rounded corners. Use a single integer to set all corners to the same radius, or four integers separated by commas to set each corner individually.
- "**border_width**" - the width in pixels of the border around the button. Defaults to 1.
- "**shadow_width**" - the width in pixels of the shadow behind the button. Defaults to 2.
- "**tool_tip_delay**" - time in seconds before the button's tool tip (if it has one) will appear. Default is "1.0".
- "**text_horiz_alignment**" - Set to "left", "right" or "center". Controls the horizontal placement of the button text, if this button has any text. Default is "center".
- "**text_vert_alignment**" - Set to "top", "bottom or "center". Controls the vertical placement of the button text, if this button has any text. Default is "center".
- "**text_horiz_alignment_padding**" - If horizontal alignment is set to 'left' or 'right' this value will control the buffer between the edge of the button and where we start placing the text. Default is "1".
- "**text_vert_alignment_padding**" - If vertical alignment is set to 'top' or 'bottom' this value will control the buffer between the edge of the button and where we start placing the text. Default is "1".
- "**text_shadow_size**" - The increased size in pixels of the text's shadow/outline. Set to "0", "1" or "2", larger than that the effect breaks down and individual letters merge together. Defaults to "0", no shadow.
- "**text_shadow_offset**" - Pixel offset in horizontal (x) and vertical (y) dimensions for where the text shadow is drawn. In the format "x,y". Defaults to "0,0".
- "**state_transitions**" - A block of parameters that define any fade transitions between button states. Normally buttons states visually change instantly, if you setup values here the button will instead fade from one state to the next. Transition definitions are one way, if you want to go in both directions, use two parameters. Transition parameters have this format:
  - "**startstate_targetstate**" - Can be set to any positive floating point value, representing the transition time in seconds.

Example

Here is an example of a button block in a JSON theme file using all the parameters described above.

Listing 12: button.json

```
{
    "button":
    {
        "colours":
        {
            "normal_bg": "#25292e",
            "hovered_bg": "#35393e",
            "disabled_bg": "#25292e",
            "selected_bg": "#25292e",
            "active_bg": "#193784",
            "normal_text": "#c5cbd8",
```

(continues on next page)

(continued from previous page)

```
            "hovered_text": "#FFFFFF",
            "selected_text": "#FFFFFF",
            "disabled_text": "#6d736f",
            "active_text": "#6d736f",
            "normal_border": "#AAAAAA",
            "hovered_border": "#B0B0B0",
            "disabled_border": "#808080",
            "selected_border": "#8080B0",
            "active_border": "#8080B0",
            "normal_text_shadow": "#10101070",
            "hovered_text_shadow": "#10101070",
            "disabled_text_shadow": "#10101070",
            "selected_text_shadow": "#10101070",
            "active_text_shadow": "#10101070"
        },
        "font":
        {
            "name": "montserrat",
            "size": "12",
            "bold": "0",
            "italic": "1",
            "regular_resource": {
                 "package": "data.fonts",
                 "resource": "Montserrat-Regular.ttf"
            },
            "bold_resource": {
                 "package": "data.fonts",
                 "resource": "Montserrat-Bold.ttf"
            },
            "italic_resource": {
                 "package": "data.fonts",
                 "resource": "Montserrat-Italic.ttf"
            },
            "bold_italic_resource": {
                 "package": "data.fonts",
                 "resource": "Montserrat-BoldItalic.ttf"
            },
        },
        "images":
        {
            "normal_image": {
                "package": "data.images",
                "resource": "buttons.png",
                "sub_surface_rect": "0,0,32,32"
            },
            "hovered_image": {
                "package": "data.images",
                "resource": "buttons.png",
                "sub_surface_rect": "32,0,32,32"
            },
            "selected_image": {
                "package": "data.images",
```

(continues on next page)

(continued from previous page)

```
                "resource": "buttons.png",
                "sub_surface_rect": "64,0,32,32"
            },
            "disabled_image": {
                "package": "data.images",
                "resource": "buttons.png",
                "sub_surface_rect": "96,0,32,32"
            }

        },
        "misc":
        {
            "shape": "rounded_rectangle",
            "shape_corner_radius": "10,0,0,0",
            "border_width": "1",
            "shadow_width": "1",
            "tool_tip_delay": "1.0",
            "text_horiz_alignment": "left",
            "text_vert_alignment": "top",
            "text_horiz_alignment_padding": "10",
            "text_vert_alignment_padding": "5",
            "text_shadow_size": "1",
            "text_shadow_offset": "0,0",
            "state_transitions":
            {
                "normal_hovered": "0.5",
                "hovered_normal": "0.5"
            }
        }
    }
}
```

UIDropDownMenu Theming Parameters

The *UIDropDownMenu* theming block id is 'drop_down_menu'.

Colours

UIDropDownMenu makes use of these colour parameters in a 'colours' block. All of these colours can also be a colour gradient:

- "**dark_bg**" - The background colour/gradient of the drop down menu. Probably not visible.
- "**normal_border**" - The border colour/gradient of the drop down menu.
- "**disabled_dark_bg**" - The colour/gradient of the menu background when disabled.
- "**disabled_border**" - The border colour/gradient of the menu when disabled.

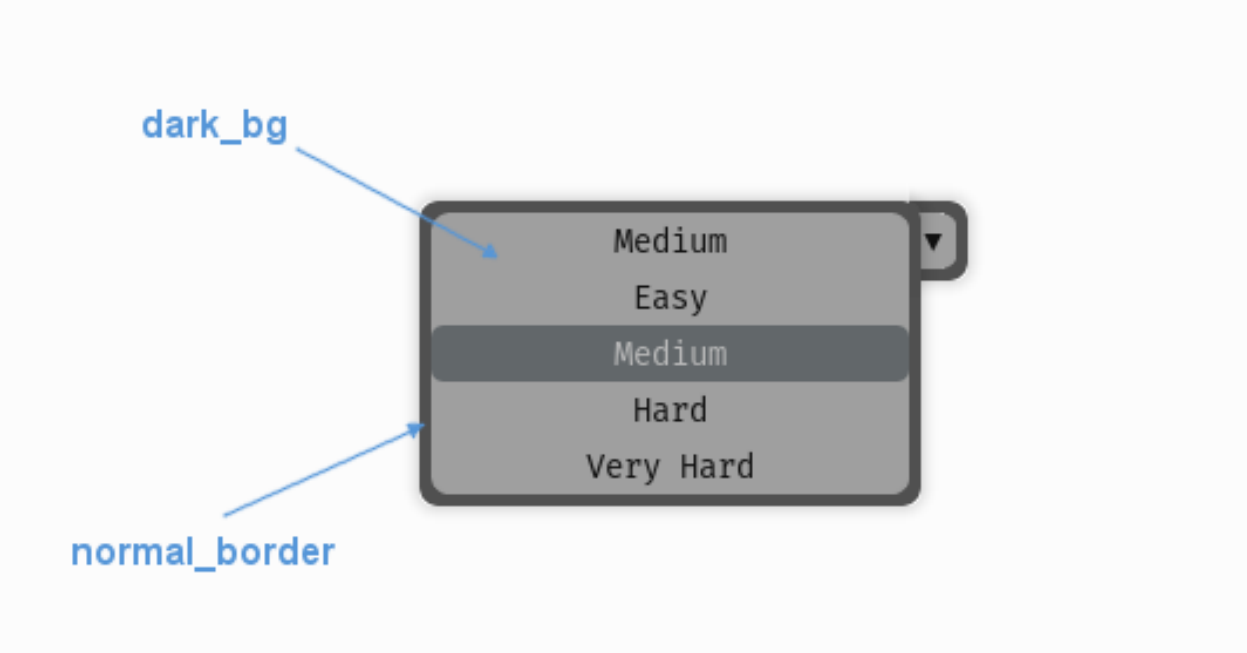


Fig. 2: A diagram of which part of the element is themed by which colour parameter. In the drop down menu it will only be possible to see the 'dark_bg' themed regions when the buttons in the menu are themed to be transparent, or semi transparent themselves.

Misc

UIDropDownMenu accepts the following miscellaneous parameters in a 'misc' block:

- "**shape**" - Can be one of 'rectangle' or 'rounded_rectangle'. Different shapes for this UI element.
- "**shape_corner_radius**" - Only used if our shape is 'rounded_rectangle'. It sets the radius, or radii, used for the rounded corners. Use a single integer to set all corners to the same radius, or four integers separated by commas to set each corner individually.
- "**expand_direction**" - Can be set to **'up'** or **'down'**. Defaults to 'down'. Changing this parameter will change the direction that the menu will expand away from the initial starting point.
- "**border_width**" - the width in pixels of the border around the drop down menu. Defaults to "1".
- "**shadow_width**" - the width in pixels of the shadow behind the button. Defaults to "1".
- "**open_button_width**" - the width of the open/close button on the right hand side of the drop down. Defaults to "20" (pixels). Set it to "0" to remove the open/close button.
- "**tool_tip_delay**" - time in seconds before the button's tool tip (if it has one) will appear. Default is "1.0".

Sub-elements

You can reference all of the buttons that are sub elements of the drop down menu with a theming block id of 'drop_down_menu.button'. You can also reference the buttons individually by adding their object IDs:

- 'drop_down_menu.#expand_button'
- 'drop_down_menu.#selected_option'

There is a also a selection list that manages the options when the drop down is expanded, you can reference that with:

- 'drop_down_menu.#drop_down_options_list'

Or the individual option buttons with:

- 'drop_down_menu.#drop_down_options_list.button'

There is more information on theming buttons at *UIButton Theming Parameters* and selection lists at :ref: *theme-selection-list*.

Example

Here is an example of a drop down menu block in a JSON theme file, using the parameters described above (and a couple from UIButton).

Listing 13: drop_down_menu.json

```
{
    "drop_down_menu":
    {
        "misc":
        {
            "expand_direction": "down"
        },

        "colours":
        {
            "normal_bg": "#25292e",
            "hovered_bg": "#35393e"
        }
    },
    "drop_down_menu.#selected_option":
    {
        "misc":
        {
           "border_width": "1",
           "open_button_width": "10"
        }
    }
}
```

UIHorizontalScrollBar Theming Parameters

The *UIHorizontalScrollBar* theming block id is 'horizontal_scroll_bar'.

Colours

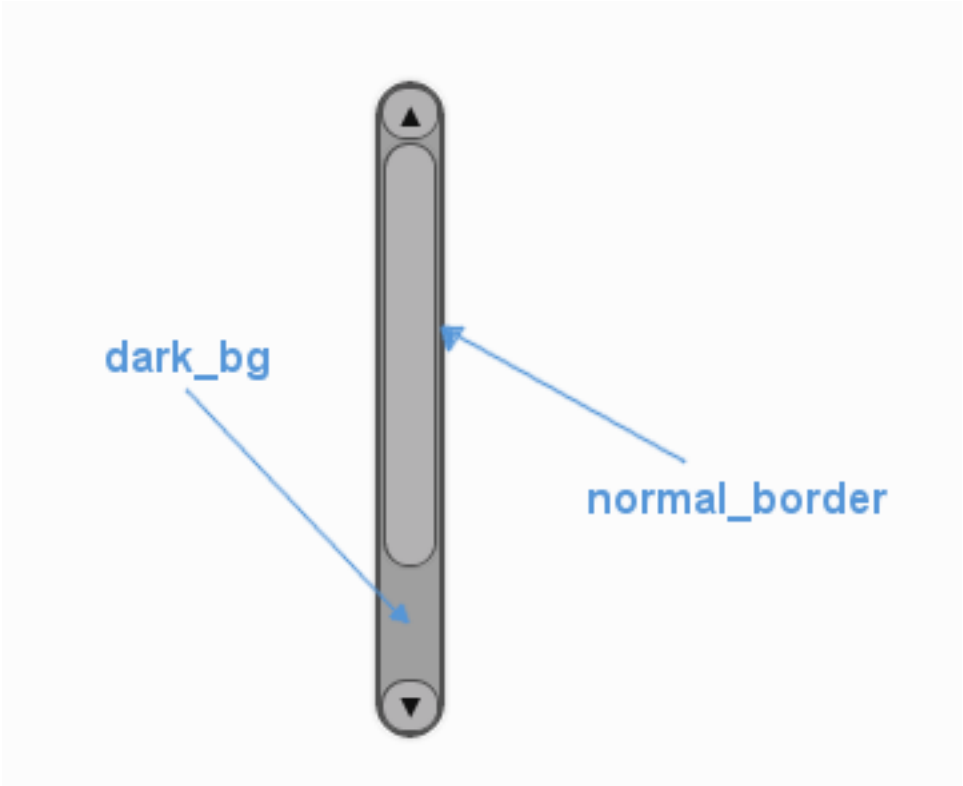


Fig. 3: A diagram of which part of the element is themed by which colour parameter. The scroll bar's buttons are themed in a separate block.

UIHorizontalScrollBar makes use of these colour parameters in a 'colours' block. All of these colours can also be a colour gradient:

- "**dark_bg**" - The background colour/gradient of the 'back' of the scroll bar, the colour of the track that the scroll bar moves along.
- "**normal_border**" - The colour/gradient of the border around the scroll bar.
- "**disabled_dark_bg**" - The colour/gradient of the track when disabled.
- "**disabled_border**" - The border colour/gradient of the slider when disabled.

Misc

UIHorizontalScrollBar accepts the following miscellaneous parameters in a 'misc' block:

- "**shape**" - Can be one of 'rectangle' or 'rounded_rectangle'. Different shapes for this UI element.
- "**shape_corner_radius**" - Only used if our shape is 'rounded_rectangle'. It sets the radius, or radii, used for the rounded corners. Use a single integer to set all corners to the same radius, or four integers separated by commas to set each corner individually.
- "**border_width**" - the width in pixels of the border around the bar. Defaults to 1.
- "**shadow_width**" - the width in pixels of the shadow behind the bar. Defaults to 1.
- "**enable_arrow_buttons**" - Enables or disables the arrow buttons for the scroll bar. "1" is enabled, "0" is disabled. Defaults to "1".
- "**tool_tip_delay**" - time in seconds before the scroll bar's tool tip (if it has one) will appear. Default is "1.0".

Sub-elements

You can reference all three of the buttons that are sub elements of the scroll bar with a theming block id of 'horizontal_scroll_bar.button'. You can reference both of the arrow buttons with the class_id: '@arrow_button'. You can also reference the three buttons individually by adding their object IDs:

- 'horizontal_scroll_bar.#left_button'
- 'horizontal_scroll_bar.#right_button'
- 'horizontal_scroll_bar.#sliding_button'

There is more information on theming buttons at *UIButton Theming Parameters*.

Example

Here is an example of some horizontal scroll bar blocks in a JSON theme file, using the parameters described above (and some from UIButton).

Listing 14: horizontal_scroll_bar.json

```
{
    "horizontal_scroll_bar":
    {
        "colours":
        {
            "normal_bg": "#25292e",
            "hovered_bg": "#35393e",
            "disabled_bg": "#25292e",
            "selected_bg": "#25292e",
            "active_bg": "#193784",
            "dark_bg": "#15191e",
            "normal_text": "#c5cbd8",
            "hovered_text": "#FFFFFF",
            "selected_text": "#FFFFFF",
            "disabled_text": "#6d736f"
        },
```

(continues on next page)

(continued from previous page)

```
        "misc":
        {
            "shape": "rectangle",
            "border_width": "0",
            "enable_arrow_buttons": "1"
        }
    },
    "horizontal_scroll_bar.button":
    {
        "misc":
        {
            "border_width": "1"
        }
    },
    "horizontal_scroll_bar.@arrow_button":
    {
        "misc":
        {
            "shadow_width": "0"
        }
    },
    "horizontal_scroll_bar.#sliding_button":
    {
        "colours":
        {
            "normal_bg": "#FF0000"
        }
    }
}
```

UIHorizontalSlider Theming Parameters

The *UIHorizontalSlider* theming block id is 'horizontal_slider'.

Colours

UIHorizontalSlider makes use of these colour parameters in a 'colours' block. All of these colours can also be a colour gradient:

- "**dark_bg**" - The background colour/gradient of the 'back' of the slider, the colour of the track that the sliding part moves along.
- "**normal_border**" - The border colour/gradient of the slider.
- "**disabled_dark_bg**" - The colour/gradient of the track when disabled.
- "**disabled_border**" - The border colour/gradient of the slider when disabled.

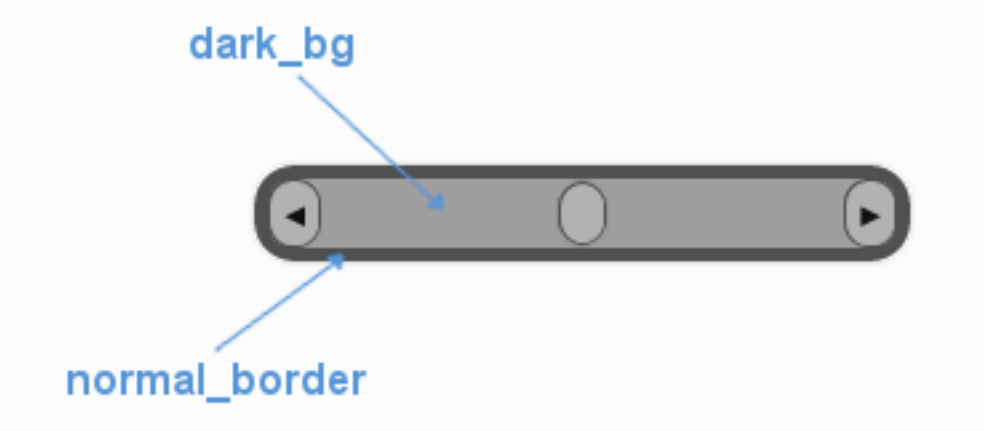


Fig. 4: A diagram of which part of the element is themed by which colour parameter. The slider's buttons are themed in a separate block.

Misc

UIHorizontalSlider accepts the following miscellaneous parameters in a 'misc' block:

- "**shape**" - Can be one of 'rectangle' or 'rounded_rectangle'. Different shapes for this UI element.
- "**shape_corner_radius**" - Only used if our shape is 'rounded_rectangle'. It sets the radius, or radii, used for the rounded corners. Use a single integer to set all corners to the same radius, or four integers separated by commas to set each corner individually.
- "**border_width**" - the width in pixels of the border around the slider. Defaults to 1.
- "**shadow_width**" - the width in pixels of the shadow behind the slider. Defaults to 1.
- "**enable_arrow_buttons**" - Enables or disables the arrow buttons for the slider. "1" is enabled, "0" is disabled. Defaults to "1".
- "**sliding_button_width**" - Sets the width of the sliding button. Defaults to "20".
- "**tool_tip_delay**" - time in seconds before the button's tool tip (if it has one) will appear. Default is "1.0".

Sub-elements

You can reference all of the buttons that are sub elements of the slider with a theming block id of 'horizontal_slider.button'. You can reference both of the arrow buttons with the class_id: '@arrow_button'. You can also reference the buttons individually by adding their object IDs:

- 'horizontal_slider.#left_button'
- 'horizontal_slider.#right_button'
- 'horizontal_slider.#sliding_button'

There is more information on theming buttons at *UIButton Theming Parameters*.

Example

Here is an example of a horizontal slider block in a JSON theme file, using the parameters described above (and some from UIButton).

Listing 15: horizontal_slider.json

```
{
    "horizontal_slider":
    {
        "colours":
        {
            "normal_bg": "#25292e",
            "hovered_bg": "#35393e",
            "disabled_bg": "#25292e",
            "selected_bg": "#25292e",
            "active_bg": "#193784",
            "dark_bg": "#15191e,#202020,0",
            "normal_text": "#c5cbd8",
            "hovered_text": "#FFFFFF",
            "selected_text": "#FFFFFF",
            "disabled_text": "#6d736f"
        },
        "misc":
        {
           "shape": "rectangle",
           "enable_arrow_buttons": "0",
           "sliding_button_width": "15"
        }
    },
    "horizontal_slider.button":
    {
        "misc":
        {
           "border_width": "1"
        }
    },
    "horizontal_slider.#sliding_button":
    {
        "colours":
        {
           "normal_bg": "#FF0000"
        }
    }
}
```

UIImage Theming Parameters

UIImage currently has no theming parameters at all. How sad.

UILabel Theming Parameters

The *UILabel* theming block id is 'label'.

Colours

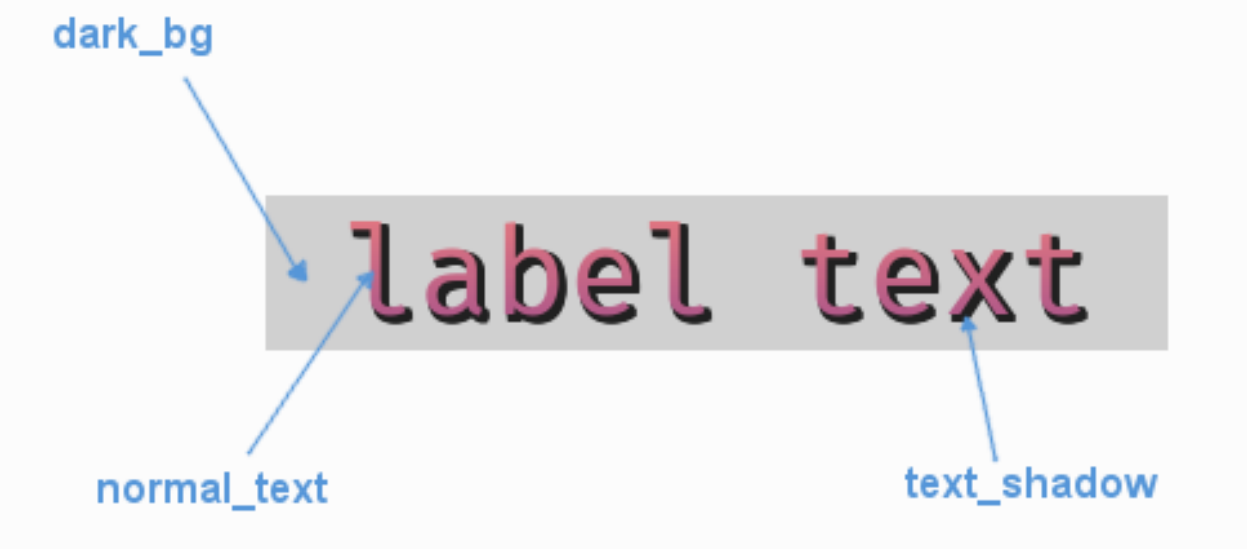


Fig. 5: A diagram of which part of the element is themed by which colour parameter. The text shadow **cannot** be themed with a colour gradient.

UILabel makes use of these colour parameters in a 'colours' block. Most of these colours can also be a colour gradient:

- "**dark_bg**" - The background colour/gradient of the label text.
- "**normal_text**" - The colour/gradient of the text itself.
- "**disabled_text**" - The colour/gradient of the text when the label has been disabled.
- "**text_shadow**" - The colour of the shadow behind the text, if any exists.

Font

UILabel accepts a font specified in the theme via a 'font' block. A 'font' block has these parameters:

- "**name**" - Necessary to make a valid block. This is the name that this font goes by in the UI, if this is a new font then subsequent font instances with different styles or sizes should use the same name.
- "**locale**" - Optional parameter to set this font as belonging to a particular locale only. See the *Localization* guide. You will need to keep repeating the locale specifier if using prototypes to make a hierarchy.
- "**size**" - Necessary to make a valid block. This is the point size of the font to use on the label.
- "**bold**" - Optional parameter. Set it to "1" to make this font bold.

- "**italic**" - Optional parameter. Set it to "1" to make this font italic.

There are two methods to refer to font resource locations. First, using packaged resources:

- **"regular_resource - The location of this font's file with no particular style applied.**
  - **package** - The name of the python package containing this resource - e.g. 'data.fonts'
  - **resource** - The file name of the resource - e.g. 'FiraCode-Regular.ttf'
- **"bold_resource" - The location of this font's file with bold style applied.**
  - **package** - The name of the python package containing this resource - e.g. 'data.fonts'
  - **resource** - The file name of the resource - e.g. 'FiraCode-Bold.ttf'
- **"italic_resource" - The location of this font's file with italic style applied.**
  - **package** - The name of the python package containing this resource - e.g. 'data.fonts'
  - **resource** - The file name of the resource - e.g. 'FiraMono-Italic.ttf'
- **"bold_italic_resource" - The location of this font's file with bold and italic style applied.**
  - **package** - The name of the python package containing this resource - e.g. 'data.fonts'
  - **resource** - The file name of the resource - e.g. 'FiraMono-BoldItalic.ttf'

Second using paths:

- "**regular_path**" - The path to this font's file with no particular style applied.
- "**bold_path**" - The path to this font's file with bold style applied.
- "**italic_path**" - The path to this font's file with italic style applied.
- "**bold_italic_path**" - The path to this font's file with bold and italic style applied.

You only need to specify locations if this is the first use of this font name in the GUI.

Misc

UILabel has the following miscellaneous parameters in a 'misc' block:

- "**text_shadow_size**" - The increased size in pixels of the shadow/outline. Set to "0", "1" or "2", larger than that the effect breaks down and individual letters merge together. Defaults to "0", no shadow.
- "**text_shadow_offset**" - Pixel offset in horizontal (x) and vertical (y) dimensions for where the shadow is drawn. In the format "x,y". Defaults to "0,0".
- "**text_horiz_alignment**" - Set to "left", "right" or "center". Controls the horizontal placement of the label text. Default is "center".
- "**text_vert_alignment**" - Set to "top", "bottom or "center". Controls the vertical placement of the label text. Default is "center".
- "**text_horiz_alignment_padding**" - If horizontal alignment is set to 'left' or 'right' this value will control the buffer between the edge of the label rectangle and where we start placing the text. Default is "0".
- "**text_vert_alignment_padding**" - If vertical alignment is set to 'top' or 'bottom' this value will control the buffer between the edge of the label rectangle and where we start placing the text. Default is "0".
- "**tool_tip_delay**" - time in seconds before the button's tool tip (if it has one) will appear. Default is "1.0".

Example

Here is an example of a label block in a JSON theme file using the parameters described above.

Listing 16: label.json

```
{
    "label":
    {
        "colours":
        {
            "dark_bg": "#25292e",
            "normal_text": "#c5cbd8",
            "text_shadow": "#505050"
        },
        "font":
        {
            "name": "montserrat",
            "size": "12",
            "bold": "0",
            "italic": "0"
        },
        "misc":
        {
            "text_shadow": "1",
            "text_shadow_size": "1",
            "text_shadow_offset": "0,0"
        }
    }
}
```

UIPanel Theming Parameters

The *UIPanel* theming block id is 'panel'.

Colours

UIPanel makes use of the following these colour parameters in a 'colours' block. All of these colours can also be a colour gradient:

- "**dark_bg**" - The background colour/gradient of the panel element.
- "**normal_border**" - The colour/gradient of the border around the panel (if it has one).

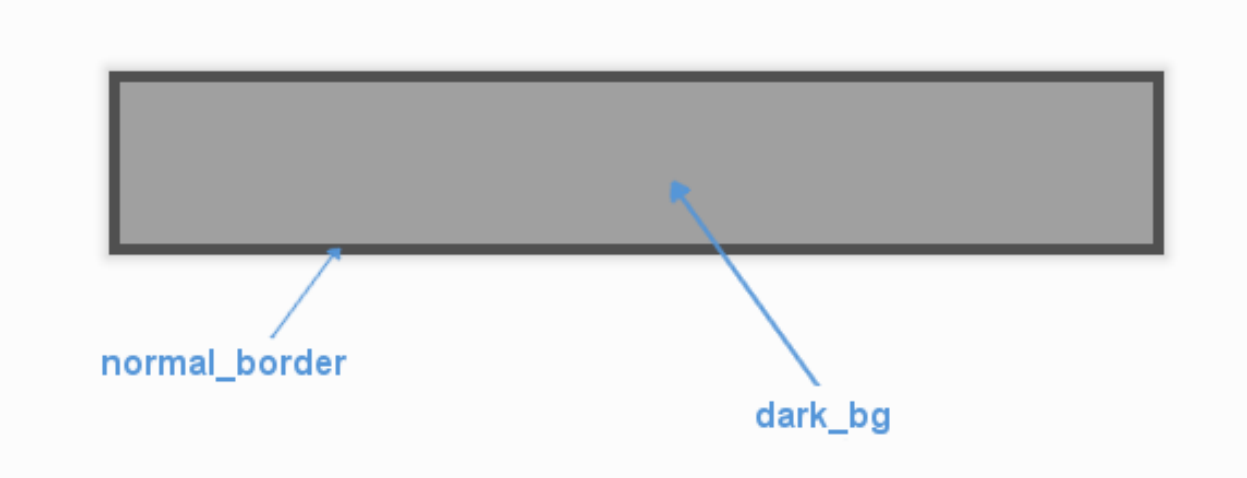


Fig. 6: A diagram of which part of the element is themed by which colour parameter.

Images

UIPanel accepts an image specified in the theme via an 'images' block. An 'images' block can have these parameters:

- "**background_image**" - The image displayed on the panel. It has the following block of sub-parameters:
  - "**path**" - The string path to the image to be displayed. OR
  - "**package** - The name of the python package containing this resource - e.g. 'data.images'
  - "**resource** - The file name of the resource in the python package - e.g. 'splat.png' - Use a 'package' and 'resource' or a 'path' not both.
  - "**sub_surface_rect**" - An optional rectangle (described like "x,y,width,height") that will be used to grab a smaller portion of the image specified. This allows us to create many image surfaces from one image file.

Misc

UIPanel accepts the following miscellaneous parameters in a 'misc' block:

- "**shape**" - Can be one of 'rectangle' or 'rounded_rectangle'. Different shapes for this UI element.
- "**shape_corner_radius**" - Only used if our shape is 'rounded_rectangle'. It sets the radius, or radii, used for the rounded corners. Use a single integer to set all corners to the same radius, or four integers separated by commas to set each corner individually.
- "**border_width**" - The width of the border around the element in pixels. Defaults to "1".
- "**shadow_width**" - The width of the shadow around the element in pixels. Defaults to "2".
- "**tool_tip_delay**" - time in seconds before the button's tool tip (if it has one) will appear. Default is "1.0".

Example

Here is an example of a panel block in a JSON theme file, using the parameters described above.

Listing 17: panel.json

```
{
    "panel":
    {
        "colours":
        {
            "dark_bg":"#21282D",
            "normal_border": "#999999"
        },

        "background_image":
        {
            "package": "data.images",
            "resource": "splat.png",
            "sub_surface_rect": "0,0,32,32"
        },

        "misc":
        {
            "shape": "rounded_rectangle",
            "shape_corner_radius": "10",
            "border_width": "1",
            "shadow_width": "15"
        }
    }
}
```

UIProgressBar Theming Parameters

The *UIProgressBar* theming block id is 'progress_bar'.

Colours

UIProgressBar makes use of these colour parameters in a 'colours' block. Most of these colours can also be a colour gradient:

- "**normal_text**" - The colour/gradient of the health bar's text.
- "**text_shadow**" - The colour of the shadow behind the text (so it stands out better).
- "**normal_border**" - The colour/gradient of the border around the health bar.
- "**filled_bar**" - The colour/gradient of the actual bar itself, of the portion of it that is still full.
- "**unfilled_bar**" - The colour/gradient of an empty portion of the health bar.

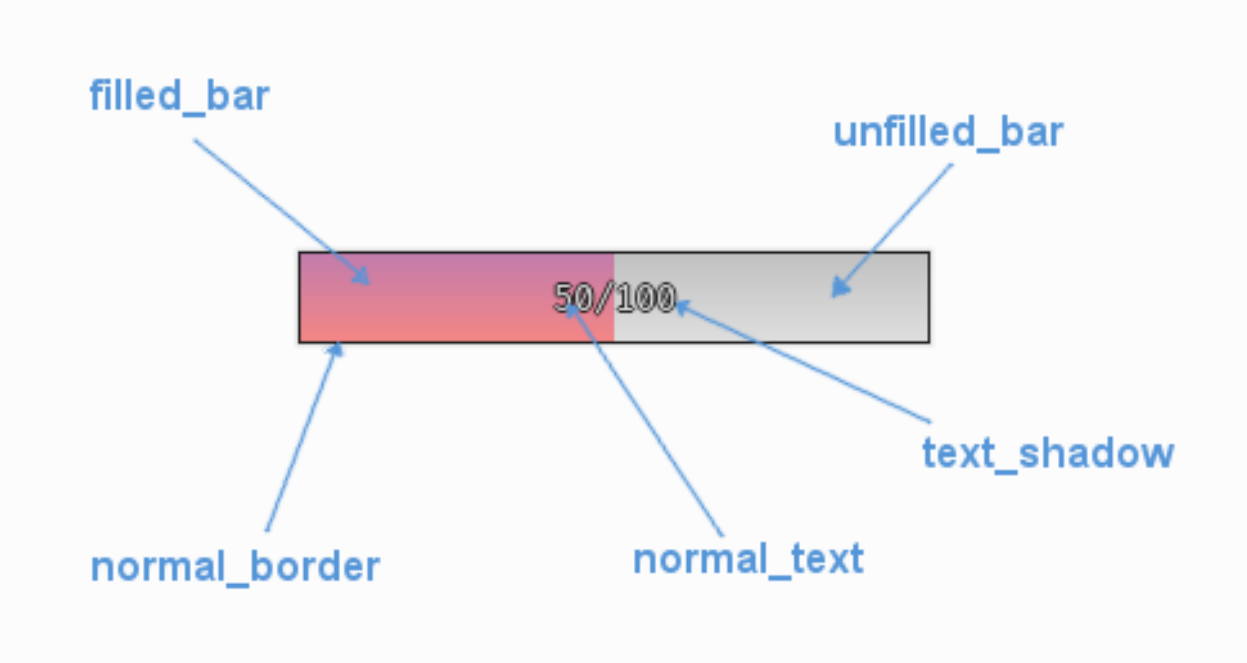


Fig. 7: A diagram of which part of the element is themed by which colour parameter. The text shadow **cannot** be themed with a colour gradient.

Misc

UIProgressBar accepts the following miscellaneous parameters in a 'misc' block:

- "**shape**" - Can be one of 'rectangle' or 'rounded_rectangle'. Different shapes for this UI element.
- "**shape_corner_radius**" - Only used if our shape is 'rounded_rectangle'. It sets the radius, or radii, used for the rounded corners. Use a single integer to set all corners to the same radius, or four integers separated by commas to set each corner individually.
- "**border_width**" - the width in pixels of the border around the bar. Defaults to 1.
- "**shadow_width**" - the width in pixels of the shadow behind the bar. Defaults to 1.
- "**tool_tip_delay**" - time in seconds before the button's tool tip (if it has one) will appear. Default is "1.0".

Font

UIProgressBar accepts a font specified in the theme via a 'font' block. A 'font' block has these parameters:

- "**name**" - Necessary to make a valid block. This is the name that this font goes by in the UI, if this is a new font then subsequent font instances with different styles or sizes should use the same name.
- "**locale**" - Optional parameter to set this font as belonging to a particular locale only. See the *Localization* guide. You will need to keep repeating the locale specifier if using prototypes to make a hierarchy.
- "**size**" - Necessary to make a valid block. This is the point size of the font to use on the health bar.
- "**bold**" - Optional parameter. Set it to "1" to make this font bold.
- "**italic**" - Optional parameter. Set it to "1" to make this font italic.

There are two methods to refer to font resource locations. First, using packaged resources:

- **"regular_resource - The location of this font's file with no particular style applied.**
  - **package** - The name of the python package containing this resource - e.g. 'data.fonts'
  - **resource** - The file name of the resource - e.g. 'FiraCode-Regular.ttf'
- **"bold_resource" - The location of this font's file with bold style applied.**
  - **package** - The name of the python package containing this resource - e.g. 'data.fonts'
  - **resource** - The file name of the resource - e.g. 'FiraCode-Bold.ttf'
- **"italic_resource" - The location of this font's file with italic style applied.**
  - **package** - The name of the python package containing this resource - e.g. 'data.fonts'
  - **resource** - The file name of the resource - e.g. 'FiraMono-Italic.ttf'
- **"bold_italic_resource" - The location of this font's file with bold and italic style applied.**
  - **package** - The name of the python package containing this resource - e.g. 'data.fonts'
  - **resource** - The file name of the resource - e.g. 'FiraMono-BoldItalic.ttf'

Second using paths:

- "**regular_path**" - The path to this font's file with no particular style applied.
- "**bold_path**" - The path to this font's file with bold style applied.
- "**italic_path**" - The path to this font's file with italic style applied.
- "**bold_italic_path**" - The path to this font's file with bold and italic style applied.

You only need to specify locations if this is the first use of this font name in the GUI.

Example

Here is an example of a progress bar block in a JSON theme file using the parameters described above.

Listing 18: progress_bar.json

```
{
    "progress_bar":
    {
        "colours":
        {
            "normal_text": "#c5cbd8",
            "text_shadow": "#777777",
            "normal_border": "#DDDDDD",
            "filled_bar": "#f4251b,#A4150b,180",
            "unfilled_bar": "#CCCCCC"
        },
        "font":
        {
            "name": "montserrat",
            "size": "12",
            "bold": "0",
            "italic": "1"
        }
```

(continues on next page)

(continued from previous page)

```
19      }
20  }
```

UIScreenSpaceHealthBar Theming Parameters

The *UIScreenSpaceHealthBar* theming block id is 'screen_space_health_bar'.

Colours

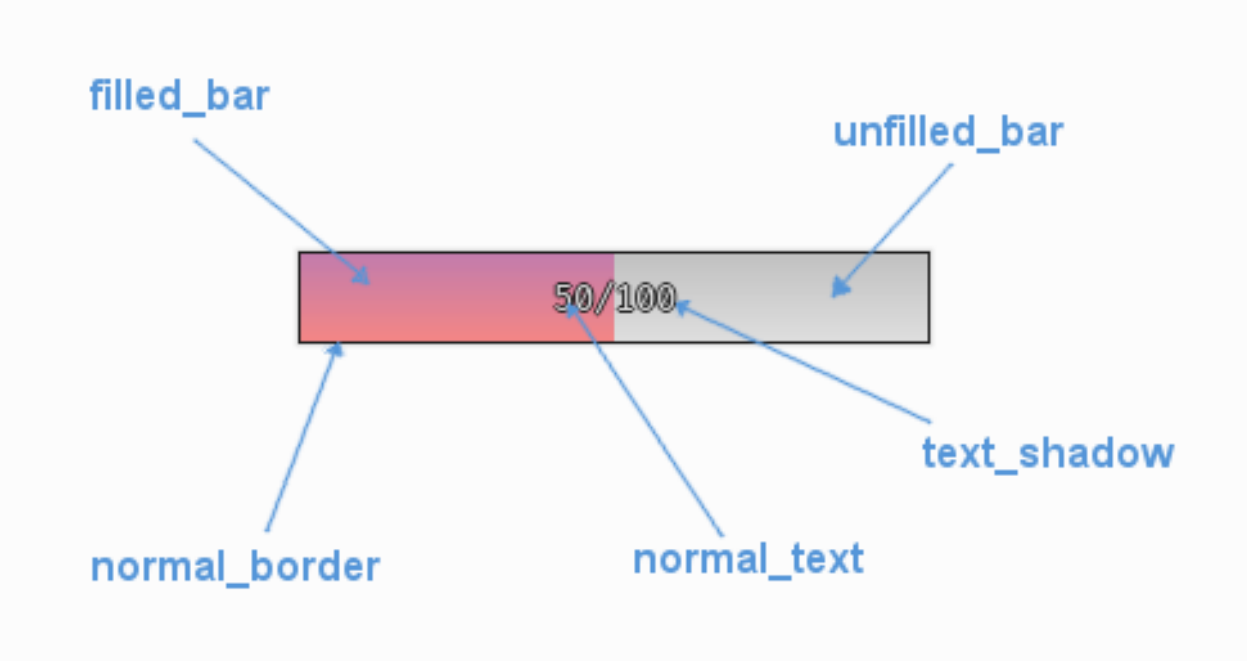


Fig. 8: A diagram of which part of the element is themed by which colour parameter. The text shadow **cannot** be themed with a colour gradient.

UIScreenSpaceHealthBar makes use of these colour parameters in a 'colours' block. Most of these colours can also be a colour gradient:

- "**normal_text**" - The colour/gradient of the health bar's text.
- "**text_shadow**" - The colour of the shadow behind the text (so it stands out better).
- "**normal_border**" - The colour/gradient of the border around the health bar.
- "**filled_bar**" - The colour/gradient of the actual bar itself, of the portion of it that is still full.
- "**unfilled_bar**" - The colour/gradient of an empty portion of the health bar.

Misc

UIScreenSpaceHealthBar accepts the following miscellaneous parameters in a 'misc' block:

- "**shape**" - Can be one of 'rectangle' or 'rounded_rectangle'. Different shapes for this UI element.
- "**shape_corner_radius**" - Only used if our shape is 'rounded_rectangle'. It sets the radius, or radii, used for the rounded corners. Use a single integer to set all corners to the same radius, or four integers separated by commas to set each corner individually.
- "**border_width**" - the width in pixels of the border around the bar. Defaults to 1.
- "**shadow_width**" - the width in pixels of the shadow behind the bar. Defaults to 1.
- "**tool_tip_delay**" - time in seconds before the button's tool tip (if it has one) will appear. Default is "1.0".

Font

UIScreenSpaceHealthBar accepts a font specified in the theme via a 'font' block. A 'font' block has these parameters:

- "**name**" - Necessary to make a valid block. This is the name that this font goes by in the UI, if this is a new font then subsequent font instances with different styles or sizes should use the same name.
- "**locale**" - Optional parameter to set this font as belonging to a particular locale only. See the *Localization* guide. You will need to keep repeating the locale specifier if using prototypes to make a hierarchy.
- "**size**" - Necessary to make a valid block. This is the point size of the font to use on the health bar.
- "**bold**" - Optional parameter. Set it to "1" to make this font bold.
- "**italic**" - Optional parameter. Set it to "1" to make this font italic.

There are two methods to refer to font resource locations. First, using packaged resources:

- **"regular_resource - The location of this font's file with no particular style applied.**
  - **package** - The name of the python package containing this resource - e.g. 'data.fonts'
  - **resource** - The file name of the resource - e.g. 'FiraCode-Regular.ttf'
- **"bold_resource" - The location of this font's file with bold style applied.**
  - **package** - The name of the python package containing this resource - e.g. 'data.fonts'
  - **resource** - The file name of the resource - e.g. 'FiraCode-Bold.ttf'
- **"italic_resource" - The location of this font's file with italic style applied.**
  - **package** - The name of the python package containing this resource - e.g. 'data.fonts'
  - **resource** - The file name of the resource - e.g. 'FiraMono-Italic.ttf'
- **"bold_italic_resource" - The location of this font's file with bold and italic style applied.**
  - **package** - The name of the python package containing this resource - e.g. 'data.fonts'
  - **resource** - The file name of the resource - e.g. 'FiraMono-BoldItalic.ttf'

Second using paths:

- "**regular_path**" - The path to this font's file with no particular style applied.
- "**bold_path**" - The path to this font's file with bold style applied.
- "**italic_path**" - The path to this font's file with italic style applied.
- "**bold_italic_path**" - The path to this font's file with bold and italic style applied.

You only need to specify locations if this is the first use of this font name in the GUI.

Example

Here is an example of a screen space health bar block in a JSON theme file using the parameters described above.

Listing 19: screen_space_health_bar.json

```
{
    "screen_space_health_bar":
    {
        "colours":
        {
            "normal_text": "#c5cbd8",
            "text_shadow": "#777777",
            "normal_border": "#DDDDDD",
            "filled_bar": "#f4251b,#A4150b,180",
            "unfilled_bar": "#CCCCCC"
        },
        "font":
        {
            "name": "montserrat",
            "size": "12",
            "bold": "0",
            "italic": "1"
        }
    }
}
```

UIScrollingContainer Theming Parameters

UIScrollingContainer currently has no theming parameters at all. How sad.

Sub-elements

You can reference the two scrollbars of the scrolling container by adding their element IDs:

- 'scrolling_container.vertical_scroll_bar'
- 'scrolling_container.horizontal_scroll_bar'

There is more information on theming vertical scroll bars at *UIVerticalScrollBar Theming Parameters* and horizontal scroll bars at *UIHorizontalScrollBar Theming Parameters*.

UISelectionList Theming Parameters

UISelectionList theming block id is 'selection_list'.

Colours

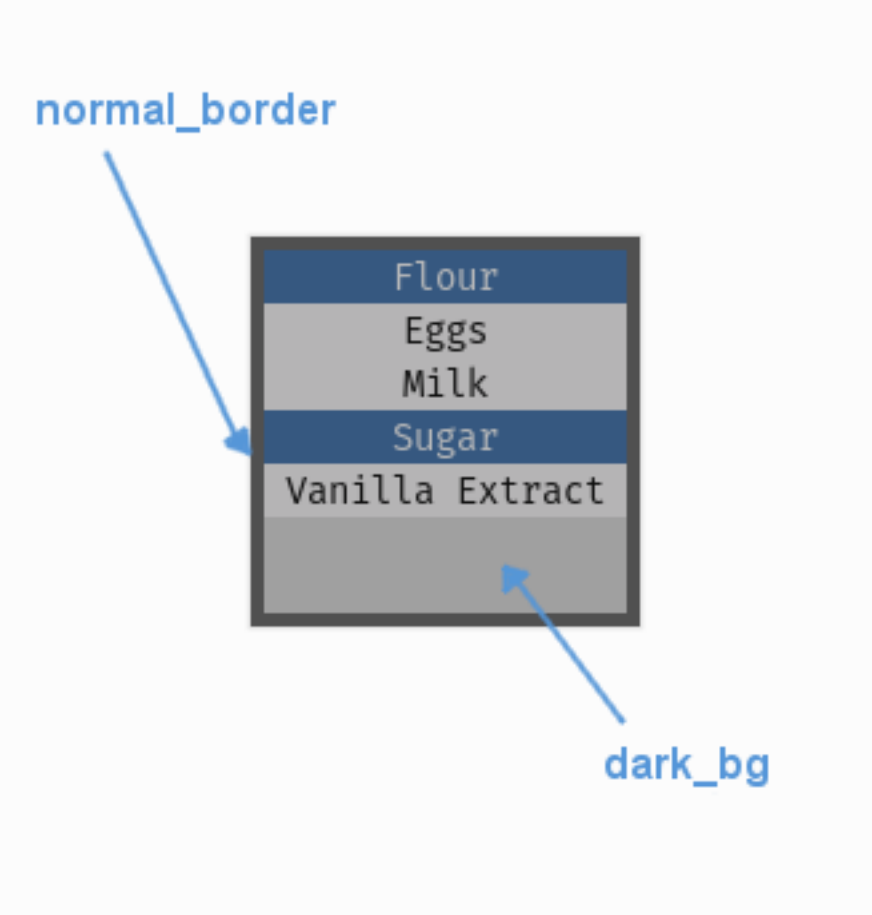


Fig. 9: A diagram of which part of the element is themed by which colour parameter.

UISelectionList makes use of the following these colour parameters in a 'colours' block. All of these colours can also be a colour gradient:

- "**dark_bg**" - The background colour/gradient of the selection list element. Often you won't see this unless your list buttons are semi-transparent.
- "**normal_border**" - The colour/gradient of the border around the selection list (if it has one).

Misc

UISelectionList accepts the following miscellaneous parameters in a 'misc' block:

- "**shape**" - Can be one of 'rectangle' or 'rounded_rectangle'. Different shapes for this UI element.
- "**shape_corner_radius**" - Only used if our shape is 'rounded_rectangle'. It sets the radius, or radii, used for the rounded corners. Use a single integer to set all corners to the same radius, or four integers separated by commas to set each corner individually.
- "**border_width**" - The width of the border around the element in pixels. Defaults to "1".
- "**shadow_width**" - The width of the shadow around the element in pixels. Defaults to "2".
- "**list_item_height**" - The pixel height of a the items in the list. Defaults to "20".
- "**tool_tip_delay**" - time in seconds before the button's tool tip (if it has one) will appear. Default is "1.0".

Sub-elements

The selection list may also contain a *UIVerticalScrollBar* which you can reference with the block id 'window.vertical_scroll_bar'. You can also reference all of the buttons that are sub elements of the scroll bar with a theming block id of 'window.vertical_scroll_bar.button', or pick out the list item buttons specifically with a class_id of '@selection_list_item'.

You can further reference the individual buttons of the scroll bar by adding their object IDs:

- 'window.vertical_scroll_bar.#top_button'
- 'window.vertical_scroll_bar.#bottom_button'
- 'window.vertical_scroll_bar.#sliding_button'

There is more information on theming the vertical scroll bar at *UIVerticalScrollBar Theming Parameters*.

Example

Here is an example of a panel block in a JSON theme file, using the parameters described above.

Listing 20: selection_list.json

```
{
    "selection_list":
    {
        "colours":
        {
            "dark_bg":"#21282D",
            "normal_border": "#999999"
        },

        "background_image":
        {
            "path": "data/images/splat.png",
            "sub_surface_rect": "0,0,32,32"
        },

        "misc":
```

(continues on next page)

(continued from previous page)

```
        {
            "shape": "rounded_rectangle",
            "shape_corner_radius": "10",
            "border_width": "1",
            "shadow_width": "15",
            "list_item_height": "30"
        }
    },
    "selection_list.@selection_list_item":
    {
        "misc":
        {
           "border_width": "2"
        }
    }
}
```

UIStatusBar Theming Parameters

The *UIStatusBar* theming block id is 'status_bar'.

Colours

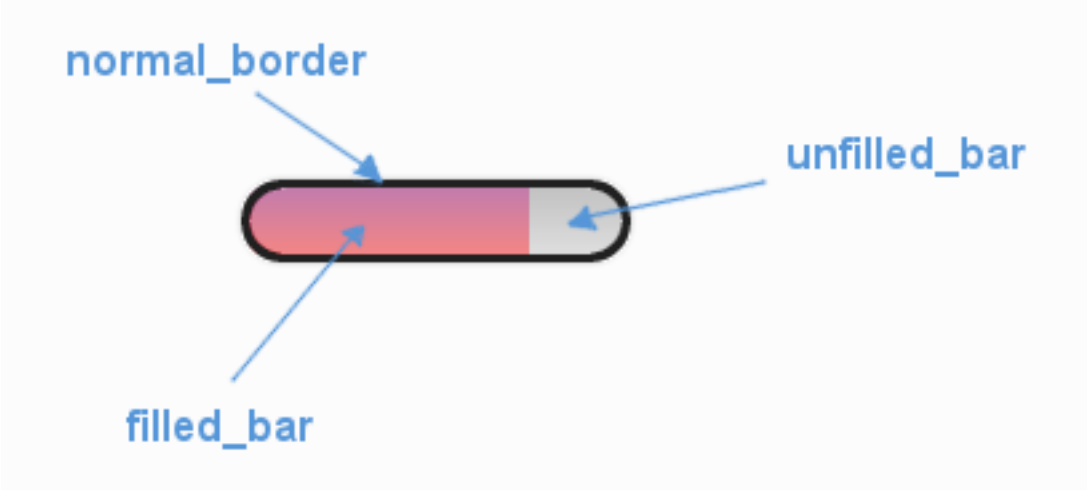


Fig. 10: A diagram of which part of the element is themed by which colour parameter.

UIStatusBar makes use of these colour parameters in a 'colours' block:

- "**normal_border**" - The colour/gradient of the health bar's border if it has one.
- "**filled_bar**" - The colour/gradient of the actual bar itself, of the portion of it that is still full.
- "**unfilled_bar**" - The colour/gradient of an empty portion of the health bar.

Misc

UIStatusBar has the following miscellaneous parameters in a 'misc' block:

- "**shape**" - Can be one of 'rectangle' or 'rounded_rectangle'. Different shapes for this UI element.
- "**shape_corner_radius**" - Only used if our shape is 'rounded_rectangle'. It sets the radius, or radii, used for the rounded corners. Use a single integer to set all corners to the same radius, or four integers separated by commas to set each corner individually.
- "**follow_sprite_offset**" - The x,y offset values for when the bar follows a sprite. Defaults to "0,0".
- "**border_width**" - The width of the border around the health bar. Defaults to "1". Can be "0" to remove the border.
- "**shadow_width**" - The width of the border around the health bar. Defaults to "1". Can be "0" to remove the border.
- "**tool_tip_delay**" - time in seconds before the button's tool tip (if it has one) will appear. Default is "1.0".

Example

Here is an example of a status bar block in a JSON theme file using the parameters described above.

Listing 21: status_bar.json

```
{
    "status_bar":
    {
        "colours":
        {
            "normal_border": "#AAAAAA",
            "filled_bar": "#f4251b",
            "unfilled_bar": "#CCCCCC"
        },
        "misc":
        {
            "follow_sprite_offset": "-5, 32",
            "border_width": "0"
        }
    }
}
```

UITextBox Theming Parameters

The *UITextBox* theming block id is 'text_box'.

Colours

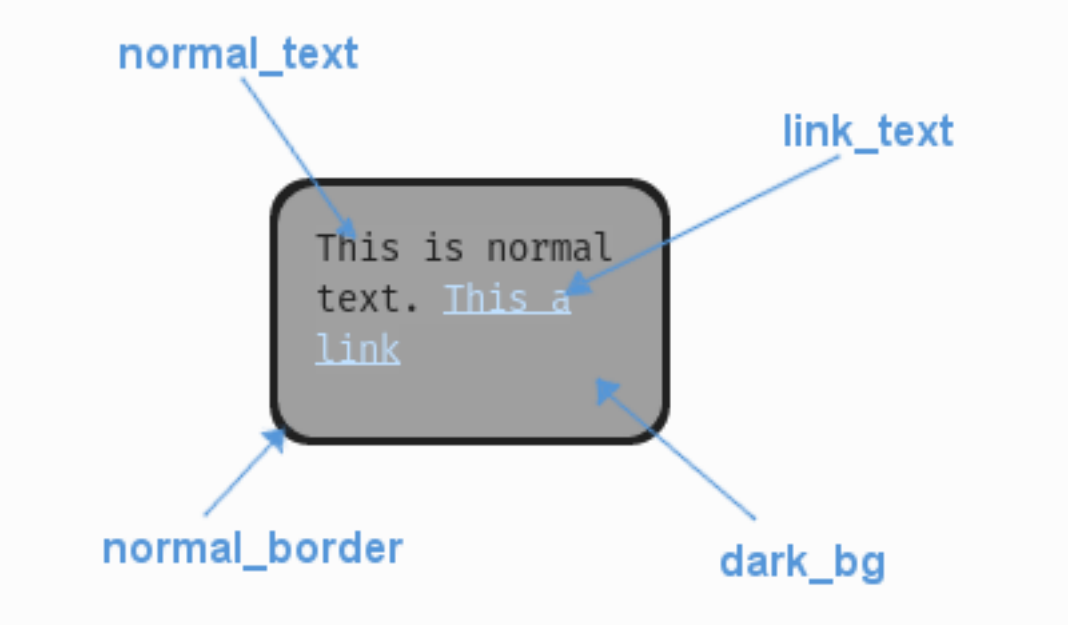


Fig. 11: A diagram of which part of the element is themed by which colour parameter. This correspondence is the same for the parameters named for the different link states e.g. the parameter 'link_hover' themes the same text as 'link_text' when the link is in the hovered state.

UITextBox makes use of these colour parameters in a 'colours' block. All of these colours can also be a colour gradient:

- "**dark_bg**" - The background colour/gradient of the text box element.
- "**selected_bg**" - The colour/gradient the background changes to when the text above it is selected.
- "**normal_border**" - The colour/gradient of the border around the text box element.
- "**selected_text**" - The colour/gradient of text when it has been selected.
- "**link_text**" - The default colour/gradient of any links in the text.
- "**link_hover**" - The colour/gradient of link text when we hover over it with the mouse.
- "**link_selected**" - The colour/gradient of link text when they are clicked on with the mouse.

Misc

UITextBox accepts the following miscellaneous parameters in a 'misc' block:

- "**shape**" - Can be one of 'rectangle' or 'rounded_rectangle'. Different shapes for this UI element.
- "**shape_corner_radius**" - Only used if our shape is 'rounded_rectangle'. It sets the radius, or radii, used for the rounded corners. Use a single integer to set all corners to the same radius, or four integers separated by commas to set each corner individually.
- "**border_width**" - The width of the border around the element in pixels. Defaults to "1".
- "**shadow_width**" - The width of the shadow around the element in pixels. Defaults to "2".
- "**padding**" - the horizontal and vertical 'padding' between the border and where we render the text. Defaults to "5,5".

- "**link_normal_underline**" - Set to either "1" or "0". Whether link text is normally underlined. Defaults to "0" (False).
- "**link_hover_underline**" - Set to either "1" or "0". Whether link text is underlined when they are hovered over with the mouse. Defaults to "1" (True).
- "**text_horiz_alignment**" - Set to "default, "left", "right" or "center". Controls the horizontal placement of all the text box text. Default is "default" which will use any in-text alignment set, all other values will override any in-text formatting.
- "**text_vert_alignment**" - Set to "default", "top", "bottom or "center". Controls the vertical placement of the text box text. Default is "default" which will use any in-text alignment set, all other values will override any in-text formatting.
- "**text_horiz_alignment_padding**" - If horizontal alignment is set to 'left' or 'right' this value will control the buffer between the edge of the box padding and where we start placing the text. Default is "0". Using "padding" is better but these parameters are included for completeness.
- "**text_vert_alignment_padding**" - If vertical alignment is set to 'top' or 'bottom' this value will control the buffer between the edge of the box padding and where we start placing the text. Default is "0". Using "padding" is better but these parameters are included for completeness.
- "**line_spacing**" - Sets the spacing of lines of text. The default is 1.25.
- "**tool_tip_delay**" - time in seconds before the button's tool tip (if it has one) will appear. Default is "1.0".

Sub-elements

The text box may also contain a *UIVerticalScrollBar* which you can reference with the block id 'text_box.vertical_scroll_bar'. You can also reference all of the buttons that are sub elements of the scroll bar with a theming block id of 'text_box.vertical_scroll_bar.button'.

You can further reference the individual buttons of the scroll bar by adding their object IDs:

- 'text_box.vertical_scroll_bar.#top_button'
- 'text_box.vertical_scroll_bar.#bottom_button'
- 'text_box.vertical_scroll_bar.#sliding_button'

There is more information on theming the vertical scroll bar at *UIVerticalScrollBar Theming Parameters*.

Example

Here is an example of a text box block in a JSON theme file, using the parameters described above.

Listing 22: text_box.json

```
{
    "text_box":
    {
        "colours":
        {
            "dark_bg":"#21282D",
            "normal_border": "#999999",
            "link_text": "#FF0000",
            "link_hover": "#FFFF00",
            "link_selected": "#FFFFFF"
```

(continues on next page)

(continued from previous page)

```
        },

        "misc":
        {
            "border_width": "1",
            "padding": "10,10",
            "link_normal_underline": "0",
            "link_hover_underline": "1",
            "line_spacing": "1.0"
        }
    },
    "text_box.vertical_scroll_bar":
    {
        "colours":
        {
           "dark_bg": "#505068"
        }
    },
    "text_box.vertical_scroll_bar.#sliding_button":
    {
        "misc":
        {
           "border_width": "1"
        }
    }
}
```

UITextEntryBox Theming Parameters

The *UITextEntryBox* theming block id is 'text_entry_box'.

Colours

UITextEntryBox makes use of these colour parameters in a 'colours' block. All of these colours can also be a colour gradient:

- "**dark_bg**" - The background colour/gradient of the text box element.
- "**selected_bg**" - The colour/gradient the background changes to when the text above it is selected.
- "**normal_border**" - The colour/gradient of the border around the text box element.
- "**selected_text**" - The colour/gradient of text when it has been selected.
- "**link_text**" - The default colour/gradient of any links in the text.
- "**link_hover**" - The colour/gradient of link text when we hover over it with the mouse.
- "**link_selected**" - The colour/gradient of link text when they are clicked on with the mouse.
- "**text_cursor**" - The colour of the text cursor.

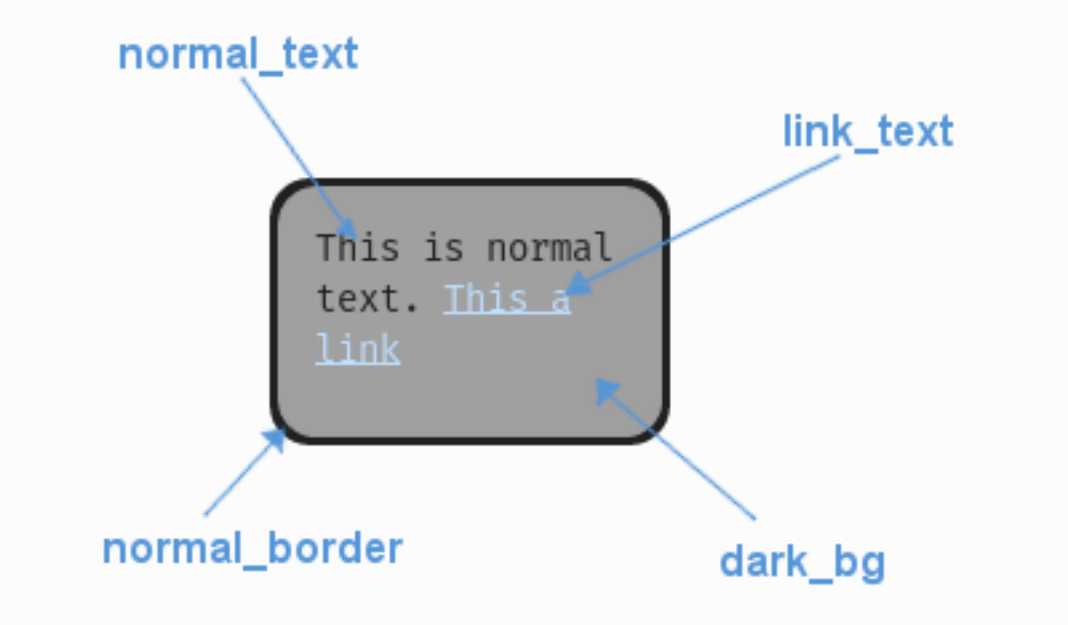


Fig. 12: A diagram of which part of the element is themed by which colour parameter. This correspondence is the same for the parameters named for the different link states e.g. the parameter 'link_hover' themes the same text as 'link_text' when the link is in the hovered state.

Misc

UITextEntryBox accepts the following miscellaneous parameters in a 'misc' block:

- "**shape**" - Can be one of 'rectangle' or 'rounded_rectangle'. Different shapes for this UI element.
- "**shape_corner_radius**" - Only used if our shape is 'rounded_rectangle'. It sets the radius, or radii, used for the rounded corners. Use a single integer to set all corners to the same radius, or four integers separated by commas to set each corner individually.
- "**border_width**" - The width of the border around the element in pixels. Defaults to "1".
- "**shadow_width**" - The width of the shadow around the element in pixels. Defaults to "2".
- "**padding**" - the horizontal and vertical 'padding' between the border and where we render the text. Defaults to "5,5".
- "**link_normal_underline**" - Set to either "1" or "0". Whether link text is normally underlined. Defaults to "0" (False).
- "**link_hover_underline**" - Set to either "1" or "0". Whether link text is underlined when they are hovered over with the mouse. Defaults to "1" (True).
- "**text_horiz_alignment**" - Set to "default, "left", "right" or "center". Controls the horizontal placement of all the text box text. Default is "default" which will use any in-text alignment set, all other values will override any in-text formatting.
- "**text_vert_alignment**" - Set to "default", "top", "bottom or "center". Controls the vertical placement of the text box text. Default is "default" which will use any in-text alignment set, all other values will override any in-text formatting.
- "**text_horiz_alignment_padding**" - If horizontal alignment is set to 'left' or 'right' this value will control the buffer between the edge of the box padding and where we start placing the text. Default is "0". Using "padding"

is better but these parameters are included for completeness.

- "**text_vert_alignment_padding**" - If vertical alignment is set to 'top' or 'bottom' this value will control the buffer between the edge of the box padding and where we start placing the text. Default is "0". Using "padding" is better but these parameters are included for completeness.
- "**line_spacing**" - Sets the spacing of lines of text. The default is 1.25.
- "**tool_tip_delay**" - time in seconds before the button's tool tip (if it has one) will appear. Default is "1.0".

Sub-elements

The text entry box may also contain a *UIVerticalScrollBar* which you can reference with the block id 'text_entry_box.vertical_scroll_bar'. You can also reference all of the buttons that are sub elements of the scroll bar with a theming block id of 'text_entry_box.vertical_scroll_bar.button'.

You can further reference the individual buttons of the scroll bar by adding their object IDs:

- 'text_entry_box.vertical_scroll_bar.#top_button'
- 'text_entry_box.vertical_scroll_bar.#bottom_button'
- 'text_entry_box.vertical_scroll_bar.#sliding_button'

There is more information on theming the vertical scroll bar at *UIVerticalScrollBar Theming Parameters*.

Example

Here is an example of a text entry box block in a JSON theme file, using the parameters described above.

Listing 23: text_entry_box.json

```
{
    "text_entry_box":
    {
        "colours":
        {
            "dark_bg":"#21282D",
            "normal_border": "#999999",
            "link_text": "#FF0000",
            "link_hover": "#FFFF00",
            "link_selected": "#FFFFFF"
        },

        "misc":
        {
            "border_width": "1",
            "padding": "10,10",
            "link_normal_underline": "0",
            "link_hover_underline": "1",
            "line_spacing": "1.0"
        }
    },
    "text_entry_box.vertical_scroll_bar":
    {
        "colours":
```

(continues on next page)

(continued from previous page)

```
        {
            "dark_bg": "#505068"
        }
    },
    "text_entry_box.vertical_scroll_bar.#sliding_button":
    {
        "misc":
        {
            "border_width": "1"
        }
    }
}
```

UITextEntryLine Theming Parameters

The *UITextEntryLine* theming block id is 'text_entry_line'.

Colours

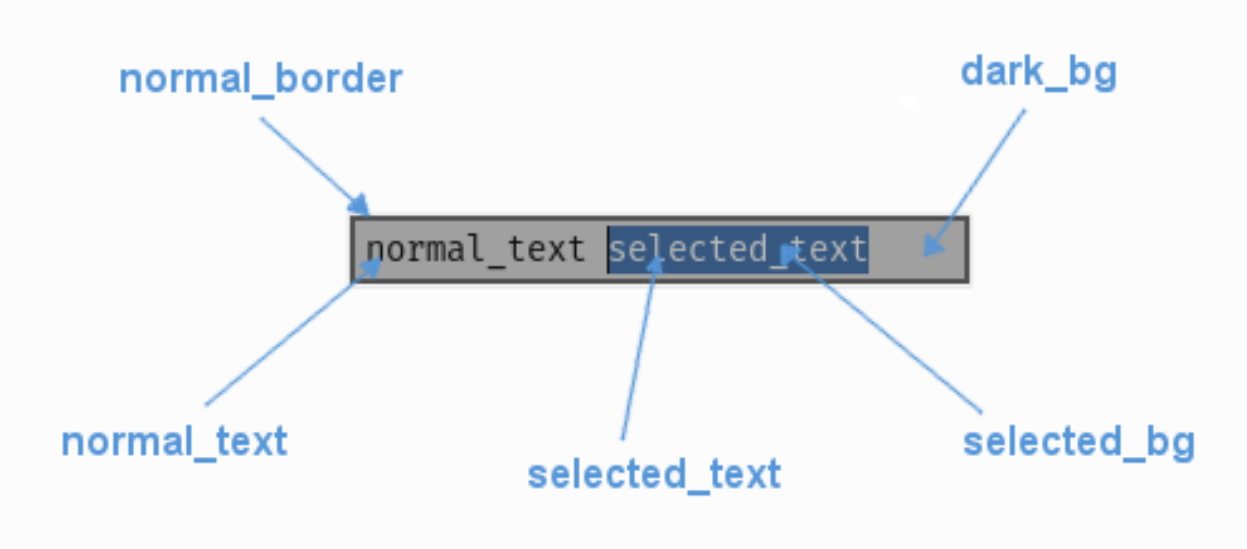


Fig. 13: A diagram of which part of the element is themed by which colour parameter.

UITextEntryLine makes use of these colour parameters in a 'colours' block. All of these colours can also be a colour gradient except the text cursor:

- "**dark_bg**" - The default colour/gradient of the background to the entry line element.
- "**selected_bg**" - The colour/gradient the background changes to when the text above it is selected.
- "**normal_text**" - The default colour/gradient of text entered into the element.
- "**selected_text**" - The colour/gradient of text when it has been selected.
- "**normal_border**" - The colour/gradient of the border around the text entry element.
- "**text_cursor**" - The colour of the text cursor.

Font

UITextEntryLine accepts a font specified in the theme via a 'font' block. A 'font' block has these parameters:

- "**name**" - Necessary to make a valid block. This is the name that this font goes by in the UI, if this is a new font then subsequent font instances with different styles or sizes should use the same name.
- "**locale**" - Optional parameter to set this font as belonging to a particular locale only. See the *Localization* guide. You will need to keep repeating the locale specifier if using prototypes to make a hierarchy.
- "**size**" - Necessary to make a valid block. This is the point size of the font to use on the text entry line.
- "**bold**" - Optional parameter. Set it to "1" to make this font bold.
- "**italic**" - Optional parameter. Set it to "1" to make this font italic.

There are two methods to refer to font resource locations. First, using packaged resources:

- **"regular_resource - The location of this font's file with no particular style applied.**
  - **package** - The name of the python package containing this resource - e.g. 'data.fonts'
  - **resource** - The file name of the resource - e.g. 'FiraCode-Regular.ttf'
- **"bold_resource" - The location of this font's file with bold style applied.**
  - **package** - The name of the python package containing this resource - e.g. 'data.fonts'
  - **resource** - The file name of the resource - e.g. 'FiraCode-Bold.ttf'
- **"italic_resource" - The location of this font's file with italic style applied.**
  - **package** - The name of the python package containing this resource - e.g. 'data.fonts'
  - **resource** - The file name of the resource - e.g. 'FiraMono-Italic.ttf'
- **"bold_italic_resource" - The location of this font's file with bold and italic style applied.**
  - **package** - The name of the python package containing this resource - e.g. 'data.fonts'
  - **resource** - The file name of the resource - e.g. 'FiraMono-BoldItalic.ttf'

Second using paths:

- "**regular_path**" - The path to this font's file with no particular style applied.
- "**bold_path**" - The path to this font's file with bold style applied.
- "**italic_path**" - The path to this font's file with italic style applied.
- "**bold_italic_path**" - The path to this font's file with bold and italic style applied.

You only need to specify locations if this is the first use of this font name in the GUI.

Misc

UITextEntryLine accepts the following miscellaneous parameters in a 'misc' block:

- "**shape**" - Can be one of 'rectangle' or 'rounded_rectangle'. Different shapes for this UI element.
- "**shape_corner_radius**" - Only used if our shape is 'rounded_rectangle'. It sets the radius, or radii, used for the rounded corners. Use a single integer to set all corners to the same radius, or four integers separated by commas to set each corner individually.
- "**border_width**" - the width of the border around the element in pixels. Defaults to "1".
- "**shadow_width**" - the width of the shadow around the element in pixels. Defaults to "1".

- "**padding**" - the horizontal and vertical 'padding' between the border and where we render the text. Defaults to "4,2".
- "**tool_tip_delay**" - time in seconds before the button's tool tip (if it has one) will appear. Default is "1.0".

Example

Here is an example of a text entry line block in a JSON theme file using all the parameters described above.

Listing 24: text_entry_line.json

```
{
    "text_entry_line":
    {
        "colours":
        {
            "dark_bg": "#25292e",
            "selected_bg": "#55595e",
            "normal_text": "#AAAAAA",
            "selected_text": "#FFFFFF",
            "normal_border": "#FFFFFF"
        },
        "font":
        {
            "name": "montserrat",
            "size": "12",
            "bold": "0",
            "italic": "1",
            "regular_resource": {
                "package": "data.fonts",
                "resource": "Montserrat-Regular.ttf"
            },
            "bold_resource": {
                "package": "data.fonts",
                "resource": "Montserrat-Bold.ttf"
            },
            "italic_resource": {
                "package": "data.fonts",
                "resource": "Montserrat-Italic.ttf"
            },
            "bold_italic_resource": {
                "package": "data.fonts",
                "resource": "Montserrat-BoldItalic.ttf"
            },
        },
        "misc":
        {
            "shape": "rounded_rectangle",
            "shape_corner_radius": 5,
            "border_width": "2",
            "shadow_width": "2",
            "padding": "6,4"
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

UITooltip Theming Parameters

The `UIToolTip` theming block id is 'tool_tip'.

Misc

UITooltip accepts the following miscellaneous parameters in a 'misc' block:

- "**rect_width**" - The width of the rectangle around the tool tip in pixels, including any shadows or borders. The height is determined dynamically.

Sub-elements

The `UIToolTip` contains a *UITextBox* so you can use the block ID 'tool_tip.text_box' to start styling it.

There is more information on theming the text box at *UITextBox Theming Parameters*.

Example

Here is an example of a tool tip block in a JSON theme file, using parameters from the text box element.

Listing 25: tool_tip.json

```
{
    "tool_tip":
    {
        "misc":
        {
           "rect_width": "170"
        }
    },
    "tool_tip.text_box":
    {
        "colours":
        {
            "dark_bg":"#505050",
            "normal_border": "#FFFFFF"
        },

        "misc":
        {
            "border_width": "2",
            "padding": "5,5"
        }
    }
}
```

UIVerticalScrollBar Theming Parameters

The *UIVerticalScrollBar* theming block id is 'vertical_scroll_bar'.

Colours

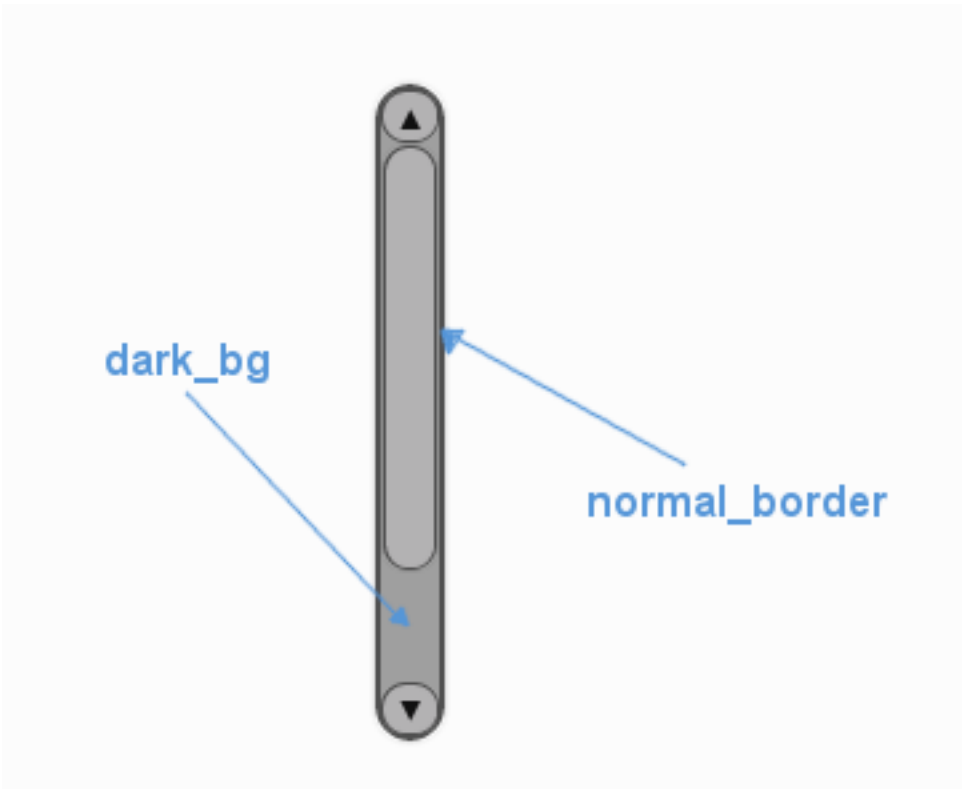


Fig. 14: A diagram of which part of the element is themed by which colour parameter. The scroll bar's buttons are themed in a separate block.

UIVerticalScrollBar makes use of these colour parameters in a 'colours' block. All of these colours can also be a colour gradient:

- "**dark_bg**" - The background colour/gradient of the 'back' of the scroll bar, the colour of the track that the scroll bar moves along.
- "**normal_border**" - The colour/gradient of the border around the scroll bar.
- "**disabled_dark_bg**" - The colour/gradient of the track when disabled.
- "**disabled_border**" - The border colour/gradient of the slider when disabled.

Misc

UIVerticalScrollBar accepts the following miscellaneous parameters in a 'misc' block:

- "**shape**" - Can be one of 'rectangle' or 'rounded_rectangle'. Different shapes for this UI element.
- "**shape_corner_radius**" - Only used if our shape is 'rounded_rectangle'. It sets the radius, or radii, used for the rounded corners. Use a single integer to set all corners to the same radius, or four integers separated by commas to set each corner individually.
- "**border_width**" - the width in pixels of the border around the bar. Defaults to 1.
- "**shadow_width**" - the width in pixels of the shadow behind the bar. Defaults to 1.
- "**enable_arrow_buttons**" - Enables or disables the arrow buttons for the scroll bar. "1" is enabled, "0" is disabled. Defaults to "1".
- "**tool_tip_delay**" - time in seconds before the scroll bar's tool tip (if it has one) will appear. Default is "1.0".

Sub-elements

You can reference all three of the buttons that are sub elements of the scroll bar with a theming block id of 'vertical_scroll_bar.button'. You can reference both of the arrow buttons with the class_id: '@arrow_button'. You can also reference the three buttons individually by adding their object IDs:

- 'vertical_scroll_bar.#top_button'
- 'vertical_scroll_bar.#bottom_button'
- 'vertical_scroll_bar.#sliding_button'

There is more information on theming buttons at *UIButton Theming Parameters*.

Example

Here is an example of some vertical scroll bar blocks in a JSON theme file, using the parameters described above (and some from UIButton).

Listing 26: vertical_scroll_bar.json

```
{
    "vertical_scroll_bar":
    {
        "colours":
        {
            "normal_bg": "#25292e",
            "hovered_bg": "#35393e",
            "disabled_bg": "#25292e",
            "selected_bg": "#25292e",
            "active_bg": "#193784",
            "dark_bg": "#15191e",
            "normal_text": "#c5cbd8",
            "hovered_text": "#FFFFFF",
            "selected_text": "#FFFFFF",
            "disabled_text": "#6d736f"
        },
```

(continues on next page)

(continued from previous page)

```
        "misc":
        {
            "shape": "rectangle",
            "border_width": "0",
            "enable_arrow_buttons": "0"
        }
    },
    "vertical_scroll_bar.button":
    {
        "misc":
        {
            "border_width": "1"
        }
    },
    "vertical_scroll_bar.#sliding_button":
    {
        "colours":
        {
            "normal_bg": "#FF0000"
        }
    }
}
```

UIWindow Theming Parameters

The *UIWindow* theming block id is 'window'.

Colours

UIWindow makes use of these colour parameters in a 'colours' block. All of these colours can also be a colour gradient:

- "**dark_bg**" - The background colour/gradient of the 'back' of the scroll bar, the colour of the track that the scroll bar moves along.
- "**normal_border**" - The colour/gradient of the border around the scroll bar.

Misc

UIWindow accepts the following miscellaneous parameters in a 'misc' block:

- "**shape**" - Can be one of 'rectangle' or 'rounded_rectangle'. Different shapes for this UI element.
- "**shape_corner_radius**" - Only used if our shape is 'rounded_rectangle'. It sets the radius, or radii, used for the rounded corners. Use a single integer to set all corners to the same radius, or four integers separated by commas to set each corner individually.
- "**border_width**" - The width of the border around the element in pixels. Defaults to "1".
- "**shadow_width**" - The width of the shadow around the element in pixels. Defaults to "2".
- "**enable_title_bar**" - Controls whether the title bar appears with it's attendant buttons. "1" is enabled and "0" is disabled. Defaults to "1".

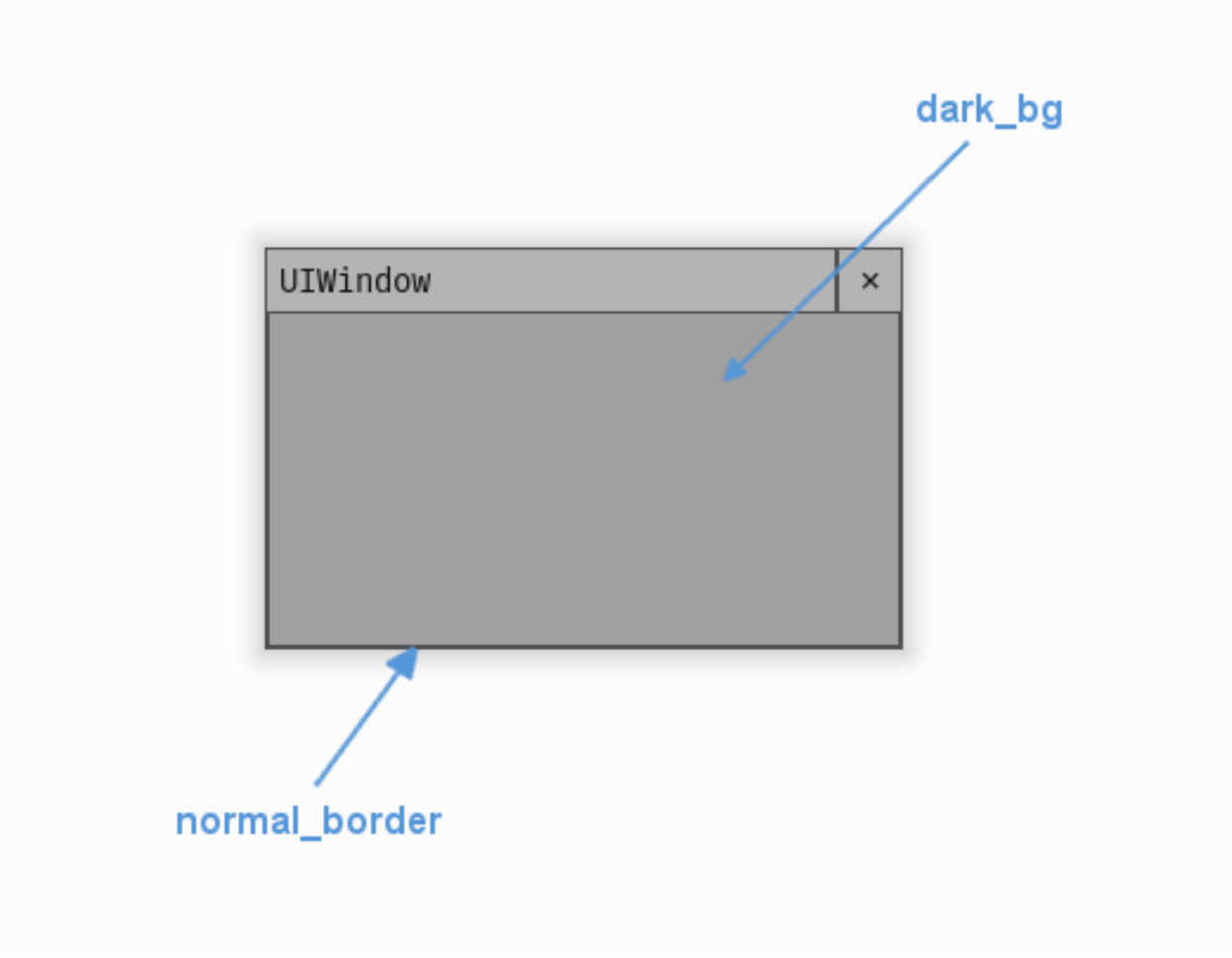


Fig. 15: A diagram of which part of the element is themed by which colour parameter.

- "**enable_close_button**" - Controls whether the close button appears on the title bar, only has any affect if the title bar exists. "1" is enabled and "0" is disabled. Defaults to "1".
- "**title_bar_height**" - The height of the title bar in pixels if it exists. Defaults to 28.

Sub-elements

You can reference all of the buttons that are sub elements of the window with a theming block id of 'window.button'. You can also reference the buttons individually by adding their object IDs:

- 'window.#title_bar'
- 'window.#close_button'

There is more information on theming buttons at *UIButton Theming Parameters*.

Example

Here is an example of a window block in a JSON theme file, using the parameters described above.

Listing 27: window.json

```
{
   "window":
   {
      "colours":
      {
         "dark_bg":"#21282D",
         "normal_border": "#999999"
      },

      "misc":
      {
         "shape": "rounded_rectangle",
         "shape_corner_radius": "10",
         "border_width": "1",
         "shadow_width": "15",
         "title_bar_height": "20"
      }
   },
   "window.#title_bar":
   {
      "misc":
      {
         "text_horiz_alignment": "center"
      }
   }
}
```

UIWorldSpaceHealthBar Theming Parameters

The *UIWorldSpaceHealthBar* theming block id is 'world_space_health_bar'.

Colours

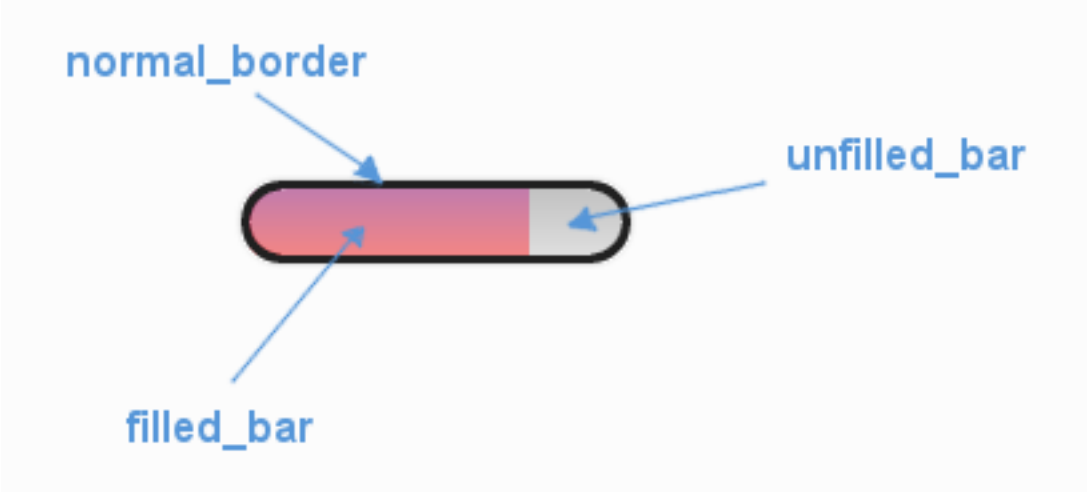


Fig. 16: A diagram of which part of the element is themed by which colour parameter.

UIWorldSpaceHealthBar makes use of these colour parameters in a 'colours' block:

- "**normal_border**" - The colour/gradient of the health bar's border if it has one.
- "**filled_bar**" - The colour/gradient of the actual bar itself, of the portion of it that is still full.
- "**unfilled_bar**" - The colour/gradient of an empty portion of the health bar.

Misc

UIWorldSpaceHealthBar has the following miscellaneous parameters in a 'misc' block:

- "**shape**" - Can be one of 'rectangle' or 'rounded_rectangle'. Different shapes for this UI element.
- "**shape_corner_radius**" - Only used if our shape is 'rounded_rectangle'. It sets the radius, or radii, used for the rounded corners. Use a single integer to set all corners to the same radius, or four integers separated by commas to set each corner individually.
- "**follow_sprite_offset**" - The x,y offset values the bar relative to the sprite. Defaults to "0,0".
- "**border_width**" - The width of the border around the health bar. Defaults to "1". Can be "0" to remove the border.
- "**shadow_width**" - The width of the border around the health bar. Defaults to "1". Can be "0" to remove the border.
- "**tool_tip_delay**" - time in seconds before the button's tool tip (if it has one) will appear. Default is "1.0".

Example

Here is an example of a world space health bar block in a JSON theme file using the parameters described above.

Listing 28: world_space_health_bar.json

```
{
    "world_space_health_bar":
    {
        "colours":
        {
            "normal_border": "#AAAAAA",
            "filled_bar": "#f4251b",
            "unfilled_bar": "#CCCCCC"
        },
        "misc":
        {
            "follow_sprite_offset": "-5, 32",
            "border_width": "0"
        }
    }
}
```

7.3.6 Theme Options Per Window

UIColourPickerDialog Theming Parameters

UIColourPickerDialog is a UIWindow with the element id of 'colour_picker_dialog' and the default object id of '#colour_picker_dialog'.

Inherited Parameters

As a UIWindow the Colour Picker Dialog has all the theming parameters of the UIWindow, which you can read more about here *UIWindow Theming Parameters*.

Sub-elements

As well as the sub-elements of the UIWindow (title bar and close button) which you can read about here *UIWindow Theming Parameters*, the Colour Picker Dialog has the following sub element IDs -

UIButtons:

- '#colour_picker_dialog.#ok_button'
- '#colour_picker_dialog.#cancel_button'

UIColourChannelEditor:

- '#colour_picker_dialog.colour_channel_editor'

You can find out more about theming buttons here: *UIButton Theming Parameters*.

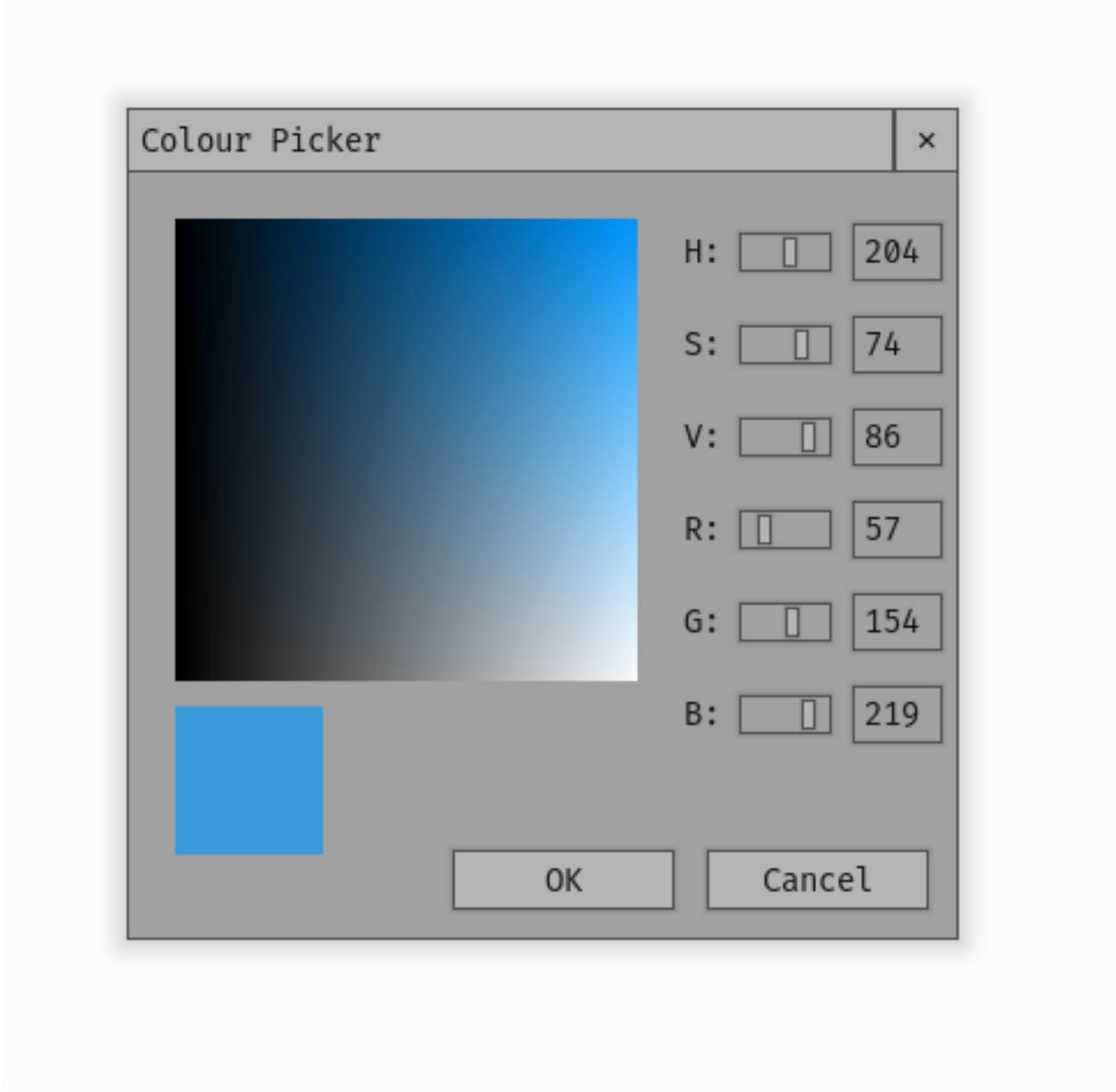


Fig. 17: An image of the Colour Picker Dialog.

UIColourChannelEditor Theming Parameters

UIColourChannelEditor has no theming parameters of its own.

Sub-elements

The Colour channel editor is only used in the colour picker dialog, and is composed of three sub-elements. You can access them for theming with the following IDs:

- '#colour_picker_dialog.colour_channel_editor.text_entry_line'
- '#colour_picker_dialog.colour_channel_editor.label'
- '#colour_picker_dialog.colour_channel_editor.horizontal_slider'

You can find out more about theming text entry lines here: *UITextEntryLine Theming Parameters*, labels here: *UILabel Theming Parameters* and horizontal sliders here: *UIHorizontalSlider Theming Parameters*.

UIConfirmationDialog Theming Parameters

UIConfirmationDialog is a UIWindow with the element id of 'confirmation_dialog' and a default object id of '#confirmation_dialog'.

Fig. 18: An image of the Confirmation Dialog.

Inherited Parameters

As a UIWindow the Confirmation Dialog has all the theming parameters of the UIWindow, which you can read more about here *UIWindow Theming Parameters*.

Sub-elements

As well as the sub-elements of the UIWindow (title bar and close button) which you can read about here *UIWindow Theming Parameters*, the Confirmation Dialog has the following sub element IDs -

UIButtons:

- 'confirmation_dialog.#confirm_button'
- 'confirmation_dialog.#cancel_button'

UITextBox:

- 'confirmation_dialog.text_box'

You can find out more about theming buttons here: *UIButton Theming Parameters* and text boxes here: *UITextBox Theming Parameters*.

UIFileDialog Theming Parameters

`UIFileDialog` is a UIWindow with the element id of 'file_dialog' and a default object id of '#file_dialog'.

Inherited Parameters

As a UIWindow the File Dialog has all the theming parameters of the UIWindow, which you can read more about here *UIWindow Theming Parameters*.

Sub-elements

As well as the sub-elements of the UIWindow (title bar and close button) which you can read about here *UIWindow Theming Parameters*, the file dialog has the following sub element IDs -

UIButtons:

- 'file_dialog.#ok_button'
- 'file_dialog.#cancel_button'
- 'file_dialog.#home_icon_button'
- 'file_dialog.#delete_icon_button'
- 'file_dialog.#parent_icon_button'
- 'file_dialog.#refresh_icon_button'

UITextEntryLine:

- 'file_dialog.#file_path_text_line'

UISelectionList:

- 'file_dialog.#file_display_list'

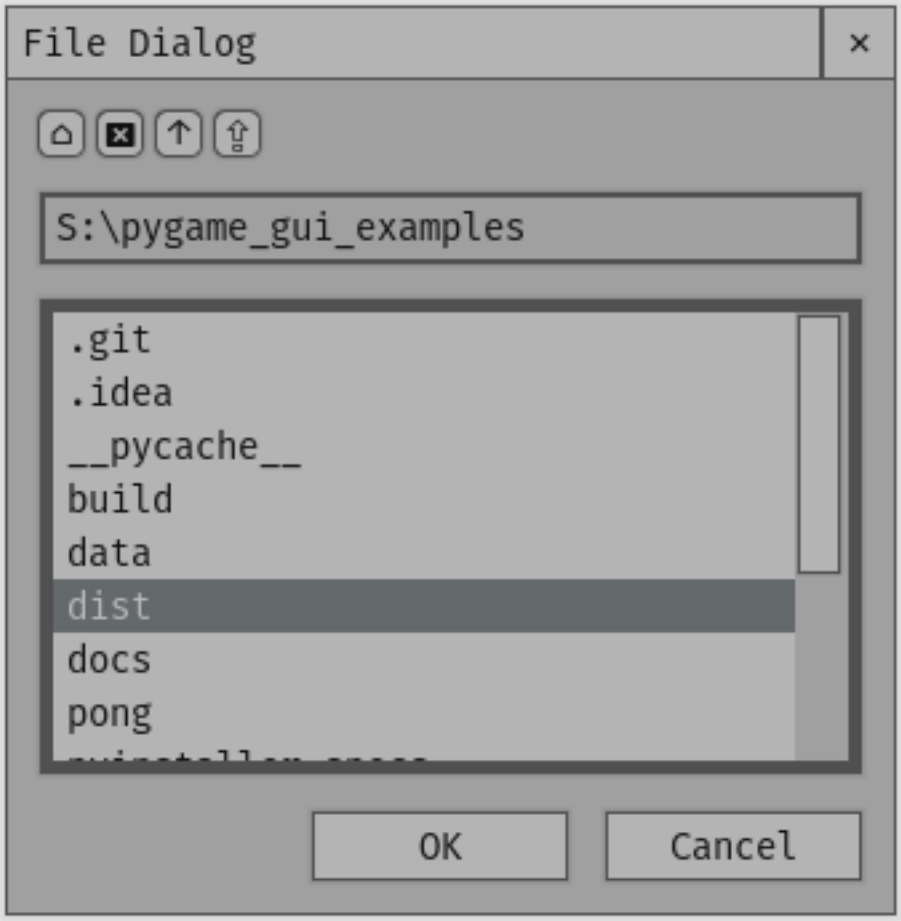


Fig. 19: An image of the File Dialog window.

You can find out more about theming buttons here: *UIButton Theming Parameters*, text entry lines here: *UITextEntryLine Theming Parameters* and selection lists here: *UISelectionList Theming Parameters*.

UIMessageWindow Theming Parameters

`UIMessageWindow` is a UIWindow with the element id of 'message_window' and a default object id of '#message_window'.

Fig. 20: An image of the Message Window.

Inherited Parameters

As a UIWindow the Message Window has all the theming parameters of the UIWindow, which you can read more about here *UIWindow Theming Parameters*.

Sub-elements

As well as the sub-elements of the UIWindow (title bar and close button) which you can read about here *UIWindow Theming Parameters*, the message window has the following sub element IDs -

UIButtons:

- 'message_window.#dismiss_button'

UITextBox:

- 'message_window.text_box'

You can find out more about theming buttons here: *UIButton Theming Parameters* and text boxes here: *UITextBox Theming Parameters*.

7.4 GUI events

Some of the UI Elements produce a `pygame.Event` when they are interacted with. These events all follow a common structure that looks something like this:

- **'type'** : An id for the specific event that has happened. e.g. 'pygame_gui.UI_BUTTON_PRESSED'
- **'ui_element'** : The UI element that fired this event.
- **'ui_object_id'** : The most unique ID that applies to the UI element that fired this event. e.g. 'hud_window.#sell_button'

Though some of the events also have additional data relevant to that event.

7.4.1 Event list

A list of all the different events by element.

`UIButton` - UI_BUTTON_PRESSED

Fired when a user presses a button by clicking on it with a mouse, and releasing the mouse button while still hovering over it.

- **'type'** : pygame_gui.UI_BUTTON_PRESSED
- **'ui_element'** : The *`UIButton`* that fired this event.
- **'ui_object_id'** : The most unique ID for the button that fired this event.
- **'mouse_button'**
  [The mouse button that did the pressing (pygame.BUTTON_LEFT etc).] You have to enable button events for buttons other than the left mouse button which is enabled by default.

Example usage:

```
for event in pygame.event.get():
    if event.type == pygame_gui.UI_BUTTON_PRESSED:
        if event.ui_element == test_button:
            print('Test button pressed')
```

`UIButton` - UI_BUTTON_START_PRESS

Fired when a user first presses down a button by clicking on it with a mouse. This is fired before you release the mouse button.

- **'type'** : pygame_gui.UI_BUTTON_START_PRESS
- **'ui_element'** : The *`UIButton`* that fired this event.
- **'ui_object_id'** : The most unique ID for the button that fired this event.

- **'mouse_button'**
  [The mouse button that did the pressing (pygame.BUTTON_LEFT etc).] You have to enable button events for buttons other than the left mouse button which is enabled by default.

Example usage:

```
for event in pygame.event.get():
    if event.type == pygame_gui.UI_BUTTON_PRESSED:
        if event.ui_element == test_button:
            print('Test button pressed')
```

UIButton - UI_BUTTON_DOUBLE_CLICKED

Fired when a user double clicks on a button by left clicking on it with a mouse, then left clicking on it again quickly.

- **'type'** : pygame_gui.UI_BUTTON_DOUBLE_CLICKED
- **'ui_element'** : The *UIButton* that fired this event.
- **'ui_object_id'** : The most unique ID for the button that fired this event.
- **'mouse_button'**
  [The mouse button that did the pressing (pygame.BUTTON_LEFT etc).] You have to enable button events for buttons other than the left mouse button which is enabled by default.

Example usage:

```
for event in pygame.event.get():
    if event.type == pygame_gui.UI_BUTTON_DOUBLE_CLICKED:
        if event.ui_element == test_button:
            print('Test button pressed')
```

UIButton - UI_BUTTON_ON_HOVERED

Fired when a user starts hovering over a button with the mouse.

- **'type'** : pygame_gui.UI_BUTTON_ON_HOVERED
- **'ui_element'** : The *UIButton* that fired this event.
- **'ui_object_id'** : The most unique ID for the button that fired this event.

Example usage:

```
for event in pygame.event.get():
    if event.type == pygame_gui.UI_BUTTON_ON_HOVERED:
        if event.ui_element == test_button:
            print('Test button hovered')
```

UIButton - UI_BUTTON_ON_UNHOVERED

Fired when a user stops hovering over a button with the mouse.

- **'type'** : pygame_gui.UI_BUTTON_ON_UNHOVERED
- **'ui_element'** : The *UIButton* that fired this event.
- **'ui_object_id'** : The most unique ID for the button that fired this event.

Example usage:

```
for event in pygame.event.get():
    if event.type == pygame_gui.UI_BUTTON_ON_UNHOVERED:
        if event.ui_element == test_button:
            print('Test button unhovered')
```

UITextBox - UI_TEXT_BOX_LINK_CLICKED

Fired when a user clicks on a HTML link in a text box.

- **'type'** : pygame_gui.UI_TEXT_BOX_LINK_CLICKED,
- **'link_target'** : The 'href' parameter of the clicked link.
- **'ui_element'** : The *UITextBox* that fired this event.
- **'ui_object_id'** : The most unique ID for the text box that fired this event.

Example usage:

```
for event in pygame.event.get():
    if event.type == pygame_gui.UI_TEXT_BOX_LINK_CLICKED:
        print(event.link_target)
```

UITextEntryLine - UI_TEXT_ENTRY_CHANGED

Fired when a user changes the text in a text entry element by entering of deleting text. Not fired when set_text() is used.

- **'type'** : pygame_gui.UI_TEXT_ENTRY_CHANGED,
- **'text'** : The user entered text in the text entry line.
- **'ui_element'** : The *UITextEntryLine* that fired this event.
- **'ui_object_id'** : The most unique ID for the text entry line that fired this event.

Example usage:

```
for event in pygame.event.get():
    if event.type == pygame_gui.UI_TEXT_ENTRY_CHANGED:
        print("Changed text:", event.text)
```

UITextEntryLine - UI_TEXT_ENTRY_FINISHED

Fired when a user presses the enter key with a text entry element active for entry.

- **'type'** : pygame_gui.UI_TEXT_ENTRY_FINISHED,
- **'text'** : The user entered text in the text entry line.
- **'ui_element'** : The *UITextEntryLine* that fired this event.
- **'ui_object_id'** : The most unique ID for the text entry line that fired this event.

Example usage:

```
for event in pygame.event.get():
    if event.type == pygame_gui.UI_TEXT_ENTRY_FINISHED:
        print("Entered text:", event.text)
```

UIDropDownMenu - UI_DROP_DOWN_MENU_CHANGED

Fired when a user selects an option in a drop down menu.

- **'type'** : pygame_gui.UI_DROP_DOWN_MENU_CHANGED,
- **'text'** : The text of the selected option.
- **'selected_option_id'** : The ID of the selected option, if no ids were specified this will be the same as 'text'.
- **'ui_element'** : The *UIDropDownMenu* that fired this event.
- **'ui_object_id'** : The most unique ID for the drop down menu that fired this event.

Example usage:

```
for event in pygame.event.get():
    if event.type == pygame_gui.UI_DROP_DOWN_MENU_CHANGED:
        print("Selected option:", event.text)
```

UIHorizontalSlider - UI_HORIZONTAL_SLIDER_MOVED

Fired when a user moves a horizontal slider by pressing an arrow button or dragging the sliding button.

- **'type'** : pygame_gui.UI_HORIZONTAL_SLIDER_MOVED
- **'value'** : The current value the slider is set to.
- **'ui_element'** : The *UIHorizontalSlider* that fired this event.
- **'ui_object_id'** : The most unique ID for the button that fired this event.

Example usage:

```
for event in pygame.event.get():
    if event.type == pygame_gui.UI_HORIZONTAL_SLIDER_MOVED:
        if event.ui_element == test_slider:
            print('current slider value:', event.value)
```

UISelectionList - UI_SELECTION_LIST_NEW_SELECTION

Fired when a user selects a new item in a selection list.

- **'type'** : pygame_gui.UI_SELECTION_LIST_NEW_SELECTION,
- **'text'** : The text of the selected item.
- **'ui_element'** : The *UISelectionList* that fired this event.
- **'ui_object_id'** : The most unique ID for the element that fired this event.

Example usage:

```
for event in pygame.event.get():
    if event.type == pygame_gui.UI_SELECTION_LIST_NEW_SELECTION:
        if event.ui_element == test_selection_list:
            print("Selected item:", event.text)
```

UISelectionList - UI_SELECTION_LIST_DROPPED_SELECTION

Fired when a user un-selects an item, dropping it from a selection list.

- **'type'** : pygame_gui.UI_SELECTION_LIST_DROPPED_SELECTION,
- **'text'** : The text of the dropped item.
- **'ui_element'** : The *UISelectionList* that fired this event.
- **'ui_object_id'** : The most unique ID for the element that fired this event.

Example usage:

```
for event in pygame.event.get():
     if event.type == pygame_gui.UI_SELECTION_LIST_DROPPED_SELECTION:
         if event.ui_element == test_selection_list:
             print("Dropped item:", event.text)
```

UISelectionList - UI_SELECTION_LIST_DOUBLE_CLICKED_SELECTION

Fired when a user double clicks on an item in a selection list.

- **'type'** : pygame_gui.UI_SELECTION_LIST_DOUBLE_CLICKED_SELECTION,
- **'text'** : The text of the double clicked item.
- **'ui_element'** : The *UISelectionList* that fired this event.
- **'ui_object_id'** : The most unique ID for the element that fired this event.

Example usage:

```
for event in pygame.event.get():
    if event.type == pygame_gui.UI_SELECTION_LIST_DOUBLE_CLICKED_SELECTION:
        if event.ui_element == test_selection_list:
            print("Double clicked item:", event.text)
```

UIWindow - UI_WINDOW_CLOSE

Fired when a window is closed.

- **'type'** : pygame_gui.UI_WINDOW_CLOSE,
- **'ui_element'** : The *UIWindow* that fired this event.
- **'ui_object_id'** : The most unique ID for the element that fired this event.

Example usage:

```
for event in pygame.event.get():
    if event.type == pygame_gui.UI_WINDOW_CLOSE:
        if event.ui_element == window:
            print("Window closed")
```

UIWindow - UI_WINDOW_MOVED_TO_FRONT

Fired when a UI window is moved to the top of the stack. This happens when they are newly created and when they are clicked on by a user.

- **'type'** : pygame_gui.UI_WINDOW_MOVED_TO_FRONT,
- **'ui_element'** : The *UIWindow* that fired this event.
- **'ui_object_id'** : The most unique ID for the element that fired this event.

Example usage:

```
for event in pygame.event.get():
    if event.type == pygame_gui.UI_WINDOW_MOVED_TO_FRONT:
        if event.ui_element == window:
            print("Window moved to front")
```

UIWindow - UI_WINDOW_RESIZED

Fired when a window is resized.

- **'type'** : pygame_gui.UI_WINDOW_RESIZED,
- **'ui_element'** : The *UIWindow* that fired this event.
- **'ui_object_id'** : The most unique ID for the element that fired this event.
- **'external_size'** : The total size of the window including title bar, borders & shadows.
- **'internal_size'** : The size inside the window where other elements are place (excluding title bar, borders & shadows).

Example usage:

```
for event in pygame.event.get():
    if event.type == pygame_gui.UI_WINDOW_RESIZED:
        if event.ui_element == window:
            print("Window resized")
```

UIConfirmationDialog - UI_CONFIRMATION_DIALOG_CONFIRMED

Fired when the 'confirm' button is chosen in a confirmation dialog.

- **'type'** : pygame_gui.UI_CONFIRMATION_DIALOG_CONFIRMED,
- **'ui_element'** : The *UIConfirmationDialog* that fired this event.
- **'ui_object_id'** : The most unique ID for the element that fired this event.

Example usage:

```
for event in pygame.event.get():
    if event.type == pygame_gui.UI_CONFIRMATION_DIALOG_CONFIRMED:
        if event.ui_element == confirmation_dialog:
            print("Confirming action.")
```

UIFileDialog - UI_FILE_DIALOG_PATH_PICKED

Fired when a path has been chosen in a file dialog.

- **'type'** : pygame_gui.UI_FILE_DIALOG_PATH_PICKED
- **'text'** : The path picked.
- **'ui_element'** : The *UIFileDialog* that fired this event.
- **'ui_object_id'** : The most unique ID for the element that fired this event.

Example usage:

```
for event in pygame.event.get():
    if event.type == pygame_gui.UI_FILE_DIALOG_PATH_PICKED:
        if event.ui_element == file_dialog:
            print("Path picked:", event.text)
```

UIColourPickerDialog - UI_COLOUR_PICKER_COLOUR_PICKED

Fired when a colour has been chosen in a colour picker dialog.

- **'type'** : pygame_gui.UI_COLOUR_PICKER_COLOUR_PICKED
- **'colour'** : The colour picked.
- **'ui_element'** : The *UIColourPickerDialog* that fired this event.
- **'ui_object_id'** : The most unique ID for the element that fired this event.

Example usage:

```
for event in pygame.event.get():
    if event.type == pygame_gui.UI_COLOUR_PICKER_COLOUR_PICKED:
        if event.ui_element == colour_picker:
            print("Colour picked:", event.colour)
```

UIColourChannelEditor - UI_COLOUR_PICKER_COLOUR_CHANNEL_CHANGED

Fired when a colour channel element has had it's value changed. This event is used by the colour picker dialog.

- **'type'** : pygame_gui.UI_COLOUR_PICKER_COLOUR_CHANNEL_CHANGED
- **'value'** : The current value of the channel.
- **'channel_index'** : The index of this colour channel in the colour (R=0, G=1, B=2 etc).
- **'ui_element'** : The UIColourChannelEditor that fired this event.
- **'ui_object_id'** : The most unique ID for the element that fired this event.

Example usage:

```
for event in pygame.event.get():
    if event.type == pygame_gui.UI_COLOUR_PICKER_COLOUR_CHANNEL_CHANGED:
        if event.ui_element == colour_channel:
            print("Colour channel value:", event.value)
```

UIConsoleWindow - UI_CONSOLE_COMMAND_ENTERED

Fired when a command is entered into a console window. Usually by typing it in and pressing enter.

- **'type'** : pygame_gui.UI_CONSOLE_COMMAND_ENTERED
- **'command'** : The entered command.
- **'ui_element'** : The *UIConsoleWindow* that fired this event.
- **'ui_object_id'** : The most unique ID for the element that fired this event.

Example usage:

```
for event in pygame.event.get():
    if event.type == pygame_gui.UI_CONSOLE_COMMAND_ENTERED:
        if event.ui_element == debug_console:
            if event.command == 'godmode':
                print("Entering godmode")
                player.enable_collision = False
```

UITextBox - UI_TEXT_EFFECT_FINISHED

Fired when a command is entered into a console window. Usually by typing it in and pressing enter.

- **'type'** : pygame_gui.UI_TEXT_EFFECT_FINISHED
- **'effect'** : The type of the effect that ended.
- **'ui_element'** : The *UITextBox* that fired this event.
- **'ui_object_id'** : The most unique ID for the element that fired this event.

Example usage:

```
for event in pygame.event.get():
    if event.type == pygame_gui.UI_TEXT_EFFECT_FINISHED:
        if event.ui_element == dialog_box:
```

(continues on next page)

(continued from previous page)

```
            if event.effect == pygame_gui.TEXT_EFFECT_FADE_OUT:
                print("Dialog faded out")
```

7.5 Text Effects

When using either *UITextBox* or *UILabel* there is an option to add some of the below text effects that will control some aspect of how the text in the element is displayed.

These effects are:

Apply to all text in the Element, or an individual chunk of tagged text in a UITextBox:

- **Fade In** (ID: pygame_gui.TEXT_EFFECT_FADE_IN)
- **Fade Out** (ID: pygame_gui.TEXT_EFFECT_FADE_OUT)
- **Typing Appear** (ID: pygame_gui.TEXT_EFFECT_TYPING_APPEAR)

Apply ONLY to tagged chunks in a UITextBox:

- **Bounce** (ID: pygame_gui.TEXT_EFFECT_BOUNCE)
- **Tilt** (ID: pygame_gui.TEXT_EFFECT_TILT)
- **Expand & Contract** (ID: pygame_gui.TEXT_EFFECT_EXPAND_CONTRACT)

7.5.1 Applying an effect

To apply an effect to a *UITextBox* element call the set_active_effect method, with the id of the effect to apply:

```
text_box.set_active_effect(pygame_gui.TEXT_EFFECT_FADE_IN)
```

The effect will start running immediately after calling this method. You can deactivate an active effect by calling:

```
text_box.set_active_effect(None)
```

7.5.2 Applying an effect to a tagged chunk in a text box

First tag up the chunk of text you want to apply an effect to with the 'effect' tag. The text should have the same style throughout, otherwise you will get two chunks each running an effect. The 'id' parameter supplied to the effect tag is used to pick it out for an effect.

```
text_box = UITextBox(
    html_text="This is an <effect id=test>EARTHQUAKE</effect>",
    relative_rect=pygame.Rect(100, 100, 200, 50),
    manager=ui_manager)
```

Then when you want to apply an effect, supply the tag to set_active_effect like so:

```
text_box.set_active_effect(pygame_gui.TEXT_EFFECT_BOUNCE, effect_tag='test')
```

7.5.3 Event when an effect is finished

For effects that start, and then end after they have finished doing their thing - rather than looping forever, A GUI event is fired that you can check for in your event loop. It is called - UI_TEXT_EFFECT_FINISHED. You can read more about it in the *GUI events* documentation.

7.5.4 Effect parameters

Most of the effects have a parameter or two to give you some control over the effect. The parameters are listed below for each effect

7.5.5 Fade In

A simple alpha fade from translucent to opaque.

Params

- **'time_per_alpha_change'** - A float that controls the time in seconds to change alpha by 1 (to a maximum of 255)

7.5.6 Fade Out

The reverse of fade in, an alpha fade from opaque to translucent.

Params

- **'time_per_alpha_change'** - A float that controls the time in seconds to change alpha by 1 (to a maximum of 255)

7.5.7 Typing Appear

This effect makes the text in the box appear letter-by-letter as if someone was typing it in by hand.

Params

- **'time_per_letter'** - A float that controls the time in seconds to add a single letter.
- **'time_per_letter_deviation'** - A float that controls the deviation in time from the average that it takes to add a letter. Defaults to 0.0

7.5.8 Bounce

This effect makes the text in a chunk bounce up and down.

Params

- **'loop'** - A bool that sets whether this effect loops indefinitely (have to manually stop it). Defaults to true
- **'bounce_max_height'** - An int that controls how high the text bounces up from the baseline in pixels. Defaults to 5.
- **'time_to_complete_bounce'** - A float that controls the time in seconds to do a complete bounce. Defaults to 0.5.

7.5.9 Tilt

This effect makes the text in a chunk rotate to a maximum rotation angle and then return back to 0.

Params

- **'loop'** - A bool that sets whether this effect loops indefinitely (have to manually stop it). Defaults to true
- **'max_rotation'** - An int that controls how far the text rotates before returning to 0. Defaults to 1080.
- **'time_to_complete_rotation'** - A float that controls the time in seconds to do a complete tilt. Defaults to 5.0.

7.5.10 Expand & Contract

This effect makes the text in a chunk expand to a maximum scale and then return back to 1.0 (default size).

Params

- **'loop'** - A bool that sets whether this effect loops indefinitely (have to manually stop it). Defaults to true
- **'max_scale'** - An float that controls how much the text expands before returning to 1.0. Defaults to 1.5.
- **'time_to_complete_expand_contract'** - A float that controls the time in seconds to do a complete expansion and contraction. Defaults to 2.0.

7.6 Localization

Localization is the process of adapting your application for different languages & cultures. Pygame GUI provides some support for localizing your GUI, though no 'one size fits all' localization effort is going to be perfect, hopefully it will serve as a starting point for some users.

7.6.1 Underlying localization package

Pygame GUI uses python-i18n for it's localization backend, if you plan to make a lot of use out of the localization features in Pygame GUI you should check out the documentation for python-i18n as well. They cover more of the pluralization options than I get to in this brief guide.

7.6.2 Basics of switching locale

To start with, locales in Pygame GUI are specified via the IS0 639-1 two letter language codes. These are just a convenient, standardised way to refer to different languages.

You can set a starting locale for the GUI by passing one of these codes to the UIManager for your application like so:

```
manager = UIManager((800, 600), starting_language='ja')
```

Which would start the GUI in Japanese.

You can also switch the language of the GUI while it is running by calling *set_locale()* on the manager like so:

```
manager.set_locale('fr')
```

Which would switch the language to French, and make any other adaptions that might be needed for French speaking countries.

7.6.3 Default supported languages

Right now the list of languages support by default by Pygame GUI elements and windows is:

- Arabic
- German
- English
- Spanish
- French
- Georgian
- Hebrew
- Indonesian
- Italian
- Japanese
- Korean
- Polish
- Portuguese
- Russian
- Ukrainian
- Vietnamese
- Simplified Chinese

Though many of the translations may be imperfect as they were largely not handled by native speakers (yet). If you would like to improve an existing translation or add a new one the current default translations are stored here and should give you a good place to start. Pull requests, or simple issue reports of translation issues are very welcome.

7.6.4 Providing custom translation files to the GUI

What if you have your own text in your application that you want to translate into a few different languages? For that you can supply your own json translation file directories to the UI Manager. The files should be in the same format as the default translation files above. Here is a quick example of one from the examples:

Listing 29: examples.fr.json

```
{
  "fr": {
    "holmes_text_test": "<p>   Ce matin-là, M. Sherlock Holmes qui, sauf␣
↪les cas assez fréquents où il passait les nuits, se levait tard, était assis devant la␣
↪table de la salle à manger. Je me tenais près de la cheminée, examinant la canne que␣
↪notre visiteur de la veille avait oubliée. C'était un joli bâton, solide, terminé par␣
↪une boule - ce qu'on est convenu d'appeler « une permission de minuit ». Immédiatement␣
↪au-dessous de la pomme, un cercle d'or, large de deux centimètres, portait␣
↪l'inscription et la date suivantes : « À M. James Mortimer, ses amis du C. C. H. -␣
↪1884 ». Cette canne, digne, grave, rassurante, ressemblait à celles dont se servent␣
↪les médecins « vieux jeu ».</p>",
    "hello_world_message_text": "Bonjour le monde"
  }
}
```

Once you have your strings translated into the languages you want to support, you pass the directory they are in to the UIManager like so:

```
manager = UIManager((800, 600),
                    starting_language='en',
                    translation_directory_paths=['data/translations'])
```

Then for the strings in your GUI, you use the 'namespace' from the filenames ('examples' in the json file above), followed by the keys for each string separated with a full stop. Like so:

```
text_box = UITextBox(
        html_text="examples.holmes_text_test",
        relative_rect=pygame.Rect(300, 100, 400, 200),
        manager=manager)
```

To see a more complete example of the current localization setup see the translation_test script in the examples project on GitHub.

7.6.5 Providing custom fonts per locale

If you make use of custom fonts in your application, and also want to support localizations to languages that may not have their characters present in your initial custom font - you will be pleased to find out that you can specify a different custom font for a particular locale.

The setup in the theme file looks like this:

Listing 30: translations_theme.json

```
{
   "label":
   {
      "font":
      [
         {
             "name": "montserrat",
             "size": "12",
             "bold": "0",
             "italic": "0",
             "regular_resource": {"package": "data.fonts",
                                  "resource": "Montserrat-Regular.ttf"}
         },
         {
             "name": "kosugimaru",
             "locale": "ja",
             "size": "12",
             "bold": "0",
             "italic": "0",
             "regular_resource": {"package": "data.fonts",
                                  "resource": "KosugiMaru-Regular.ttf"}
         }
      ]
   }
}
```

Note that the font block now contains the square brackets for a list/array and the addition of a "locale" entry on the second font to designate it to be used instead by the japanese language.

As always, please let us know how you get on with localization using the library. It is a new feature for the library and undoubtedly has lots of bugs and areas that have yet to be considered.

7.7 Freezing with PyInstaller & Nuitka

Pygame GUI has some support for turning your application into an executable using either PyInstaller or Nuitka. Be aware however that these utilities working successfully is also dependent on the version of python you are using, support in any other libraries you are using (including pygame CE) and your own handling of data files.

As a general principle it is better to be slightly back from the bleeding edge of Python progress if you want to successfully freeze your python scripts into an exe. Both nuitka and PyInstaller are community projects and as such are dependent on the python community to keep pace with core python development. There is usually some amount of lag.

That out of the way, lets look at each of them more closely.

7.7.1 PyInstaller

First things first you will need to have the PyInstaller package from PyPI:

```
pip install pyinstaller
```

Once that is installed you need to create a .spec file for your project that will handle gathering up all the data files you are using and any other options you might wish to use. There are a lot of options for spec files and you can read more about them in the pyinstaller documentation.

Here is one I cooked up for the pyinstaller_example script in the pygame_gui_examples repository:

```
# -*- mode: python -*-

block_cipher = None


a = Analysis(['../pyinstaller_test.py'],
             pathex=[],
             binaries=[],
             datas=[],
             hiddenimports=[],
             hookspath=[],
             runtime_hooks=[],
             excludes=[],
             win_no_prefer_redirects=False,
             win_private_assemblies=False,
             cipher=block_cipher)

a.datas += Tree('data', prefix='data')

pyz = PYZ(a.pure, a.zipped_data,
          cipher=block_cipher)

# enable this to see verbose output
# options = [ ('v', None, 'OPTION')]
exe = EXE(pyz,
          a.scripts,
          # options,
          exclude_binaries=True,
          name='pyinstaller_test_release',
          debug=False, # set this to True for debug output
          strip=False,
          upx=True,
          console=True ) # set this to False this to remove the console
coll = COLLECT(exe,
               a.binaries,
               a.zipfiles,
               a.datas,
               strip=False,
               upx=True,
               name='pyinstaller_test_release')
```

Once you have your spec file, and made sure it includes any custom data you are using in your script (fonts, images, sounds, theme files etc) you are ready to try and compile the executable.

```
pyinstaller path_to_your_spec_file.spec
```

Pygame GUI now includes a Pyinstaller 'hook' file which should bring in all the data files used by Pygame GUI itself, however if you are using other packages that have their own data files you may have to manage this process yourself.

Good luck, and make yourself known on the GitHub issues page if you have any problems.

7.7.2 Nuitka

As with PyInstaller, when using Nuitka the first step is to download it:

```
pip install nuitka
```

Once you've done that you need to carefully specify the Nuitka command line to include all the data directories for pygame_gui (along with any data directories for any other libraries you are using).

The crucial line is:

```
--include-plugin-directory=pygame_gui/data
```

Here is an example command line:

```
nuitka --onefile --plugin-enable=numpy --plugin-enable=pylint-warnings --include-plugin-
↪directory=pygame_gui/data -o package/YourExeName.exe --output-dir=package
```

There haven't been as many users of the Pygame GUI asking about Nuitka yet, so I am not as familiar with any other issues that may arise. Please check out the official documentation for more information on Nuitka.

As always let us know any issues you have on the GitHub issues page.

7.8 Change List

A record of changes between versions of Pygame GUI.

7.8.1 Version 0.6.10

See the github release notes here.

Hello!

It's been almost a year since the last Pygame GUI release, a period when I've personally had less time to work on the library. However that has been compensated for by the lovely Pygame GUI community contributors with this release having contributions from ten new contributors - our largest number ever.

I had intended at one point for 0.7.0 to be the next release of the library with lots of changes to the handling of loaded images - but as things go this release happened instead, with the focus mainly on changes to handling of text.

Please let me know about any bugs, and I apologise in advance to anyone doing more complicated things with the library as I have messed around with the innards quite substantially again and probably broken a few things some of you might have been relying on. I promise I will stop when we hit version 1.0.0.

Big changes

- Switched from using pygame.freetype submodule to the pygame.font submodule internally. This has allowed us to support a wider range of written language features like right-to-left and text-shaping (via the pygame.font->SDLttf->Harfbuzz tech stack). As a result we can support languages like Arabic, Hebrew and Georgian. Though I will caveat that by saying I do not personally speak these languages, nor have any experience using GUIs designed with them in mind so I consider us at the start of the journey here for support of many of these non-latin alphabet languages. Please let me know how you get on, and PRs to adjust things are very welcome!
- Partially as a result of the above we have basic language support for six new languages Vietnamese, Korean, Georgian, Ukrainian, Arabic & Hebrew. That brings us up to 17 languages with basic translation support now.
- Text Selection overhaul. You can now select text and 'CTRL+C' shortcut copy text in regular Text boxes as well as Text Entry boxes, hopefully the experience of selecting text is smoother now as many bugs were located and squished in this area.
- **Three new UI Elements were added, thanks to our contributors:**
  - A Tabbed container, allowing you to switch between containers of elements with the other un-selected tabs kept hidden - Added by @LondonClass.
  - An Auto-resizing container that expands to fit whenever elements are added outside of it's boundaries - Added by @GimLala
  - A 2D slider - that lets you select a two-dimensional value in a rectangular space. It's been incorporated into the colour picker window if you want to try it out. - Added by @GimLala

7.8.2 Version 0.6.9

See the github release notes here.

Hello!

A new version of Pygame GUI has been released. The main reason is because of the switch over from old Pygame to the new Pygame CE distribution. This forked distribution of pygame is more up to date with developments in SDL and has already allowed this library to improve copy and paste handling and boost drawing performance. I'm hopeful it will eventually enable support of a wider range of languages too.

You can find more information on Pygame CE at its home page: https://pyga.me/ but it is currently a drop in replacement for old pygame so you shouldn't have too much to do to deal with the switch over.

I've also added a couple of minor new features and hopefully squished more bugs than I have created, you can read the full change list below.

7.8.3 Version 0.6.8

See the github release notes here.

Just a small release for the new year. A few documentation updates, un-draggable windows and a parameter for line spacing in UITextBoxes.

7.8.4 Version 0.6.7

See the github release notes here.

Hello!

We had a lovely large contribution since 0.6.6 was released and I didn't want to wait too long before putting it out.

Main Feature

@cobyj33 added a new colour parsing module which means there are many more ways to specify colours in theming files beyond the previous support for hex triplet style colours (e.g. # FF60AF). Now you can also specify colours as RGB/A 0-255 integers (e.g. (100, 128, 255)), as shortened hex (e.g. # F6A), or by HSL/A, HSV/A or CMY colour representations and finally there are some string colour names as well. @cobyj33' s PR [has an excellent table](https://github.com/MyreMylar/pygame_gui/pull/374) which gives examples in greater detail than I've put here.

Colours!

What else...

Snuck in a small improvement to tool tips and some house keeping to keep up with Python's new versions.

7.8.5 Version 0.6.6

See the github release notes here.

Just a quick bug fix release. We are also dropping active support for Python 3.6 as GitHub test runners have stopped supporting it making it very hard to test it it regularly.

Thanks to all our contributors!

7.8.6 Version 0.6.5 - Now with multi-line Text Entry

See the github release notes here.

Upgrade with: `pip install pygame_gui -U`

This update was building on the work done in 0.6.0 with text to finally add a multi-line text entry UI element, and along the way a range of other smaller issues were fixed.

Major Features

- **NEW `UITextEntryBox`** - Allows for the entering/editing of plain text into a word-wrapped, multi-line box. The aim was to model it closely on Windows Notepad in terms of features and behaviour.

Minor features & enhancements

- **NEW `TEXT_EFFECT_SHAKE`** - Another text effect that applies a shaking effect to words in text boxes.
- **Better support for key word arguments for translations** - most elements with text now support a text_kwargs type parameter that you can pass a dictionary of keywords too - useful if you have data to insert into translated strings.
- **NEW Polish translation** - Thanks to @marcinbarylka.
- **UIDropDown enhancements** - It should now open with the selected item visible in the list, allow scrolling the list with the mouse and allow for the list to be updated without having to recreate the drop down from scratch.
- **Dynamic theming changes** - there are now options to update the theming of elements via functions while the application is running.
- **Centre anchors for layouts** - should make it easier to stick elements in the middle of containers and have them stay there.
- **Reducing boilerplate code** - `UIManager`s should now be an optional parameter for all elements (you still need to make at least one), I've also begun some work on passing in a four element tuple instead of a pygame.Rect which should pare down the text passed into elements down to the very minimum required.

7.8.7 Version 0.6.4

Actually fix the slider this time...

7.8.8 Version 0.6.3

Quick release to fix the slider I broke in a new way in 0.6.2.

7.8.9 Version 0.6.2

Another smallish release, with some bug fixes and a new type of element - the UIStatusBar by @ConquerProgramming1. There should be a new example in the examples project demonstrating the new status bar

Bug Fixes & Other Changes

- **Fix bug with UITextLine background** - Should now remain the background colour when clicked on.
- **Fix bug with UISlider changed event** - They weren't generating on a single arrow button click.
- **Added fallback characters for hidden text int text entry line** - Some fonts don't have the circle character.
- **Added support for py.typed** - Thanks to @ChrisChou-freeman for mentioning this (there are still lots of mypy typing errors to fix)
- **Made K_KP_ENTER key behave the same as K_RETURN for UITextEntryLine** - Thanks to @Jamieakuma on the discord.
- **New theming option for the text cursor in the UITextLine** - No more hardcoded white, now you can match it to your UI theme a lot easier.
- **Add a 'pixel_size' option to the html subset font tag** - so you can set the font size directly.

Further thanks & Pull Requests

Several people stepped up to help improve the google translate localization efforts since the 0.6.0 release.

- **New UIStatusBar element** - Thanks to @ConquerProgramming1, see (pull #246)
- **Fixed typos in Russian localization** - Thanks to @SophieSilver, see (pull #241)
- **Fixes to UIManager docs** - Thanks again to @ConquerProgramming1, see (pull #240)
- **Change UIManager process_events() to return True if it uses the event** - Thanks once more to @ConquerProgramming1, see (pull #239)

7.8.10 Version 0.6.1

A quick bug fix release.

Bug Fixes & Other Changes

- **PyInstaller should work correctly with pygame_gui** - Fixed PyInstaller hook added in 0.6.0 not being picked up.
- **Added set_text() to UITextBox** - There were several requests for it.
- **Minor fixes to TextEffect interface** - Added default 'None' in a few places.
- **Fixed API docs not building on read** - the docs (thanks to @lionel42)
- **Made K_KP_ENTER key behave the same as K_RETURN for UITextEntryLine** - Thanks to @Jamieakuma on the discord.

Further thanks & Pull Requests

Several people stepped up to help improve the google translate localization efforts since the 0.6.0 release.

- **Improved Japanese localization** - Thanks to @KansaiGaijin, see (pull #231)
- **Improved Indonesian localization** - Thanks to @avaxar, see (pull #232)
- **Improved Russian localization** - Thanks to @SophieSilver, see (pull #237)

7.8.11 Version 0.6.0 - The text update

The focus of this update was on everything to do with text in the GUI

Dropped compatibility & Breaking changes

- **Dropped support for Pygame 1** - Pygame 2 has been out for some time now and switching fully to Pygame 2 allows the library to adopt its new features and remove some old compatibility hacks.
- **Dropped support for Python 3.5** - Python 3.5 has been end-of-life for some time. Removing support for it allows the library to use 3.6 onwards features like f strings. This is following pygame 2 also dropping 3.5 (and earlier) support.
- **Simplified UI events** - New events are generated with 'type' set to the previous 'user_type' values. This makes event processing code simpler. Old events will continue to exist until 0.8.0 but please move to the new style of events as they are the only ones that will get new attributes, new events added in 0.6.0 are only in the new style.

Major Features

- **Localization Support** - There is now some basic support for switching the language of the GUI to one of ten supported languages.
- **New Console Window** - A new default GUI element that provides support for text shell/console type user interaction.
- **Rewritten & unified text backend** - The text displaying and laying out portions of the GUI have all been massively changed and all the GUI elements now all share common code. This makes it easier to add new features to the text, and also have them work everywhere.

Minor features

- **UIButtons & UILabels can now scale based off their text** - passing in -1 for a dimension will cause that dimension to be set based on the height or width of the element's text.
- **More default options to allow only certain characters in UITextLine** - 'alpha_numeric' was added as an option for the latin alphabet. The underlying system was adjusted to allow for localised versions of these character sets, but these do not yet exist.
- **set_text_hidden() added to UITextLine** - To enable a 'password' style entry line.
- **text shadow theming options added to UIButton** - Previously these were only on the UILabel.
- **<img> tag images can now be added to to a UITextBox** - Makes it easier to wrap text around images and have inline images in text (colourful emoji?)
- **get_relative_mouse_pos() added to UIWindow** - gets a mouse position relative to the UIWindow you call it from.
- **UISlider now moves in customisable fixed increments when clicking arrow button** - Makes it easier to have precise sliders.
- **UIButton events can now be produced by any mouse button** - new 'mouse_button' attribute on button events & 'generate_click_events_from' parameter to UIButton.
- **UIDropDown open/close drop down button width added as theming option** - Called 'open_button_width'.
- **Text alignment theming options for UITextBox & UILabel** - See their theming pages for details.
- **Improved text effects** - Effects can now be applied to tagged chunks of text in a text box, some effects can also be applied to UILabels. There are parameters for effects, and an event fired when an effect finishes.

Bug Fixes & Other Changes

- **PyInstaller should work correctly with pygame_gui** - A 'hook' file has been added to scoop up the default data for pygame_gui, and documentation added on using Pyinstaller & Nuitka with the library. See (issue #166)
- **Fixed issue with window resolution changes** - Thanks to @lonelycorn (issue #215)
- **
 tag fixed to produce blank lines** - See (issue #217)
- **Fixed missing type cast in UIFileDialog** - Thanks to @GUI-GUY (issue #207)
- **Fixed issues with adding lines to bottom of UITextBox** - Demonstrated in new UIConsoleWindow window. See issues (issue #69) and (issue #78)
- **Fixed issues with positioning UIDropDown inside container** - See issues (issue #179) and (issue #153)
- **Improved scaling support** - I still don't have the hardware to test this properly, but thanks to @jlaumonier, see (issue #210) it should work a bit better.
- **Fixed html link click events firing multiple times in some circumstances** - Thanks to @RedFlames for finding and fixing this. See (issue #206)
- **Various documentation improvements and updates** - Thanks to everyone who pointed out things they didn't understand on GitHub, in Discord or in person. I've tried to make things clearer wherever I can. Keep letting me know when you get stuck!

Further thanks & Pull Requests

While I was very slowly rebuilding the text back end for 0.6.0 the library also received several pull requests that will now make their way into the released version. After 1.0.0, when I (@MyreMylar) finish my main work on it, pull requests like this will be the main way the library changes from version to version.

For now I'm putting them in their own section of this changes document to highlight them (unless there is a pull request that adds a big feature that is going up top as well)

- **Fixed redundant redrawing of UITextEntryLine()** - Thanks to @glipR, see (pull #178)
- **Fixed double clicking folder in UIFileDialog** - Thanks to @glipR, see (pull #197)
- **Fixed hiding & showing disabled buttons** - Thanks to @xirsoi, see (pull #185)
- **Fixed grammatical errors in index.rst** - Thanks to @nonoesimposible, see (pull #208)
- **Added ability to set default values for UISelectionList** - Thanks to @teaguejt, see (pull #213)
- **Fix invalid URL for game project examples** - Thanks to @Grimmys, see (pull #216)

7.8.12 Version 0.5.7 - Hiding and better pygame 2 support

Major Features

- **show() & hide() feature added to all elements**. Allows you to temporarily hide and show a UIElement or UIWindow rather than having to kill() and recreate it each time when you want it out of sight for a bit. This feature was contributed by @ylenard so all thanks goes to them.
- **switch to using premultiplied alpha blending for pygame 2** - For a long while now features like rounded corners have not worked correctly with pygame 2. Thanks to some recent improvements in the latest version of pygame 2.0.0.dev10 pygame_gui has been able to switch to using pre-multiplied alpha blending when dev10 is also installed. This resolves all the visual issues with rounded corners and I think runs a teeny bit faster too.

Minor features

- **enable() & disable() have been added to many more elements and windows** - Maybe all of them now, even where it doesn't really make sense. Disable things to your heart's content.
- **focus sets** - This is a new concept I'm trialling in the UI to indicate a group of elements that together constitute a thing that should all have interaction focus at the same time. So far it's working fairly well and has made it easy to extend pygame 2's scrollwheel functionality so that you should now scroll the content of what you are hovering with the wheel (at least in most cases). In the future this idea may make it easier to handle keyboard only input and input via controllers.
- **class IDs for UIElement objects** - UIelement objects could always have an Object ID, but those were designed to be unique specifiers for events as well as theming and sometimes you want to pick out a specific group of elements for theming that all already have unique object IDs. Enter class IDs, there is a new datatype 'ObjectID' that you can pass when you create an element and it lets you set two string IDs, the old unique *object_id* and the new *class_id*. Once you have some objects sharing a *class_id* you can theme theme in a theme file theming block the same way you would with an object ID. It's also worth noting here that you can load multiple theme files into a single UIManager if you want to organise your theme data some more.

Dropped compatibility

- **No longer supporting pygame 1.9.3 & pygame 1.9.4** - Keeping up with the bugs in these old versions of pygame was holding back the GUI so I made the decision to drop support in this version. If you are still using pygame 1.9.3 or 1.9.4, my apologies.

Bug Fixes & Other Changes

- **Switched to using a custom Sprite and SpriteGroup class as base for UI elements*** - previously I was using the pygame classes but after getting up close and personal with them recently I realised that the existing sprite base was doing things that we weren't using and that a slimmed down sprite could speed things up. In my tests on windows this has made the draw loop about 10% faster.
- **A series of fixes to the drop down menus** - they should now not break when they would have overlapped previously and correctly set the height of the background when the height of a list item is set to a custom value. Thanks to all the people who submitted bugs with these.
- **fixed a bunch of LGTM alerts** - gotta have that A+ rating.

Further thanks

- Thanks once again to @ylenard for all their hard work put into this release.
- Thank you to everyone who reported issues in the GUI this time around. If you don't report 'em, we can't fix 'em.

7.8.13 Version 0.5.6 - Loading changes & minor optimisations

Major Feature

- **Improved loading system - Pygame GUI now supports:**
  - **Incremental loading** - By passing in a loader you create yourself to the UIManager, you can get progress updates on how your GUI resources are loading. See *IncrementalThreadedResourceLoader* in *pygame_gui.core*, or the new loading examples in the [examples repository](https://github.com/MyreMylar/pygame_gui_examples).
  - **Loading resources from python packages** - This is, probably, the wave of the future for python projects. Instead of putting your resources in plain old directories and using boring file paths you can now add an exciting empty dunder *__init__.py* file to your resource directories, transforming them into packages which can then be loaded with a similar style to how we import code. There is a new *PackageResource* class at module scope to support this and some new ways to specify resources in theme files. See the [examples](https://github.com/MyreMylar/pygame_gui_examples) for a few usages and the [documentation](https://pygame-gui.readthedocs.io/en/latest/theme_reference/theme_button.html).
  - **Loading with threads** - As always with anything parallel, this comes with an extra frisson of danger. But in theory you should be able to see some improvement in how fast your resources are loaded. On my hard drive I've seen something like a 10% loading speed increase in my tests, but that can increase to almost 2x faster if your drive access speed is slow - as I discovered loading from a network drive. Care should probably be taken not to try and use any of the resources *while* they are being loaded as heck know what pygame will make of that. Threaded loading is enabled by default, so let us know if any problems crop up and I'll implement a fall-back, sequential-loading-only loader.

Breaking interface change

If you have any code that looks like this:

```
background.fill(manager.ui_theme.get_colour(None, None, 'dark_bg'))
```

Or

```
background.fill(manager.ui_theme.get_colour([], [], 'dark_bg'))
```

Then you will now have to change it to:

```
background.fill(manager.ui_theme.get_colour('dark_bg'))
```

This actually resulted from general optimisation changes but I think it is a solid improvement to the interface for getting default colours from a theme so I am enforcing it.

- **Custom UI elements** - If you've made any custom UI element classes (inheriting from UIElement) with their own theming then the procedure for getting theming IDs and theming parameters has changed slightly. You can see an example of adapting to these changes in the [pygame_paint repository here](https://github.com/MyreMylar/pygame_paint/commit/c5e7023bd0998b461b574f816b033dcf193399d3)

Bug Fixes & Other Changes

- The speed of creating 100+ buttons in a single frame should now be slightly faster than the 0.4.0 era of Pygame GUI rather than 3x *slower* (fix for #91)
- Mildly improved exception handling internally - This is an ongoing project.
- Abstract interface classes now properly enforce their interface on inheriting classes. Oops.

7.8.14 Version 0.5.5 - The Windows Update, Update

No major features, just a smattering of bug fixes, a few new elements and probably some new bugs.

New Elements

- **UIHorizontalScrollBar** - Just like the vertical scroll bar, but in the x axis.
- **UIScrollingContainer** - Another type of ContainerLike element. this one is largely invisible except for scroll bars that appear on the right hand side and at the bottom when the content inside the container is larger than the container itself.

Minor Features

- UIFileDialog has a couple of new options on creation mainly to support make file dialogs for loading and saving files. Probably still more bugs to find in this bad boy.
- New simple method to set the title of a window.
- New events for when text is changed in a text entry event, when a button is 'clicked once' (pushed down, but not yet released) to match the double click event and when buttons are hovered and unhovered.

Bug Fixes & Other Changes

- Added more interfaces to the code base which should make autocomplete more reliable when using the methods of the library.
- Fixed a bug with containers not using 'hover_point()' method for testing hovering collisions with the mouse thus messing up various interactions slightly.
- Fixed a bug with removing the close button on a window theme not correctly resizing the title bar.
- Changed UIElement to take a copy of passed in rectangles in case they are re-used elsewhere.
- Fixed bugs in UIPanel and UISelection list where anchors and containers of the element were not being copied to their root container leading to shenanigans.
- Resizing the element container for the UIWindow element was missing off the border leading to overlaps. This is now fixed.
- Fix for elements owning root containers anchored to the top and bottom of containers having their root containers incorrectly resized before they were positioned, thereby causing a mess of appearance bugs. It was a bad scene. Should now be fixed.

7.8.15 Version 0.5.1

7.8.16 Bug Fixes

- Getting the library working with pygame 1.9.3
- Removing window's title bar now works correctly.

7.8.17 Version 0.5.0 - The Windows Update

Major system features

- **Big UIWindow class refactoring**. UIWindow features like dragging windows, title bars and close buttons added as core features of the class. The class has moved from 'core' submodule to the 'elements' submodule. You can now create usable UIWindows without inheriting from the class first.
- **Windows now support dynamic user resizing**. You can grab corners and sides of windows and stretch them around.
- **Layout 'anchoring' system**. For laying out UI elements inside Containers (including Windows & Panels). This lets users place elements relative to other sides of their containers not just the default 'top left' every time.
- **Button state transition 'cross-fade' effect.** A bit of flash.
- **Theming files now support 'prototype' blocks.** To help reduce repetitive styling data. Theming parameter inheritance has also been changed to be more generous - e.g. now if you theme the 'button' block it will also affect buttons inside windows unless they have a more specific theming block.

New Elements

- **UISelectionList** - a list of elements that let users select either one, or multiple items on it depending on how it is configured.
- **UIPanel** - A new type of Container like element that you can place other elements inside of and set to start drawing at a specific layer in the UI. Designed for HUDs and the like.

New Windows

- **UIConfirmationDialog** - A Dialog Window which presents a choice to users to perform an action or cancel it.
- **UIFileDialog** - A Dialog that helps users navigate a file system and pick a file from it.
- **UIColourPickerDialog** - A Dialog window that lets you pick a colour.

Minor Features

- Drop down menu now supports larger lists of items in smaller space using a scroll bar and a parameter at creation to limit the vertical size. By default it will limit it's expansion to the boundaries of the container it is insider of.
- Drop downs can now be expanded by clicking on the selected item button as well as the little arrow.
- New theming options to remove the arrow buttons from horizontal sliders and vertical scroll bars.
- Layer debug function on the UI Manager that lets you inspect what's going on with the UI Layers.
- You can now set UIPanels and UIWindows as the 'container' parameter for all UIElements directly on creation.
- Lots of new UI events to support the new elements and a new one for when the horizontal slider has moved.

API Breaking changes

- Lots of stuff with UIWindow. It's moved submodules, it has lots of new features that previously had to be provided in sub classes or didn't exist anywhere. The container for elements now excludes the title bar, shadow and borders of the window. Adapting is largely a case of deleting code, but it's a job of work.
- UIMessageWindow has also changed a lot, it's now themed by it's object ID '#message_window' rather than an element ID like before, and it has lost lots of code to the underlying UIWindow class.
- Object IDs for UI Events have changed to be the most specific ID that can be found or the element that generates them. This means code that was checking previously for '#my_window_ok_button' will probably need to be changed to check for '#my_window.#my_window_ok_button' or, you could change the button object ID to make it something like: '#my_window.#ok_button' because that identifier will now be more unique which was the general goal of the change.
- Theming files may not perform exactly the same way they did before. Again, you can probably do lots of deleting if you make use of the prototype block system and I've tried to keep it mostly the same.
- Default parameters have changed for 'text_box' and 'button'.

I try to minimise API breaking changes, but before we hit 1.0.0 I'd rather make changes that improve the overall module than skip them and preserve an API that isn't working anymore.

Bug Fixes & Other Changes

- Images loaded by the theming system should now work in pyinstaller -onefile .exe builds.
- Drop down element should update the selected_option variable upon picking an option.
- set_position, set_relative_position and set_dimensions methods should now work much more consistently across all elements.
- Text boxes should expand correctly when the appropriate dimension is set to -1 or when the 'wrap_to_height' parameter is set to True on startup.
- Text entry line text selection is smoother now.
- UIContainer class now used all over the place - replacing the old 'root window' as 'root container', inside sliders & scroll bars.
- Lots of refactoring to please Python Linting tools flake8 and pylint. Always more work to do here, but the code should be a few percent cleaner now.
- Made use of interface/ABC meta classes to remove bothersome circular dependency problems.
- More tests. Always more tests.

- Text line documentation bug fixed by contributor **St3veR0nin**

7.9 API Reference

7.9.1 pygame_gui package

Subpackages

pygame_gui.core package

Subpackages

pygame_gui.core.drawable_shapes package

Submodules

pygame_gui.core.drawable_shapes.drawable_shape module

class pygame_gui.core.drawable_shapes.drawable_shape.**DrawableShape**(*containing_rect: Rect*, *theming_parameters: Dict*, *states: List[str]*, *manager:* IUIManagerInterface, ***, *allow_text_outside_width_border=True*, *allow_text_outside_height_border=True*, *text_x_scroll_enabled=False*, *editable_text=False*)

Bases: object

Base class for a graphical 'shape' that we can use for many different UI elements. The intent is to make it easy to switch between UI elements having normal rectangles, circles or rounded rectangles as their visual shape while having the same non-shape related functionality.

Parameters

- **containing_rect** -- The rectangle which this shape is entirely contained within (including shadows, borders etc)
- **theming_parameters** -- A dictionary of user supplied data that alters the appearance of the shape.
- **states** -- Names for the different states the shape can be in, each may have different sets of colours & images.
- **manager** -- The UI manager for this UI.

align_all_text_rows()

Aligns the text drawing position correctly according to our theming options.

apply_active_text_changes()

Updates the shape surface with any changes to the text surface. Useful when we've made small edits to the text surface

build_text_layout()

Build a text box layout for this drawable shape if it has some text.

clean_up_temp_shapes()

This method is declared for derived classes to implement but has no default implementation.

collide_point(*point: Vector2 | Tuple[int, int] | Tuple[float, float]*)

This method is declared for derived classes to implement but has no default implementation.

Parameters
point -- A point to collide with this shape.

finalise_images_and_text(*image_state_str: str*, *state_str: str*, *text_colour_state_str: str*, *text_shadow_colour_state_str: str*, *add_text: bool*)

Rebuilds any text or image used by a specific state in the drawable shape. Effectively this means adding them on top of whatever is already in the state's surface. As such it should generally be called last in the process of building up a finished drawable shape state.

Parameters

- **add_text** --
- **image_state_str** -- image ID of the state we are going to be adding images and text to.
- **state_str** -- normal ID of the state we are going to be adding images and text to.
- **text_colour_state_str** -- text ID of the state we are going to be adding images and text to.
- **text_shadow_colour_state_str** -- text shadow ID of the state we are going to be adding images and text to.

finalise_text(*state_str*, *text_colour_state_str: str = ''*, *text_shadow_colour_state_str: str = ''*, *only_text_changed: bool = False*)

Finalise the text to a surface with some last-minute data that doesn't require the text be re-laid out.

Parameters

- **only_text_changed** --
- **state_str** -- The name of the shape's state we are finalising.
- **text_colour_state_str** -- The string identifying the text colour to use.
- **text_shadow_colour_state_str** -- The string identifying the text shadow colour to use.

finalise_text_onto_active_state()

Lets us draw the active state with no text and then paste the finalised surface from the text layout on top. Handy if we are doing some text effects in the text layout we don't want to lose by recreating the text from scratch.

full_rebuild_on_size_change()

Triggered when we've changed the size of the shape and need to rebuild basically everything to account for it.

get_active_state_surface() → Surface

Get the main surface from the active state.

Returns
The surface asked for, or the best available substitute.

get_fresh_surface() → Surface

Gets the surface of the active state and resets the state's 'has_fresh_surface' variable.

Returns
The active state's main pygame.surface.Surface.

get_surface(*state_name: str*) → Surface

Get the main surface from a specific state.

Parameters
state_name -- The state we are trying to get the surface from.

Returns
The surface asked for, or the best available substitute.

has_fresh_surface() → bool

Lets UI elements find out when a state has finished building a fresh surface for times when we have to delay it for whatever reason.

Returns
True if there is a freshly built surface waiting, False if the shape has not changed.

insert_text(*text: str*, *layout_index: int*, *parser: HTMLParser | None = None*)

Update the theming when we insert text, then pass down to the layout to do the actual inserting. :param text: the text to insert :param layout_index: where to insert it :param parser: an optional parser :return:

redraw_active_state_no_text()

Redraw the currently active state with no text.

redraw_all_states(*force_full_redraw: bool = False*)

Starts the redrawing process for all states of this shape that auto pre-generate. Redrawing is done one state at a time so will take a few loops of the game to complete if this shape has many states.

redraw_state(*state_str: str*, *add_text: bool = True*)

This method is declared for derived classes to implement but has no default implementation.

Parameters

- **add_text** --
- **state_str** -- The ID/name of the state to redraw.

set_active_state(*state_id: str*)

Changes the currently active state for the drawable shape and, if setup in the theme, creates a transition blend from the previous state to the newly active one.

Parameters
state_id -- the ID of the new state to make active.

set_dimensions(*dimensions: Vector2 | Tuple[int, int] | Tuple[float, float]*)

This method is declared for derived classes to implement but has no default implementation.

Parameters
dimensions -- The new dimensions for our shape.

set_position(*point: Vector2 | Tuple[int, int] | Tuple[float, float]*)

This method is declared for derived classes to implement but has no default implementation.

Parameters
point -- A point to set this shapes position to.

set_text(*text: str*)

Set the visible text that the drawable shape has on it. This call will build a text layout and then redraw the final shape with the new, laid out text on top.

Parameters
text -- the new string of text to stick on the shape.

set_text_alpha(*alpha: int*)

Set the alpha of just the text and redraw the shape with the new text on top.

Parameters
alpha -- the alpha to set.

toggle_text_cursor()

Toggle the edit text cursor/carat between visible and invisible. Usually this is run to make the cursor appear to flash so it catches user attention.

update(*time_delta: float*)

Updates the drawable shape to process rebuilds and update blends between states.

Parameters
time_delta -- amount of time passed between now and the previous frame in seconds.

class pygame_gui.core.drawable_shapes.drawable_shape.**DrawableShapeState**(*state_id: str*)

Bases: object

Represents a single state of a drawable shape.

Parameters
state_id -- The ID/name of this state.

get_surface() → Surface

Gets the pygame.surface.Surface of this state. Will be a blend of this state and the previous one if we are in a transition.

Returns
A pygame Surface for this state.

update(*time_delta: float*)

Updates any transitions this state is in

Parameters
time_delta -- The time passed between frames, measured in seconds.

class pygame_gui.core.drawable_shapes.drawable_shape.**DrawableStateTransition**(*states: Dict[str, DrawableShapeState]*, *start_state_id: str*, *target_state_id: str*, *duration: float*, *, *progress: float = 0.0*)

Bases: object

Starts & controls a transition between two states of a drawable shape.

Parameters

- **states** -- A dictionary of all the drawable states.

- **start_state_id** -- The state to start from.
- **target_state_id** -- The state to transition to.
- **duration** -- The length of the transition
- **progress** -- The initial progress along the transition.

produce_blended_result() → Surface

Produces a blend between the images of our start state and our target state. The progression of the blend is dictated by the progress of time through the transition.

Returns
The blended surface.

update(*time_delta: float*)

Updates the timer for this transition.

Parameters
time_delta -- The time passed between frames, measured in seconds.

pygame_gui.core.drawable_shapes.ellipse_drawable_shape module

class pygame_gui.core.drawable_shapes.ellipse_drawable_shape.**EllipseDrawableShape**(*containing_rect: Rect*, *theming_parameters: Dict[str, Any]*, *states: List[str]*, *manager: IUIManagerInterface*, *, *allow_text_outside_width_bo allow_text_outside_height_bo text_x_scroll_enabled=Fals editable_text=False*)

Bases: *DrawableShape*

A drawable ellipse shape for the UI, has theming options for a border, a shadow, colour gradients and text.

Parameters

- **containing_rect** -- The layout rectangle that surrounds and controls the size of this shape.
- **theming_parameters** -- Various styling parameters that control the final look of the shape.
- **states** -- The different UI states the shape can be in. Shapes have different surfaces for each state.
- **manager** -- The UI manager.

static clear_and_create_shape_surface(*surface: Surface*, *rect: Rect*, *overlap: int*, *aa_amount: int*, *clear: bool = True*) → Surface

Clear a space for a new shape surface on the main state surface for this state. The surface created will be plain white so that it can be easily multiplied with a colour surface.

Parameters

- **surface** -- The surface we are working on.
- **rect** -- Used to size and position the new shape.
- **overlap** -- The amount of overlap between this surface and the one below.
- **aa_amount** -- The amount of Anti Aliasing to use for this shape.
- **clear** -- Whether we should clear our surface.

Returns

The new shape surface.

collide_point(*point: Vector2 | Tuple[int, int] | Tuple[float, float]*) → bool

Checks collision between a point and this ellipse.

Parameters

point -- The point to test against the shape.

Returns

True if the point is inside the shape.

full_rebuild_on_size_change()

Completely redraw the shape from it's theming parameters and dimensions.

redraw_state(*state_str: str*, *add_text: bool = True*)

Redraws the shape's surface for a given UI state.

Parameters

- **add_text** -- Whether to add the text to the shape in this redraw.
- **state_str** -- The ID string of the state to rebuild.

set_dimensions(*dimensions: Vector2 | Tuple[int, int] | Tuple[float, float]*)

Expensive size change of the ellipse shape.

Parameters

dimensions -- The new size to set the shape to.

set_position(*point: Vector2 | Tuple[int, int] | Tuple[float, float]*)

Move the shape. Only really impacts the position of the 'click_area' hot spot.

Parameters

point -- The new position to move it to.

pygame_gui.core.drawable_shapes.rect_drawable_shape module

class pygame_gui.core.drawable_shapes.rect_drawable_shape.**RectDrawableShape**(*containing_rect: Rect*, *theming_parameters: Dict[str, Any]*, *states: List[str]*, *manager: IUIManagerInterface*, *, allow_text_outside_width_border=Tr allow_text_outside_height_border=Ti text_x_scroll_enabled=False*, *editable_text=False*)

Bases: *DrawableShape*

A rectangle shape for UI elements has theming options for a border, a shadow, colour gradients and text.

Parameters

- **containing_rect** -- The layout rectangle that surrounds and controls the size of this shape.
- **theming_parameters** -- Various styling parameters that control the final look of the shape.
- **states** -- The different UI states the shape can be in. Shapes have different surfaces for each state.
- **manager** -- The UI manager.

collide_point(*point: Vector2 | Tuple[int, int] | Tuple[float, float]*) → bool

Tests if a point is colliding with our Drawable shape's 'click area' hot spot.

Parameters
point -- The point to test.

Returns
True if we are colliding.

full_rebuild_on_size_change()

Completely rebuilds the rectangle shape from it's dimensions and parameters.

Everything needs rebuilding if we change the size of the containing rectangle.

redraw_state(*state_str: str*, *add_text: bool = True*)

Redraws the shape's surface for a given UI state.

Parameters

- **add_text** --
- **state_str** -- The ID string of the state to rebuild.

set_dimensions(*dimensions: Vector2 | Tuple[int, int] | Tuple[float, float]*)

Changes the size of the rectangle shape. Relatively expensive to do.

Parameters
dimensions -- The new dimensions.

set_position(*point: Vector2 | Tuple[int, int] | Tuple[float, float]*)

Move the shape. Only really impacts the position of the 'click_area' hot spot.

Parameters

point -- The new position to move it to.

pygame_gui.core.drawable_shapes.rounded_rect_drawable_shape module

class pygame_gui.core.drawable_shapes.rounded_rect_drawable_shape.**RoundedRectangleShape**(*containing_rect: Rect*, *theming_parameters: Dict[str, Any]*, *states: List[str]*, *manager: IUIManagerInterface*, *, allow_text_outside_w allow_text_outside_he text_x_scroll_enabled editable_text=False*)

Bases: *DrawableShape*

A drawable rounded rectangle shape for the UI, has theming options for a border, a shadow, colour gradients and text.

Parameters

- **containing_rect** -- The layout rectangle that surrounds and controls the size of this shape.
- **theming_parameters** -- Various styling parameters that control the final look of the shape.
- **states** -- The different UI states the shape can be in. Shapes have different surfaces for each state.
- **manager** -- The UI manager.

clean_up_temp_shapes()

Clean up some temporary surfaces we use repeatedly when rebuilding multiple states of the shape but have no need of afterward.

clear_and_create_shape_surface(*surface: Surface*, *rect: Rect*, *overlap: int*, *corner_radii: List[int]*, *aa_amount: int*, *clear: bool = True*) → Surface

Clear a space for a new shape surface on the main state surface for this state. The surface created will be plain white so that it can be easily multiplied with a colour surface.

Parameters

- **surface** -- The surface we are working on.
- **rect** -- Used to size and position the new shape.
- **overlap** -- The amount of overlap between this surface and the one below.
- **corner_radii** -- The radii of the rounded corners.
- **aa_amount** -- The amount of Anti Aliasing to use for this shape.
- **clear** -- Whether we should clear our surface.

Returns
The new shape surface.

collide_point(*point: Vector2 | Tuple[int, int] | Tuple[float, float]*) → bool

Checks collision between a point and this rounded rectangle.

Parameters
point -- The point to test collision with.

Returns
True, if the point is inside the shape.

create_subtract_surface(*subtract_size: Tuple[int, int]*, *corner_radii: List[int]*, *aa_amount: int*)

Create a rounded rectangle shaped surface that can be used to subtract everything from a surface to leave a transparent hole in it.

static draw_colourless_rounded_rectangle(*large_corner_radius: List[int]*, *large_shape_surface: Surface*, *corner_offset: int = 0*)

Draw a rounded rectangle shape in pure white, so it is ready to be multiplied by a colour or gradient.

Parameters

- **large_corner_radius** -- The radius of the corners.
- **large_shape_surface** -- The surface to draw onto, the shape fills the surface.
- **corner_offset** -- Offsets the corners, used to help avoid overlaps that look bad.

full_rebuild_on_size_change()

Completely rebuilds the rounded rectangle shape from its dimensions and parameters.

Everything needs rebuilding if we change the size of the containing rectangle.

redraw_state(*state_str: str*, *add_text: bool = True*)

Redraws the shape's surface for a given UI state.

Parameters

- **add_text** --
- **state_str** -- The ID string of the state to rebuild.

set_dimensions(*dimensions: Vector2 | Tuple[int, int] | Tuple[float, float]*)

Changes the size of the rounded rectangle shape. Relatively expensive to completely do so has support for 'temporary rapid resizing' while the dimensions are being changed frequently.

Parameters
dimensions -- The new dimensions.

set_position(*point: Vector2 | Tuple[int, int] | Tuple[float, float]*)

Move the shape. Only really impacts the position of the 'click_area' hot spot.

Parameters
point -- The new position to move it to.

Module contents

class pygame_gui.core.drawable_shapes.**DrawableShape**(*containing_rect: Rect*, *theming_parameters: Dict*, *states: List[str]*, *manager: IUIManagerInterface*, *, *allow_text_outside_width_border=True*, *allow_text_outside_height_border=True*, *text_x_scroll_enabled=False*, *editable_text=False*)

Bases: object

Base class for a graphical 'shape' that we can use for many different UI elements. The intent is to make it easy to switch between UI elements having normal rectangles, circles or rounded rectangles as their visual shape while having the same non-shape related functionality.

Parameters

- **containing_rect** -- The rectangle which this shape is entirely contained within (including shadows, borders etc)
- **theming_parameters** -- A dictionary of user supplied data that alters the appearance of the shape.
- **states** -- Names for the different states the shape can be in, each may have different sets of colours & images.
- **manager** -- The UI manager for this UI.

align_all_text_rows()

Aligns the text drawing position correctly according to our theming options.

apply_active_text_changes()

Updates the shape surface with any changes to the text surface. Useful when we've made small edits to the text surface

build_text_layout()

Build a text box layout for this drawable shape if it has some text.

clean_up_temp_shapes()

This method is declared for derived classes to implement but has no default implementation.

collide_point(*point: Vector2 | Tuple[int, int] | Tuple[float, float]*)

This method is declared for derived classes to implement but has no default implementation.

Parameters
point -- A point to collide with this shape.

finalise_images_and_text(*image_state_str: str*, *state_str: str*, *text_colour_state_str: str*, *text_shadow_colour_state_str: str*, *add_text: bool*)

Rebuilds any text or image used by a specific state in the drawable shape. Effectively this means adding them on top of whatever is already in the state's surface. As such it should generally be called last in the process of building up a finished drawable shape state.

Parameters

- **add_text** --

- **image_state_str** -- image ID of the state we are going to be adding images and text to.
- **state_str** -- normal ID of the state we are going to be adding images and text to.
- **text_colour_state_str** -- text ID of the state we are going to be adding images and text to.
- **text_shadow_colour_state_str** -- text shadow ID of the state we are going to be adding images and text to.

finalise_text(*state_str*, *text_colour_state_str: str = ''*, *text_shadow_colour_state_str: str = ''*, *only_text_changed: bool = False*)

Finalise the text to a surface with some last-minute data that doesn't require the text be re-laid out.

Parameters

- **only_text_changed** --
- **state_str** -- The name of the shape's state we are finalising.
- **text_colour_state_str** -- The string identifying the text colour to use.
- **text_shadow_colour_state_str** -- The string identifying the text shadow colour to use.

finalise_text_onto_active_state()

Lets us draw the active state with no text and then paste the finalised surface from the text layout on top. Handy if we are doing some text effects in the text layout we don't want to lose by recreating the text from scratch.

full_rebuild_on_size_change()

Triggered when we've changed the size of the shape and need to rebuild basically everything to account for it.

get_active_state_surface() → Surface

Get the main surface from the active state.

Returns
The surface asked for, or the best available substitute.

get_fresh_surface() → Surface

Gets the surface of the active state and resets the state's 'has_fresh_surface' variable.

Returns
The active state's main pygame.surface.Surface.

get_surface(*state_name: str*) → Surface

Get the main surface from a specific state.

Parameters
state_name -- The state we are trying to get the surface from.

Returns
The surface asked for, or the best available substitute.

has_fresh_surface() → bool

Lets UI elements find out when a state has finished building a fresh surface for times when we have to delay it for whatever reason.

Returns
True if there is a freshly built surface waiting, False if the shape has not changed.

insert_text(*text: str*, *layout_index: int*, *parser: HTMLParser | None = None*)

Update the theming when we insert text, then pass down to the layout to do the actual inserting. :param text: the text to insert :param layout_index: where to insert it :param parser: an optional parser :return:

redraw_active_state_no_text()

Redraw the currently active state with no text.

redraw_all_states(*force_full_redraw: bool = False*)

Starts the redrawing process for all states of this shape that auto pre-generate. Redrawing is done one state at a time so will take a few loops of the game to complete if this shape has many states.

redraw_state(*state_str: str*, *add_text: bool = True*)

This method is declared for derived classes to implement but has no default implementation.

Parameters

- **add_text** --
- **state_str** -- The ID/name of the state to redraw.

set_active_state(*state_id: str*)

Changes the currently active state for the drawable shape and, if setup in the theme, creates a transition blend from the previous state to the newly active one.

Parameters

state_id -- the ID of the new state to make active.

set_dimensions(*dimensions: Vector2 | Tuple[int, int] | Tuple[float, float]*)

This method is declared for derived classes to implement but has no default implementation.

Parameters

dimensions -- The new dimensions for our shape.

set_position(*point: Vector2 | Tuple[int, int] | Tuple[float, float]*)

This method is declared for derived classes to implement but has no default implementation.

Parameters

point -- A point to set this shapes position to.

set_text(*text: str*)

Set the visible text that the drawable shape has on it. This call will build a text layout and then redraw the final shape with the new, laid out text on top.

Parameters

text -- the new string of text to stick on the shape.

set_text_alpha(*alpha: int*)

Set the alpha of just the text and redraw the shape with the new text on top.

Parameters

alpha -- the alpha to set.

toggle_text_cursor()

Toggle the edit text cursor/carat between visible and invisible. Usually this is run to make the cursor appear to flash so it catches user attention.

update(*time_delta: float*)

Updates the drawable shape to process rebuilds and update blends between states.

Parameters

time_delta -- amount of time passed between now and the previous frame in seconds.

class pygame_gui.core.drawable_shapes.**EllipseDrawableShape**(*containing_rect: Rect*, *theming_parameters: Dict[str, Any]*, *states: List[str]*, *manager: IUIManagerInterface*, ***, *allow_text_outside_width_border=True*, *allow_text_outside_height_border=True*, *text_x_scroll_enabled=False*, *editable_text=False*)

Bases: *DrawableShape*

A drawable ellipse shape for the UI, has theming options for a border, a shadow, colour gradients and text.

Parameters

- **containing_rect** -- The layout rectangle that surrounds and controls the size of this shape.
- **theming_parameters** -- Various styling parameters that control the final look of the shape.
- **states** -- The different UI states the shape can be in. Shapes have different surfaces for each state.
- **manager** -- The UI manager.

static clear_and_create_shape_surface(*surface: Surface*, *rect: Rect*, *overlap: int*, *aa_amount: int*, *clear: bool = True*) → Surface

Clear a space for a new shape surface on the main state surface for this state. The surface created will be plain white so that it can be easily multiplied with a colour surface.

Parameters

- **surface** -- The surface we are working on.
- **rect** -- Used to size and position the new shape.
- **overlap** -- The amount of overlap between this surface and the one below.
- **aa_amount** -- The amount of Anti Aliasing to use for this shape.
- **clear** -- Whether we should clear our surface.

Returns
The new shape surface.

collide_point(*point: Vector2 | Tuple[int, int] | Tuple[float, float]*) → bool

Checks collision between a point and this ellipse.

Parameters
point -- The point to test against the shape.

Returns
True if the point is inside the shape.

full_rebuild_on_size_change()

Completely redraw the shape from it's theming parameters and dimensions.

redraw_state(*state_str: str*, *add_text: bool = True*)

Redraws the shape's surface for a given UI state.

Parameters

- **add_text** -- Whether to add the text to the shape in this redraw.
- **state_str** -- The ID string of the state to rebuild.

set_dimensions(*dimensions: Vector2 | Tuple[int, int] | Tuple[float, float]*)

Expensive size change of the ellipse shape.

Parameters
dimensions -- The new size to set the shape to.

set_position(*point: Vector2 | Tuple[int, int] | Tuple[float, float]*)

Move the shape. Only really impacts the position of the 'click_area' hot spot.

Parameters
point -- The new position to move it to.

class pygame_gui.core.drawable_shapes.**RectDrawableShape**(*containing_rect: Rect*, *theming_parameters: Dict[str, Any]*, *states: List[str]*, *manager: IUIManagerInterface*, ***, *allow_text_outside_width_border=True*, *allow_text_outside_height_border=True*, *text_x_scroll_enabled=False*, *editable_text=False*)

Bases: *DrawableShape*

A rectangle shape for UI elements has theming options for a border, a shadow, colour gradients and text.

Parameters

- **containing_rect** -- The layout rectangle that surrounds and controls the size of this shape.
- **theming_parameters** -- Various styling parameters that control the final look of the shape.
- **states** -- The different UI states the shape can be in. Shapes have different surfaces for each state.
- **manager** -- The UI manager.

collide_point(*point: Vector2 | Tuple[int, int] | Tuple[float, float]*) → bool

Tests if a point is colliding with our Drawable shape's 'click area' hot spot.

Parameters
point -- The point to test.

Returns
True if we are colliding.

full_rebuild_on_size_change()

Completely rebuilds the rectangle shape from it's dimensions and parameters.

Everything needs rebuilding if we change the size of the containing rectangle.

redraw_state(*state_str: str*, *add_text: bool = True*)

Redraws the shape's surface for a given UI state.

Parameters

- **add_text** --
- **state_str** -- The ID string of the state to rebuild.

set_dimensions(*dimensions: Vector2 | Tuple[int, int] | Tuple[float, float]*)

Changes the size of the rectangle shape. Relatively expensive to do.

Parameters
dimensions -- The new dimensions.

set_position(*point: Vector2 | Tuple[int, int] | Tuple[float, float]*)

Move the shape. Only really impacts the position of the 'click_area' hot spot.

Parameters

point -- The new position to move it to.

class pygame_gui.core.drawable_shapes.**RoundedRectangleShape**(*containing_rect: Rect*, *theming_parameters: Dict[str, Any]*, *states: List[str]*, *manager: IUIManagerInterface*, ***, *allow_text_outside_width_border=True*, *allow_text_outside_height_border=True*, *text_x_scroll_enabled=False*, *editable_text=False*)

Bases: *DrawableShape*

A drawable rounded rectangle shape for the UI, has theming options for a border, a shadow, colour gradients and text.

Parameters

- **containing_rect** -- The layout rectangle that surrounds and controls the size of this shape.
- **theming_parameters** -- Various styling parameters that control the final look of the shape.
- **states** -- The different UI states the shape can be in. Shapes have different surfaces for each state.
- **manager** -- The UI manager.

clean_up_temp_shapes()

Clean up some temporary surfaces we use repeatedly when rebuilding multiple states of the shape but have no need of afterward.

clear_and_create_shape_surface(*surface: Surface*, *rect: Rect*, *overlap: int*, *corner_radii: List[int]*, *aa_amount: int*, *clear: bool = True*) → Surface

Clear a space for a new shape surface on the main state surface for this state. The surface created will be plain white so that it can be easily multiplied with a colour surface.

Parameters

- **surface** -- The surface we are working on.
- **rect** -- Used to size and position the new shape.
- **overlap** -- The amount of overlap between this surface and the one below.
- **corner_radii** -- The radii of the rounded corners.
- **aa_amount** -- The amount of Anti Aliasing to use for this shape.
- **clear** -- Whether we should clear our surface.

Returns

The new shape surface.

collide_point(*point: Vector2 | Tuple[int, int] | Tuple[float, float]*) → bool

Checks collision between a point and this rounded rectangle.

Parameters

point -- The point to test collision with.

Returns
True, if the point is inside the shape.

`create_subtract_surface`(*subtract_size: Tuple[int, int]*, *corner_radii: List[int]*, *aa_amount: int*)

Create a rounded rectangle shaped surface that can be used to subtract everything from a surface to leave a transparent hole in it.

`static draw_colourless_rounded_rectangle`(*large_corner_radius: List[int]*, *large_shape_surface: Surface*, *corner_offset: int = 0*)

Draw a rounded rectangle shape in pure white, so it is ready to be multiplied by a colour or gradient.

Parameters

- **large_corner_radius** -- The radius of the corners.
- **large_shape_surface** -- The surface to draw onto, the shape fills the surface.
- **corner_offset** -- Offsets the corners, used to help avoid overlaps that look bad.

`full_rebuild_on_size_change`()

Completely rebuilds the rounded rectangle shape from its dimensions and parameters.

Everything needs rebuilding if we change the size of the containing rectangle.

`redraw_state`(*state_str: str*, *add_text: bool = True*)

Redraws the shape's surface for a given UI state.

Parameters

- **add_text** --
- **state_str** -- The ID string of the state to rebuild.

`set_dimensions`(*dimensions: Vector2 | Tuple[int, int] | Tuple[float, float]*)

Changes the size of the rounded rectangle shape. Relatively expensive to completely do so has support for 'temporary rapid resizing' while the dimensions are being changed frequently.

Parameters
dimensions -- The new dimensions.

`set_position`(*point: Vector2 | Tuple[int, int] | Tuple[float, float]*)

Move the shape. Only really impacts the position of the 'click_area' hot spot.

Parameters
point -- The new position to move it to.

pygame_gui.core.interfaces package

Submodules

pygame_gui.core.interfaces.container_interface module

`class pygame_gui.core.interfaces.container_interface.IContainerLikeInterface`

Bases: `object`

A metaclass that defines the interface for containers used by elements.

This interface lets us treat classes like UIWindows and UIPanels like containers for elements even though they actually pass this functionality off to the proper UIContainer class.

are_contents_hovered() → bool

Are any of the elements in the container hovered? Used for handling mousewheel events.

Returns
True if one of the elements is hovered, False otherwise.

abstract get_container() → *IUIContainerInterface*

Gets an actual container from this container-like UI element.

abstract hide()

Hides the container, which means the container will not get drawn and will not process events. Should also hide all the children elements. If the container was hidden before - ignore.

abstract show()

Shows the container, which means the container will get drawn and will process events. Should also show all the children elements. If the container was visible before - ignore.

class pygame_gui.core.interfaces.container_interface.**IUIContainerInterface**

Bases: *IUIElementInterface*

Interface for the actual container class. Not to be confused with the IContainerLikeInterface which is an interface for all the things we can treat like containers when creating elements.

abstract add_element(*element:* IUIElementInterface)

Add a UIElement to the container. The UI's relative_rect parameter will be relative to this container.

Parameters
element -- A UIElement to add to this container.

calc_add_element_changes_thickness(*element:* IUIElementInterface)

This function checks if a single added element will increase the containers thickness and if so updates containers recursively.

Parameters
element -- the element to check.

abstract change_layer(*new_layer: int*)

Change the layer of this container. Layers are used by the GUI to control the order in which things are drawn and which things should currently be interactive (so you can't interact with things behind other things).

This particular method is most often used to shift the visible contents of a window in front of any others when it is moved to the front of the window stack.

Parameters
new_layer -- The layer to move our container to.

abstract check_hover(*time_delta: float*, *hovered_higher_element: bool*) → bool

A method that helps us to determine which, if any, UI Element is currently being hovered by the mouse.

Parameters

- **time_delta** -- A float, the time in seconds between the last call to this function and now (roughly).
- **hovered_higher_element** -- A boolean, representing whether we have already hovered a 'higher' element.

Returns
A boolean that is true if we have hovered a UI element, either just now or before this method.

abstract clear()

Removes and kills all the UI elements inside this container.

abstract get_image_clipping_rect() → Rect | None

Obtain the current image clipping rect.

Returns
The current clipping rect. It may be None.

abstract get_rect() → Rect

Access to the container's rect

Returns
a pygame rectangle

abstract get_size() → Tuple[int, int]

Get the container's pixel size.

Returns
the pixel size as tuple [x, y]

abstract get_thickness() → int

Get the container's layer thickness.

Returns
the thickness as an integer.

abstract get_top_layer() → int

Assuming we have correctly calculated the 'thickness' of this container, this method will return the 'highest' layer in the LayeredDirty UI Group.

Returns
An integer representing the current highest layer being used by this container.

abstract kill()

Overrides the standard kill method of UI Elements (and pygame sprites beyond that) to also call the kill method on all contained UI Elements.

on_contained_elements_changed(*target:* IUIElementInterface) → None

Update the contents of this container that one of their layout anchors may have moved, or been resized.

Parameters
target -- the UI element that has been benn moved or resized.

abstract recalculate_container_layer_thickness()

This function will iterate through the elements in our container and determine the maximum 'height' that they reach in the 'layer stack'. We then use that to determine the overall 'thickness' of this container. The thickness value is used to determine where to place overlapping windows in the layers

abstract remove_element(*element:* IUIElementInterface)

Remove a UIElement from this container.

Parameters
element -- A UIElement to remove from this container.

abstract set_dimensions(*dimensions: Vector2 | Tuple[float, float]*, *clamp_to_container: bool = False*)

Set the dimension of this container and update the positions of elements within it accordingly.

Parameters

- **clamp_to_container** --

- **dimensions** -- the new dimensions.

abstract set_position(*position: Vector2 | Tuple[float, float]*)

Set the absolute position of this container - it is usually less chaotic to deal with setting relative positions.

Parameters

position -- the new absolute position to set.

abstract set_relative_position(*position: Vector2 | Tuple[float, float]*)

Set the position of this container, relative to the container it is within.

Parameters

position -- the new relative position to set.

abstract update_containing_rect_position()

This function is called when we move the container to update all the contained UI Elements to move as well.

pygame_gui.core.interfaces.manager_interface module

class pygame_gui.core.interfaces.manager_interface.**IUIManagerInterface**

Bases: object

A metaclass that defines the interface that a UI Manager uses.

Interfaces like this help us evade cyclical import problems by allowing us to define the actual manager class later on and have it make use of the classes that use the interface.

abstract add_font_paths(*font_name: str*, *regular_path: str*, *bold_path: str | None = None*, *italic_path: str | None = None*, *bold_italic_path: str | None = None*)

Add file paths for custom fonts you want to use in the UI.

Parameters

- **font_name** -- The name of the font that will be used to reference it elsewhere in the GUI.
- **regular_path** -- The path of the font file for this font with no styles applied.
- **bold_path** -- The path of the font file for this font with just bold style applied.
- **italic_path** -- The path of the font file for this font with just italic style applied.
- **bold_italic_path** -- The path of the font file for this font with bold & italic style applied.

abstract calculate_scaled_mouse_position(*position: Tuple[int, int]*) → Tuple[int, int]

Scaling an input mouse position by a scale factor.

abstract clear_and_reset()

Clear the whole UI.

abstract create_tool_tip(*text: str*, *position: Tuple[int, int]*, *hover_distance: Tuple[int, int]*, *parent_element:* IUIElementInterface, *object_id:* ObjectID, ***, *wrap_width: int | None = None*, *text_kwargs: Dict[str, str] | None = None*) → *IUITooltipInterface*

Creates a tool tip ands returns it.

Parameters

- **text** -- The tool tips text, can utilise the HTML subset used in all UITextBoxes.
- **position** -- The screen position to create the tool tip for.

- **hover_distance** -- The distance we should hover away from our target position.
- **parent_element** -- The UIElement that spawned this tool tip.
- **object_id** -- the object_id of the tooltip.
- **wrap_width** -- an optional width for the tool tip, will overwrite any value from the theme file.
- **text_kwargs** -- a dictionary of variable arguments to pass to the translated string useful when you have multiple translations that need variables inserted in the middle.

Returns
A tool tip placed somewhere on the screen.

abstract draw_ui(*window_surface: Surface*)

Draws the UI.

Parameters
window_surface -- The screen or window surface on which we are going to draw all of our UI Elements.

abstract get_double_click_time() → float

Returns time between clicks that counts as a double click.

Returns
A float, time measured in seconds.

abstract get_focus_set() → Set[*IUIElementInterface*]

Gets the focused set.

Returns
The set of elements that currently have interactive focus. If None, nothing is currently focused.

abstract get_hovering_any_element() → bool

True if any UI element (other than the root container) is hovered by the mouse.

Combined with 'get_focus_set()' and the return value from process_events(), it should make it easier to switch input events between the UI and other parts of an application.

abstract get_locale() → str

Get the locale language code being used in the UIManager

Returns
A two-letter ISO 639-1 code for the current locale.

abstract get_mouse_position() → Tuple[int, int]

Get the position of the mouse in the UI.

abstract get_root_container() → *IUIContainerInterface*

Returns the 'root' container. The one all UI elements are placed in by default if they are not placed anywhere else, fills the whole OS/pygame window.

Returns
A container.

abstract get_shadow(*size: Tuple[int, int]*, *shadow_width: int = 2*, *shape: str = 'rectangle'*, *corner_radius: List[int] | None = None*) → Surface

Returns a 'shadow' surface scaled to the requested size.

Parameters

- **size** -- The size of the object we are shadowing + it's shadow.

- **shadow_width** -- The width of the shadowed edge.
- **shape** -- The shape of the requested shadow.
- **corner_radius** -- The radius of the shadow corners if this is a rectangular shadow.

Returns
A shadow as a pygame Surface.

abstract get_sprite_group() → *LayeredGUIGroup*

Gets the sprite group used by the entire UI to keep it in the correct order for drawing and processing input.

Returns
The UI's sprite group.

abstract get_theme() → *IUIAppearanceThemeInterface*

Gets the theme so the data in it can be accessed.

Returns
The theme data used by this UIManager

abstract get_universal_empty_surface() → Surface

Sometimes we want to hide sprites or just have sprites with no visual component, when we do we can just use this empty surface to save having lots of empty surfaces all over memory.

Returns
An empty and therefore invisible pygame.surface.Surface

abstract get_window_stack() → *IUIWindowStackInterface*

The UIWindowStack organises any windows in the UI Manager so that they are correctly sorted and move windows we interact with to the top of the stack.

Returns
The stack of windows.

abstract preload_fonts(*font_list: List[Dict[str, str | int | float]]*)

Pre-loads a list of fonts.

Parameters
font_list -- A list of font descriptions in dictionary format as described above.

abstract print_layer_debug()

Print some formatted information on the current state of the UI Layers.

Handy for debugging layer problems.

abstract print_unused_fonts()

Prints a list of fonts that have been loaded but are not being used.

abstract process_events(*event: Event*)

This is the top level method through which all input to UI elements is processed and reacted to.

Parameters
event -- pygame.event.Event - the event to process.

abstract set_active_cursor(*cursor: Tuple[Tuple[int, int], Tuple[int, int], Tuple[int, ...], Tuple[int, ...]]*)

This is for users of the library to set the currently active cursor, it will be currently only be overridden by the resizing cursors.

The expected input is in the same format as the standard pygame cursor module, except without expanding the initial Tuple. So, to call this function with the default pygame arrow cursor you would do:

manager.set_active_cursor(pygame.cursors.arrow)

abstract set_focus_set(*focus:* IUIElementInterface | *Set[*IUIElementInterface*]* | *None*)

Set a set of element as the focused set.

Parameters
focus -- The set of element to focus on.

abstract set_locale(*locale: str*)

Set a locale language code to use in the UIManager

Parameters
locale -- A two letter ISO 639-1 code for a supported language.

TODO: Make this raise an exception for an unsupported language?

abstract set_text_hovered(*hovering_text_input: bool*)

Set to true when hovering an area containing selectable text.

Currently, switches the cursor to the I-Beam cursor.

Parameters
hovering_text_input -- set to True to toggle the I-Beam cursor

abstract set_visual_debug_mode(*is_active: bool*)

Loops through all our UIElements to turn visual debug mode on or off. Also calls print_layer_debug()

Parameters
is_active -- True to activate visual debug and False to turn it off.

abstract set_window_resolution(*window_resolution: Tuple[int, int]*)

Sets the window resolution.

Parameters
window_resolution -- the resolution to set.

abstract update(*time_delta: float*)

Update the UIManager.

Parameters
time_delta -- The time passed since the last call to update, in seconds.

pygame_gui.core.interfaces.window_interface module

class pygame_gui.core.interfaces.window_interface.**IWindowInterface**

Bases: object

A metaclass that defines the interface that the window stack uses to interface with the UIWindow class.

Interfaces like this help us evade cyclical import problems by allowing us to define the actual window class later on and have it make use of the window stack.

abstract property always_on_top: bool

Whether the window is always above normal windows or not. :return:

abstract can_hover() → bool

Called to test if this window can be hovered.

abstract change_layer(*layer: int*)

Change the drawing layer of this window.

Parameters
layer -- the new layer to move to.

abstract check_clicked_inside_or_blocking(*event: Event*) → bool

A quick event check outside the normal event processing so that this window is brought to the front of the window stack if we click on any of the elements contained within it.

Parameters
event -- The event to check.

Returns
returns True if the event represents a click inside this window or the window is blocking.

abstract check_hover(*time_delta: float*, *hovered_higher_element: bool*) → bool

For the window the only hovering we care about is the edges if this is a resizable window.

Parameters

- **time_delta** -- time passed in seconds between one call to this method and the next.
- **hovered_higher_element** -- Have we already hovered an element/window above this one?

abstract get_hovering_edge_id() → str

Gets the ID of the combination of edges we are hovering for use by the cursor system.

Returns
a string containing the edge combination ID (e.g. xy,yx,xl,xr,yt,yb)

get_layer_thickness() → int

The layer 'thickness' of this window/ :return: an integer

abstract get_top_layer() → int

Returns the 'highest' layer used by this window so that we can correctly place other windows on top of it.

Returns
The top layer for this window as a number (greater numbers are higher layers).

abstract kill()

Overrides the basic kill() method of a pygame sprite so that we also kill all the UI elements in this window, and remove if from the window stack.

abstract property layer: int

The layer of this window (read-only)

abstract on_moved_to_front()

Called when a window is moved to the front of the stack.

abstract process_event(*event: Event*) → bool

Handles resizing & closing windows. Gives UI Windows access to pygame events. Derived windows should super() call this class if they implement their own process_event method.

Parameters
event -- The event to process.

Return bool
Return True if this element should consume this event and not pass it to the rest of the UI.

abstract rebuild()

Rebuilds the window when the theme has changed.

abstract rebuild_from_changed_theme_data()

Called by the UIManager to check the theming data and rebuild whatever needs rebuilding for this element when the theme data has changed.

abstract set_blocking(*state: bool*)

Sets whether this window being open should block clicks to the rest of the UI or not. Defaults to False.

Parameters
state -- True if this window should block mouse clicks.

abstract set_dimensions(*dimensions: Vector2 | Tuple[float, float]*)

Set the size of this window and then re-sizes and shifts the contents of the windows container to fit the new size.

Parameters
dimensions -- The new dimensions to set.

abstract set_display_title(*new_title: str*)

Set the title of the window.

Parameters
new_title -- The title to set.

abstract set_minimum_dimensions(*dimensions: Vector2 | Tuple[float, float]*)

If this window is resizable, then the dimensions we set here will be the minimum that users can change the window to. They are also used as the minimum size when 'set_dimensions' is called.

Parameters
dimensions -- The new minimum dimension for the window.

abstract set_position(*position: Vector2 | Tuple[float, float]*)

Method to directly set the absolute screen rect position of an element.

Parameters
position -- The new position to set.

abstract set_relative_position(*position: Vector2 | Tuple[float, float]*)

Method to directly set the relative rect position of an element.

Parameters
position -- The new position to set.

abstract should_use_window_edge_resize_cursor() → bool

Returns true if this window is in a state where we should display one of the resizing cursors

Returns
True if a resizing cursor is needed.

abstract update(*time_delta: float*)

A method called every update cycle of our application. Designed to be overridden by derived classes but also has a little functionality to make sure the window's layer 'thickness' is accurate and to handle window resizing.

Parameters
time_delta -- time passed in seconds between one call to this method and the next.

Module contents

class pygame_gui.core.interfaces.**IColourGradientInterface**

Bases: object

A meta class that defines the interface that a colour gradient uses.

Interfaces like this help us evade cyclical import problems by allowing us to define the actual manager class later on and have it make use of the classes that use the interface.

abstract apply_gradient_to_surface(*input_surface: Surface*, *rect: Rect | None = None*)

Applies this gradient to a specified input surface using blending multiplication. As a result this method works best when the input surface is a mostly white, stencil shape type surface.

Parameters

- **input_surface** --
- **rect** -- The rectangle on the surface to apply the gradient to. If None, applies to the whole surface.

class pygame_gui.core.interfaces.**IContainerLikeInterface**

Bases: object

A metaclass that defines the interface for containers used by elements.

This interface lets us treat classes like UIWindows and UIPanels like containers for elements even though they actually pass this functionality off to the proper UIContainer class.

are_contents_hovered() → bool

Are any of the elements in the container hovered? Used for handling mousewheel events.

Returns
True if one of the elements is hovered, False otherwise.

abstract get_container() → *IUIContainerInterface*

Gets an actual container from this container-like UI element.

abstract hide()

Hides the container, which means the container will not get drawn and will not process events. Should also hide all the children elements. If the container was hidden before - ignore.

abstract show()

Shows the container, which means the container will get drawn and will process events. Should also show all the children elements. If the container was visible before - ignore.

class pygame_gui.core.interfaces.**IGUIFontInterface**

Bases: object

A font interface, so we can easily switch between pygame.freetype.Font and pygame.Font.

get_direction() → int

Returns

abstract get_metrics(*text: str*)

Parameters
text --

Returns

abstract get_padding_height()

Returns

abstract get_point_size()

abstract get_rect(*text: str*) → Rect

Not sure if we want this. :return:

abstract render_premul(*text: str*, *text_color: Color*) → Surface

Draws text to a surface ready for pre-multiplied alpha-blending

render_premul_to(*text: str*, *text_colour: Color*, *surf_size: Tuple[int, int]*, *surf_position: Tuple[int, int]*)

Parameters

- **text** --
- **text_colour** --
- **surf_size** --
- **surf_position** --

Returns

size(*text: str*) → Tuple[int, int]

Return the pixel size of a given text string in this font

Parameters
text -- the text to check.

Returns
the width & height in pixels.

abstract property underline: bool

Returns

abstract property underline_adjustment: float

Returns

class pygame_gui.core.interfaces.**IGUISpriteInterface**

Bases: object

A sprite Interface class specifically designed for the GUI. Very similar to pygame's DirtySprite but without the Dirty flag.

abstract add(**groups*)

add the sprite to groups

Parameters
groups -- sprite groups to add this sprite to.

Any number of Group instances can be passed as arguments. The Sprite will be added to the Groups it is not already a member of.

abstract add_internal(*group*)

For adding this sprite to a group internally.

Parameters
group -- The group we are adding to.

abstract alive()

does the sprite belong to any groups

Sprite.alive(): return bool

Returns True when the Sprite belongs to one or more Groups.

abstract property blendmode

Layer property can only be set before the sprite is added to a group, after that it is read only and a sprite's layer in a group should be set via the group's change_layer() method.

Overwrites dynamic property from sprite class for speed.

abstract groups()

list of Groups that contain this Sprite

Sprite.groups(): return group_list

Returns a list of all the Groups that contain this Sprite.

abstract property image

Layer property can only be set before the sprite is added to a group, after that it is read only and a sprite's layer in a group should be set via the group's change_layer() method.

Overwrites dynamic property from sprite class for speed.

abstract kill()

remove the Sprite from all Groups

Sprite.kill(): return None

The Sprite is removed from all the Groups that contain it. This won't change anything about the state of the Sprite. It is possible to continue to use the Sprite after this method has been called, including adding it to Groups.

abstract property layer

Layer property can only be set before the sprite is added to a group, after that it is read only and a sprite's layer in a group should be set via the group's change_layer() method.

Overwrites dynamic property from sprite class for speed.

abstract property rect

Layer property can only be set before the sprite is added to a group, after that it is read only and a sprite's layer in a group should be set via the group's change_layer() method.

Overwrites dynamic property from sprite class for speed.

abstract remove(**groups*)

remove the sprite from groups

Parameters
groups -- sprite groups to remove this sprite from.

Any number of Group instances can be passed as arguments. The Sprite will be removed from the Groups it is currently a member of.

abstract remove_internal(*group*)

For removing this sprite from a group internally.

Parameters
group -- The group we are removing from.

abstract update(*time_delta: float*)

A stub to override.

Parameters
time_delta -- the time passed in seconds between calls to this function.

abstract property visible

You can make this sprite disappear without removing it from the group assign 0 for invisible and 1 for visible

class pygame_gui.core.interfaces.**IUIAppearanceThemeInterface**

Bases: object

A meta class that defines the interface that a UI Appearance Theme uses.

Interfaces like this help us evade cyclical import problems by allowing us to define the actual manager class later on and have it make use of the classes that use the interface.

abstract build_all_combined_ids(*element_base_ids: None | List[str | None]*, *element_ids: None | List[str]*, *class_ids: None | List[str | None]*, *object_ids: None | List[str | None]*) → List[str]

Construct a list of combined element ids from the element's various accumulated ids.

Parameters

- **element_base_ids** -- when an element is also another element (e.g. a file dialog is also a window)
- **element_ids** -- All the ids of elements this element is contained within.
- **class_ids** -- All the ids of 'classes' that this element is contained within.
- **object_ids** -- All the ids of objects this element is contained within.

Returns
A list of IDs that reference this element in order of decreasing specificity.

abstract check_need_to_reload() → bool

Check if we need to reload our theme file because it's been modified. If so, trigger a reload and return True so that the UIManager can trigger elements to rebuild from the theme data.

Return bool
True if we need to reload elements because the theme data has changed.

abstract get_colour(*colour_id: str*, *combined_element_ids: List[str] | None = None*) → Color

Uses data about a UI element and a specific ID to find a colour from our theme.

Parameters

- **combined_element_ids** -- A list of IDs representing an element's location in a hierarchy of elements.
- **colour_id** -- The id for the specific colour we are looking for.

Return pygame.Color
A pygame colour.

abstract get_colour_or_gradient(*colour_id: str*, *combined_ids: List[str] | None = None*) → Color | *IColourGradientInterface*

Uses data about a UI element and a specific ID to find a colour, or a gradient, from our theme. Use this function if the UIElement can handle either type.

Parameters

- **combined_ids** -- A list of IDs representing an element's location in a hierarchy of elements.
- **colour_id** -- The id for the specific colour we are looking for.

Return pygame.Color or ColourGradient
A colour or a gradient object.

abstract get_font(*combined_element_ids: List[str]*) → *IGUIFontInterface*

Uses some data about a UIElement to get a font object.

Parameters
combined_element_ids -- A list of IDs representing an element's location in a interleaved hierarchy of elements.

Return IGUIFontInterface
An interface to a pygame font object wrapper.

abstract get_font_dictionary() → *IUIFontDictionaryInterface*

Lets us grab the font dictionary, which is created by the theme object, so we can access it directly.

Return UIFontDictionary
The font dictionary.

abstract get_font_info(*combined_element_ids: List[str]*) → Dict[str, Any]

Uses some data about a UIElement to get font data as dictionary

Parameters
combined_element_ids -- A list of IDs representing an element's location in a interleaved hierarchy of elements.

Return dictionary
Data about the font requested

abstract get_image(*image_id: str*, *combined_element_ids: List[str]*) → Surface

Will raise an exception if no image with the ids specified is found. UI elements that have an optional image display will need to handle the exception.

Parameters

- **combined_element_ids** -- A list of IDs representing an element's location in a hierarchy of elements.
- **image_id** -- The id identifying the particular image spot in the UI we are looking for an image to add to.

Returns
A pygame.surface.Surface

abstract get_misc_data(*misc_data_id: str*, *combined_element_ids: List[str]*) → str | Dict

Uses data about a UI element and a specific ID to try and find a piece of miscellaneous theming data. Raises an exception if it can't find the data requested, UI elements requesting optional data will need to handle this exception.

Parameters

- **combined_element_ids** -- A list of IDs representing an element's location in a interleaved hierarchy of elements.
- **misc_data_id** -- The id for the specific piece of miscellaneous data we are looking for.

Return Any
Returns a string or a Dict

abstract load_theme(*file_path: str | PathLike | StringIO | PackageResource | dict*)

Loads a theme, and currently, all associated data like fonts and images required by the theme.

Parameters

file_path -- The location of the theme, or the theme data we want to load.

abstract reload_theming()

We need to load our theme file to see if anything expensive has changed, if so trigger it to reload/rebuild.

abstract update_caching(*time_delta: float*)

Updates the various surface caches.

abstract update_single_element_theming(*element_name: str*, *new_theming_data: str*)

Update theming data, via string - for a single element. :param element_name: :param new_theming_data: :return:

abstract update_theming(*new_theming_data: str*, *rebuild_all: bool = True*)

Update theming data, via string - for the whole UI.

Parameters

- **new_theming_data** --
- **rebuild_all** --

Returns

class pygame_gui.core.interfaces.**IUIContainerInterface**

Bases: *IUIElementInterface*

Interface for the actual container class. Not to be confused with the IContainerLikeInterface which is an interface for all the things we can treat like containers when creating elements.

abstract add_element(*element:* IUIElementInterface)

Add a UIElement to the container. The UI's relative_rect parameter will be relative to this container.

Parameters

element -- A UIElement to add to this container.

calc_add_element_changes_thickness(*element:* IUIElementInterface)

This function checks if a single added element will increase the containers thickness and if so updates containers recursively.

Parameters

element -- the element to check.

abstract change_layer(*new_layer: int*)

Change the layer of this container. Layers are used by the GUI to control the order in which things are drawn and which things should currently be interactive (so you can't interact with things behind other things).

This particular method is most often used to shift the visible contents of a window in front of any others when it is moved to the front of the window stack.

Parameters

new_layer -- The layer to move our container to.

abstract check_hover(*time_delta: float*, *hovered_higher_element: bool*) → bool

A method that helps us to determine which, if any, UI Element is currently being hovered by the mouse.

Parameters

- **time_delta** -- A float, the time in seconds between the last call to this function and now (roughly).
- **hovered_higher_element** -- A boolean, representing whether we have already hovered a 'higher' element.

Returns
A boolean that is true if we have hovered a UI element, either just now or before this method.

abstract clear()

Removes and kills all the UI elements inside this container.

abstract get_image_clipping_rect() → Rect | None

Obtain the current image clipping rect.

Returns
The current clipping rect. It may be None.

abstract get_rect() → Rect

Access to the container's rect

Returns
a pygame rectangle

abstract get_size() → Tuple[int, int]

Get the container's pixel size.

Returns
the pixel size as tuple [x, y]

abstract get_thickness() → int

Get the container's layer thickness.

Returns
the thickness as an integer.

abstract get_top_layer() → int

Assuming we have correctly calculated the 'thickness' of this container, this method will return the 'highest' layer in the LayeredDirty UI Group.

Returns
An integer representing the current highest layer being used by this container.

abstract kill()

Overrides the standard kill method of UI Elements (and pygame sprites beyond that) to also call the kill method on all contained UI Elements.

on_contained_elements_changed(*target:* IUIElementInterface) → None

Update the contents of this container that one of their layout anchors may have moved, or been resized.

Parameters
target -- the UI element that has been benn moved or resized.

abstract recalculate_container_layer_thickness()

This function will iterate through the elements in our container and determine the maximum 'height' that they reach in the 'layer stack'. We then use that to determine the overall 'thickness' of this container. The thickness value is used to determine where to place overlapping windows in the layers

abstract remove_element(*element:* IUIElementInterface)

Remove a UIElement from this container.

Parameters
element -- A UIElement to remove from this container.

abstract set_dimensions(*dimensions: Vector2 | Tuple[float, float]*, *clamp_to_container: bool = False*)

Set the dimension of this container and update the positions of elements within it accordingly.

Parameters

- **clamp_to_container** --
- **dimensions** -- the new dimensions.

abstract set_position(*position: Vector2 | Tuple[float, float]*)

Set the absolute position of this container - it is usually less chaotic to deal with setting relative positions.

Parameters
position -- the new absolute position to set.

abstract set_relative_position(*position: Vector2 | Tuple[float, float]*)

Set the position of this container, relative to the container it is within.

Parameters
position -- the new relative position to set.

abstract update_containing_rect_position()

This function is called when we move the container to update all the contained UI Elements to move as well.

class pygame_gui.core.interfaces.**IUIElementInterface**

Bases: *IGUISpriteInterface*

Interface for the ui element class. This is so we can refer to ui elements in other classes before the UIElement has itself been defined.

abstract can_hover() → bool

A stub method to override. Called to test if this method can be hovered.

abstract change_layer(*new_layer: int*)

Changes the layer this element is on.

Parameters
new_layer -- The layer to change this element to.

abstract check_hover(*time_delta: float*, *hovered_higher_element: bool*) → bool

A method that helps us to determine which, if any, UI Element is currently being hovered by the mouse.

Parameters

- **time_delta** -- A float, the time in seconds between the last call to this function and now (roughly).
- **hovered_higher_element** -- A boolean, representing whether we have already hovered a 'higher' element.

Return bool
A boolean that is true if we have hovered a UI element, either just now or before this method.

abstract disable()

Disables elements so they are no longer interactive.

Elements should handle their own enabling and disabling.

abstract enable()

Enables elements so they are interactive again.

Elements should handle their own enabling and disabling.

abstract focus()

A stub to override. Called when we focus this UI element.

abstract get_abs_rect() → Rect

The absolute positioning rect.

Returns
A pygame rect.

abstract get_anchor_targets() → list

Get any anchor targets this element has, so we can update them when their targets change :return: the list of anchor targets.

abstract get_anchors() → Dict[str, str | *IUIElementInterface*]

A dictionary containing all the anchors defining what the relative rect is relative to

Returns
A dictionary containing all the anchors defining what the relative rect is relative to

abstract get_class_ids() → List[str]

A list of all the class IDs in this element's theming/event hierarchy.

Returns
a list of strings, one for each element in the hierarchy.

abstract get_element_base_ids() → List[str]

A list of all the element base IDs in this element's theming/event hierarchy.

Returns
a list of strings, one for each element in the hierarchy.

abstract get_element_ids() → List[str]

A list of all the element IDs in this element's theming/event hierarchy.

Returns
a list of strings, one for each element in the hierarchy.

abstract get_focus_set() → Set[Any]

Return the set of elements to focus when we focus this element.

abstract get_image_clipping_rect() → Rect | None

Obtain the current image clipping rect.

Returns
The current clipping rect. Maybe None.

abstract get_object_ids() → List[str]

A list of all the object IDs in this element's theming/event hierarchy.

Returns
a list of strings, one for each element in the hierarchy.

abstract get_relative_rect() → Rect

The relative positioning rect.

Returns
A pygame rect.

abstract get_starting_height() → int

Get the starting layer height of this element. (i.e. the layer we start placing it on *above* it's container, it may use more layers above this layer)

Returns
an integer representing the starting layer height.

abstract get_top_layer() → int

Assuming we have correctly calculated the 'thickness' of it, this method will return the top of this element.

Return int
An integer representing the current highest layer being used by this element.

abstract hide()

Hides the widget, which means the widget will not get drawn and will not process events. Clear hovered state.

abstract hover_point(*hover_x: float*, *hover_y: float*) → bool

Test if a given point counts as 'hovering' this UI element. Normally that is a straightforward matter of seeing if a point is inside the rectangle. Occasionally it will also check if we are in a wider zone around a UI element once it is already active, this makes it easier to move scroll bars and the like.

Parameters

- **hover_x** -- The x (horizontal) position of the point.
- **hover_y** -- The y (vertical) position of the point.

Returns
Returns True if we are hovering this element.

abstract property hovered: bool

Are we hovering over this element with the mouse pointer or other input highlighting method.

Returns
True if hovered.

abstract join_focus_sets(*element*)

Join this element's focus set with another element's focus set.

Parameters
element -- The other element whose focus set we are joining with.

abstract kill()

Overriding regular sprite kill() method to remove the element from its container.

abstract on_fresh_drawable_shape_ready()

Called when our drawable shape has finished rebuilding the active surface. This is needed because sometimes we defer rebuilding until a more advantageous (read quieter) moment.

abstract on_hovered()

A stub to override. Called when this UI element first enters the 'hovered' state.

abstract on_locale_changed()

Called for each element when the locale is changed on their UIManager

abstract on_unhovered()

A stub to override. Called when this UI element leaves the 'hovered' state.

abstract process_event(*event: Event*) → bool

A stub to override. Gives UI Elements access to pygame events.

Parameters
event -- The event to process.

Returns
Should return True if this element makes use of this event.

abstract rebuild()

Takes care of rebuilding this element. Most derived elements are going to override this, and hopefully call the super() class method.

abstract rebuild_from_changed_theme_data()

A stub to override. Used to test if the theming data for this element has changed and rebuild the element if so.

abstract remove_element_from_focus_set(*element*)

remove an element from this sets focus group.

Parameters
element -- The element to remove.

abstract set_anchors(*anchors: Dict[str, str | IUIElementInterface] | None*) → None

Wraps the setting of the anchors with some validation

Parameters
anchors -- A dictionary of anchors defining what the relative rect is relative to

Returns
None

abstract set_dimensions(*dimensions: Vector2 | Tuple[float, float]*, *clamp_to_container: bool = False*)

Method to directly set the dimensions of an element.

NOTE: Using this on elements inside containers with non-default anchoring arrangements may make a mess of them.

Parameters

- **dimensions** -- The new dimensions to set.
- **clamp_to_container** -- Whether we should clamp the dimensions to the dimensions of the container or not.

abstract set_focus_set(*focus_set: Set[Any]*)

Set the focus set to a specific set of elements.

Parameters
focus_set -- The focus set to set.

abstract set_image(*new_image: Surface | None*)

Deprecated for most elements - to avoid confusion with setting the image for the UIImage element.

Generally the average user shouldn't be directly setting what this was setting.

Parameters
new_image -- The new image to set.

abstract set_position(*position: Vector2 | Tuple[float, float]*)

Method to directly set the absolute screen rect position of an element.

Parameters
position -- The new position to set.

abstract set_relative_position(*position: Vector2 | Tuple[float, float]*)

Method to directly set the relative rect position of an element.

Parameters
position -- The new position to set.

abstract set_visual_debug_mode(*activate_mode: bool*)

Enables a debug mode for the element which displays layer information on top of it in a tiny font.

Parameters
activate_mode -- True or False to enable or disable the mode.

abstract show()

Shows the widget, which means the widget will get drawn and will process events.

abstract unfocus()

A stub to override. Called when we stop focusing this UI element.

abstract update(*time_delta: float*)

Updates this element's drawable shape, if it has one.

Parameters
time_delta -- The time passed between frames, measured in seconds.

abstract update_containing_rect_position()

Updates the position of this element based on the position of its container. Usually called when the container has moved.

abstract while_hovering(*time_delta: float*, *mouse_pos: Vector2*)

A stub method to override. Called when this UI element is currently hovered.

Parameters

- **time_delta** -- A float, the time in seconds between the last call to this function and now (roughly).
- **mouse_pos** -- The current position of the mouse as 2D Vector.

class pygame_gui.core.interfaces.**IUIFontDictionaryInterface**

Bases: object

A metaclass that defines the interface that a font dictionary uses.

Interfaces like this help us evade cyclical import problems by allowing us to define the actual manager class later on and have it make use of the classes that use the interface.

abstract add_font_path(*font_name: str*, *font_path: str*, *bold_path: str | None = None*, *italic_path: str | None = None*, *bold_italic_path: str | None = None*)

Adds paths to different font files for a font name.

Parameters

- **font_name** -- The name to assign to these font files.

- **font_path** -- The path to the font's file with no particular style.
- **bold_path** -- The path to the font's file with a bold style.
- **italic_path** -- The path to the font's file with an italic style.
- **bold_italic_path** -- The path to the font's file with a bold and an italic style.

abstract check_font_preloaded(*font_id: str*) → bool

Check if a font is already preloaded or not.

Parameters
font_id -- The ID of the font to check for

Returns
True or False.

abstract convert_html_to_point_size(*html_size: float*) → int

Takes in an HTML style font size and converts it into a point font size.

Parameters
html_size -- Size in HTML style.

Return int
A 'point' font size.

abstract create_font_id(*font_size: int*, *font_name: str*, *bold: bool*, *italic: bool*, *antialiased: bool = True*) → str

Create an id for a particularly styled and sized font from those characteristics.

Parameters

- **font_size** -- The size of the font.
- **font_name** -- The name of the font.
- **bold** -- Whether the font is bold styled or not.
- **italic** -- Whether the font is italic styled or not.
- **antialiased** -- Whether the font is antialiased or not.

Return str
The finished font id.

abstract ensure_debug_font_loaded()

Ensure the font we use for debugging purposes is loaded. Generally called after we start a debugging mode.

abstract find_font(*font_size: int*, *font_name: str*, *bold: bool = False*, *italic: bool = False*, *antialiased: bool = True*, *script: str = 'Latn'*, *direction: int = 0*) → *IGUIFontInterface*

Find a loaded font from the font dictionary. Will load a font if it does not already exist, and we have paths to the needed files, however it will issue a warning after doing so because dynamic file loading is normally a bad idea as you will get frame rate hitches while the running program waits for the font to load.

Instead, it's best to preload all your needed files at another time in your program when you have more control over the user experience.

Parameters

- **font_size** -- The size of the font to find.
- **font_name** -- The name of the font to find.
- **bold** -- Whether the font is bold or not.

- **italic** -- Whether the font is italic or not.
- **antialiased** -- Whether the font is antialiased or not.
- **script** -- The ISO 15924 script code used for text shaping as a string.
- **direction** -- the direction of text e.g. left to right or right to left. An integer.

Return IGUIFontInterface
Returns either the font we asked for, or the default font.

abstract get_default_font() → *IGUIFontInterface*

Grab the default font.

Returns
The default font.

abstract preload_font(*font_size: int*, *font_name: str*, *bold: bool = False*, *italic: bool = False*, *force_immediate_load: bool = False*, *antialiased: bool = True*, *script: str = 'Latn'*, *direction: int = 0*)

Lets us load a font at a particular size and style before we use it. While you can get away with relying on dynamic font loading during development, it is better to eventually preload all your font data at a controlled time, which is where this method comes in.

Parameters

- **font_size** -- The size of the font to load.
- **font_name** -- The name of the font to load.
- **bold** -- Whether the font is bold styled or not.
- **italic** -- Whether the font is italic styled or not.
- **force_immediate_load** -- bypasses any asynchronous threaded loading setup to immediately load the font on the main thread.
- **antialiased** -- Whether the font is antialiased or not.
- **script** -- The ISO 15924 script code used for text shaping as a string.
- **direction** -- the direction of text e.g. left to right or right to left. An integer.

abstract print_unused_loaded_fonts()

Can be called to check if the UI is loading any fonts that we haven't used by the point this function is called. If a font is truly unused then we can remove it from our loading and potentially speed up the overall loading of the program.

This is not a foolproof check because this function could easily be called before we have explored all the code paths in a project that may use fonts.

set_locale(*new_locale: str*)

This may change the default font.

Parameters
new_locale -- The new locale to set, a two-letter country code ISO 639-1

class pygame_gui.core.interfaces.**IUIManagerInterface**

Bases: object

A metaclass that defines the interface that a UI Manager uses.

Interfaces like this help us evade cyclical import problems by allowing us to define the actual manager class later on and have it make use of the classes that use the interface.

abstract add_font_paths(*font_name: str*, *regular_path: str*, *bold_path: str | None = None*, *italic_path: str | None = None*, *bold_italic_path: str | None = None*)

Add file paths for custom fonts you want to use in the UI.

Parameters

- **font_name** -- The name of the font that will be used to reference it elsewhere in the GUI.
- **regular_path** -- The path of the font file for this font with no styles applied.
- **bold_path** -- The path of the font file for this font with just bold style applied.
- **italic_path** -- The path of the font file for this font with just italic style applied.
- **bold_italic_path** -- The path of the font file for this font with bold & italic style applied.

abstract calculate_scaled_mouse_position(*position: Tuple[int, int]*) → Tuple[int, int]

Scaling an input mouse position by a scale factor.

abstract clear_and_reset()

Clear the whole UI.

abstract create_tool_tip(*text: str*, *position: Tuple[int, int]*, *hover_distance: Tuple[int, int]*, *parent_element:* IUIElementInterface, *object_id:* ObjectID, ***, *wrap_width: int | None = None*, *text_kwargs: Dict[str, str] | None = None*) → *IUITooltipInterface*

Creates a tool tip ands returns it.

Parameters

- **text** -- The tool tips text, can utilise the HTML subset used in all UITextBoxes.
- **position** -- The screen position to create the tool tip for.
- **hover_distance** -- The distance we should hover away from our target position.
- **parent_element** -- The UIElement that spawned this tool tip.
- **object_id** -- the object_id of the tooltip.
- **wrap_width** -- an optional width for the tool tip, will overwrite any value from the theme file.
- **text_kwargs** -- a dictionary of variable arguments to pass to the translated string useful when you have multiple translations that need variables inserted in the middle.

Returns

A tool tip placed somewhere on the screen.

abstract draw_ui(*window_surface: Surface*)

Draws the UI.

Parameters

window_surface -- The screen or window surface on which we are going to draw all of our UI Elements.

abstract get_double_click_time() → float

Returns time between clicks that counts as a double click.

Returns

A float, time measured in seconds.

abstract get_focus_set() → Set[*IUIElementInterface*]

Gets the focused set.

Returns
The set of elements that currently have interactive focus. If None, nothing is currently focused.

abstract get_hovering_any_element() → bool

True if any UI element (other than the root container) is hovered by the mouse.

Combined with 'get_focus_set()' and the return value from process_events(), it should make it easier to switch input events between the UI and other parts of an application.

abstract get_locale() → str

Get the locale language code being used in the UIManager

Returns
A two-letter ISO 639-1 code for the current locale.

abstract get_mouse_position() → Tuple[int, int]

Get the position of the mouse in the UI.

abstract get_root_container() → *IUIContainerInterface*

Returns the 'root' container. The one all UI elements are placed in by default if they are not placed anywhere else, fills the whole OS/pygame window.

Returns
A container.

abstract get_shadow(*size: Tuple[int, int]*, *shadow_width: int = 2*, *shape: str = 'rectangle'*, *corner_radius: List[int] | None = None*) → Surface

Returns a 'shadow' surface scaled to the requested size.

Parameters

- **size** -- The size of the object we are shadowing + it's shadow.
- **shadow_width** -- The width of the shadowed edge.
- **shape** -- The shape of the requested shadow.
- **corner_radius** -- The radius of the shadow corners if this is a rectangular shadow.

Returns
A shadow as a pygame Surface.

abstract get_sprite_group() → *LayeredGUIGroup*

Gets the sprite group used by the entire UI to keep it in the correct order for drawing and processing input.

Returns
The UI's sprite group.

abstract get_theme() → *IUIAppearanceThemeInterface*

Gets the theme so the data in it can be accessed.

Returns
The theme data used by this UIManager

abstract get_universal_empty_surface() → Surface

Sometimes we want to hide sprites or just have sprites with no visual component, when we do we can just use this empty surface to save having lots of empty surfaces all over memory.

Returns
An empty and therefore invisible pygame.surface.Surface

abstract get_window_stack() → *IUIWindowStackInterface*

The UIWindowStack organises any windows in the UI Manager so that they are correctly sorted and move windows we interact with to the top of the stack.

Returns
The stack of windows.

abstract preload_fonts(*font_list: List[Dict[str, str | int | float]]*)

Pre-loads a list of fonts.

Parameters
font_list -- A list of font descriptions in dictionary format as described above.

abstract print_layer_debug()

Print some formatted information on the current state of the UI Layers.

Handy for debugging layer problems.

abstract print_unused_fonts()

Prints a list of fonts that have been loaded but are not being used.

abstract process_events(*event: Event*)

This is the top level method through which all input to UI elements is processed and reacted to.

Parameters
event -- pygame.event.Event - the event to process.

abstract set_active_cursor(*cursor: Tuple[Tuple[int, int], Tuple[int, int], Tuple[int, ...], Tuple[int, ...]]*)

This is for users of the library to set the currently active cursor, it will be currently only be overridden by the resizing cursors.

The expected input is in the same format as the standard pygame cursor module, except without expanding the initial Tuple. So, to call this function with the default pygame arrow cursor you would do:

manager.set_active_cursor(pygame.cursors.arrow)

abstract set_focus_set(*focus:* IUIElementInterface | *Set[*IUIElementInterface*]* | *None*)

Set a set of element as the focused set.

Parameters
focus -- The set of element to focus on.

abstract set_locale(*locale: str*)

Set a locale language code to use in the UIManager

Parameters
locale -- A two letter ISO 639-1 code for a supported language.

TODO: Make this raise an exception for an unsupported language?

abstract set_text_hovered(*hovering_text_input: bool*)

Set to true when hovering an area containing selectable text.

Currently, switches the cursor to the I-Beam cursor.

Parameters
hovering_text_input -- set to True to toggle the I-Beam cursor

abstract set_visual_debug_mode(*is_active: bool*)

Loops through all our UIElements to turn visual debug mode on or off. Also calls print_layer_debug()

Parameters
is_active -- True to activate visual debug and False to turn it off.

abstract set_window_resolution(*window_resolution: Tuple[int, int]*)

Sets the window resolution.

Parameters
window_resolution -- the resolution to set.

abstract update(*time_delta: float*)

Update the UIManager.

Parameters
time_delta -- The time passed since the last call to update, in seconds.

class pygame_gui.core.interfaces.**IUITextOwnerInterface**

Bases: object

A common interface for UIElements that own some text, to help make it easier to run text effects across multiple UIElements.

abstract clear_all_active_effects(*sub_chunk:* pygame_gui.core.text.text_line_chunk.TextLineChunkFTFont | *None* = *None*)

Clears any active effects and redraws the text. A full reset, usually called before firing off a new effect if one is already in progress.

Parameters
sub_chunk -- An optional chunk so we only clear the effect from this chunk.

abstract clear_text_surface(*sub_chunk:* pygame_gui.core.text.text_line_chunk.TextLineChunkFTFont | *None* = *None*)

Clear the text surface

Parameters
sub_chunk -- An optional chunk so we only clear the surface for this chunk.

abstract get_object_id() → str

The UI object ID of this text owner for use in effect events.

Returns
the ID string

abstract get_text_letter_count(*sub_chunk:* pygame_gui.core.text.text_line_chunk.TextLineChunkFTFont | *None* = *None*) → int

The amount of letters in the text

Parameters
sub_chunk -- An optional chunk to restrict the count to only this chunk.

Returns
number of letters as an int

abstract set_active_effect(*effect_type:* UITextEffectType | *None*, *params: Dict[str, Any]* | *None* = *None*, *effect_tag: str* | *None* = *None*)

Set an animation effect to run on the text box. The effect will start running immediately after this call.

These effects are currently supported:

- TEXT_EFFECT_TYPING_APPEAR - Will look as if the text is being typed in.
- TEXT_EFFECT_FADE_IN - The text will fade in from the background colour.
- TEXT_EFFECT_FADE_OUT - The text will fade out to the background colour.

Parameters

- **effect_tag** -- if not None, only apply the effect to chunks with this tag.
- **params** -- Any parameters for the effect you are setting, if none are set defaults will be used.
- **effect_type** -- The type of the effect to set. If set to None instead it will cancel any active effect.

abstract set_text_alpha(*alpha: int*, *sub_chunk:* pygame_gui.core.text.text_line_chunk.TextLineChunkFTFont | *None* = *None*)

Set the global alpha value for the text

Parameters

- **alpha** -- the alpha to set.
- **sub_chunk** -- An optional chunk so we only set the alpha for this chunk.

abstract set_text_offset_pos(*offset: Tuple[int, int]*, *sub_chunk:* pygame_gui.core.text.text_line_chunk.TextLineChunkFTFont | *None* = *None*)

Move the text around by this offset.

Parameters

- **offset** -- the offset to set
- **sub_chunk** -- An optional chunk so we only set the offset for this chunk.

Returns

abstract set_text_rotation(*rotation: int*, *sub_chunk:* pygame_gui.core.text.text_line_chunk.TextLineChunkFTFont | *None* = *None*)

rotate the text by this int in degrees

Parameters

- **rotation** -- the rotation to set
- **sub_chunk** -- An optional chunk so we only set the rotation for this chunk.

Returns

abstract set_text_scale(*scale: float*, *sub_chunk:* pygame_gui.core.text.text_line_chunk.TextLineChunkFTFont | *None* = *None*)

Scale the text by this float

Parameters

- **scale** -- the scale to set

- **sub_chunk** -- An optional chunk so we only set the rotation for this chunk.

Returns

abstract stop_finished_effect(*sub_chunk:* pygame_gui.core.text.text_line_chunk.TextLineChunkFTFont | *None* = *None*)

Stops a finished effect. Will leave effected text in the state it was in when effect ended. Used when an effect reaches a natural end where we might want to keep it in the end of effect state (e.g. a fade out)

Parameters

sub_chunk -- An optional chunk so we only clear the effect from this chunk.

abstract update_text_effect(*time_delta: float*)

Update any active text effect on the text owner

Parameters

time_delta -- the time delta in seconds

abstract update_text_end_position(*end_pos: int*, *sub_chunk:* pygame_gui.core.text.text_line_chunk.TextLineChunkFTFont | *None* = *None*)

The position in the text to render up to.

Parameters

- **end_pos** -- The current end position as an int
- **sub_chunk** -- An optional chunk to restrict the end_position to only this chunk.

class pygame_gui.core.interfaces.**IUITooltipInterface**

Bases: object

A metaclass that defines the interface that a UI Tool tip uses.

Interfaces like this help us evade cyclical import problems by allowing us to define the actual manager class later on and have it make use of the classes that use the interface.

abstract find_valid_position(*position: Vector2 | Tuple[float, float]*) → bool

Finds a valid position for the tool tip inside the root container of the UI.

The algorithm starts from the position of the target we are providing a tool tip for then it tries to fit the rectangle for the tool tip onto the screen by moving it above, below, to the left and to the right, until we find a position that fits the whole tooltip rectangle on the screen at once.

If we fail to manage this then the method will return False. Otherwise, it returns True and set the position of the tool tip to our valid position.

Parameters

position -- A 2D vector representing the position of the target this tool tip is for.

Returns

returns True if we find a valid (visible) position and False if we do not.

abstract kill()

Overrides the UIElement's default kill method to also kill the text block element that helps make up the complete tool tip.

abstract rebuild()

Rebuild anything that might need rebuilding.

abstract rebuild_from_changed_theme_data()

Called by the UIManager to check the theming data and rebuild whatever needs rebuilding for this element when the theme data has changed.

abstract set_dimensions(*dimensions: Vector2 | Tuple[float, float]*)

Directly sets the dimensions of this tool tip. This will overwrite the normal theming.

Parameters
dimensions -- The new dimensions to set

abstract set_position(*position: Vector2 | Tuple[float, float]*)

Sets the absolute screen position of this tool tip, updating its subordinate text box at the same time.

Parameters
position -- The absolute screen position to set.

abstract set_relative_position(*position: Vector2 | Tuple[float, float]*)

Sets the relative screen position of this tool tip, updating its subordinate text box at the same time.

Parameters
position -- The relative screen position to set.

class pygame_gui.core.interfaces.**IUIWindowStackInterface**

Bases: object

A class for managing a stack of GUI windows so that only one is 'in front' at a time and the rest are sorted based on the last time they were interacted with/created.

abstract add_new_window(*window:* IWindowInterface)

Adds a window to the top of the stack.

Parameters
window -- The window to add.

abstract clear()

Empties the whole stack removing and killing all windows.

abstract get_full_stack() → List[*IWindowInterface*]

Returns the full stack of normal and always on top windows.

Returns
a list of Windows

abstract is_window_at_top(*window:* IWindowInterface) → bool

Checks if a window is at the top of the window stack or not.

Parameters
window -- The window to check.

Returns
returns True if this window is at the top of the stack.

is_window_at_top_of_top(*window:* IWindowInterface) → bool

Checks if a window is at the top of the top window stack or not.

Parameters
window -- The window to check.

Returns
returns True if this window is at the top of the stack.

abstract move_window_to_front(*window_to_front:* IWindowInterface)

Moves the passed in window to the top of the window stack and resorts the other windows to deal with the change.

Parameters
window_to_front -- the window to move to the front.

abstract remove_window(*window_to_remove:* IWindowInterface)

Removes a window from the stack and resorts the remaining windows to adjust for it's absence.

Parameters
window_to_remove -- the window to remove.

class pygame_gui.core.interfaces.**IWindowInterface**

Bases: object

A metaclass that defines the interface that the window stack uses to interface with the UIWindow class.

Interfaces like this help us evade cyclical import problems by allowing us to define the actual window class later on and have it make use of the window stack.

abstract property always_on_top: bool

Whether the window is always above normal windows or not. :return:

abstract can_hover() → bool

Called to test if this window can be hovered.

abstract change_layer(*layer: int*)

Change the drawing layer of this window.

Parameters
layer -- the new layer to move to.

abstract check_clicked_inside_or_blocking(*event: Event*) → bool

A quick event check outside the normal event processing so that this window is brought to the front of the window stack if we click on any of the elements contained within it.

Parameters
event -- The event to check.

Returns
returns True if the event represents a click inside this window or the window is blocking.

abstract check_hover(*time_delta: float*, *hovered_higher_element: bool*) → bool

For the window the only hovering we care about is the edges if this is a resizable window.

Parameters

- **time_delta** -- time passed in seconds between one call to this method and the next.
- **hovered_higher_element** -- Have we already hovered an element/window above this one?

abstract get_hovering_edge_id() → str

Gets the ID of the combination of edges we are hovering for use by the cursor system.

Returns
a string containing the edge combination ID (e.g. xy,yx,xl,xr,yt,yb)

get_layer_thickness() → int

The layer 'thickness' of this window/ :return: an integer

abstract get_top_layer() → int

Returns the 'highest' layer used by this window so that we can correctly place other windows on top of it.

Returns
The top layer for this window as a number (greater numbers are higher layers).

abstract kill()

Overrides the basic kill() method of a pygame sprite so that we also kill all the UI elements in this window, and remove if from the window stack.

abstract property layer: int

The layer of this window (read-only)

abstract on_moved_to_front()

Called when a window is moved to the front of the stack.

abstract process_event(*event: Event*) → bool

Handles resizing & closing windows. Gives UI Windows access to pygame events. Derived windows should super() call this class if they implement their own process_event method.

Parameters
event -- The event to process.

Return bool
Return True if this element should consume this event and not pass it to the rest of the UI.

abstract rebuild()

Rebuilds the window when the theme has changed.

abstract rebuild_from_changed_theme_data()

Called by the UIManager to check the theming data and rebuild whatever needs rebuilding for this element when the theme data has changed.

abstract set_blocking(*state: bool*)

Sets whether this window being open should block clicks to the rest of the UI or not. Defaults to False.

Parameters
state -- True if this window should block mouse clicks.

abstract set_dimensions(*dimensions: Vector2 | Tuple[float, float]*)

Set the size of this window and then re-sizes and shifts the contents of the windows container to fit the new size.

Parameters
dimensions -- The new dimensions to set.

abstract set_display_title(*new_title: str*)

Set the title of the window.

Parameters
new_title -- The title to set.

abstract set_minimum_dimensions(*dimensions: Vector2 | Tuple[float, float]*)

If this window is resizable, then the dimensions we set here will be the minimum that users can change the window to. They are also used as the minimum size when 'set_dimensions' is called.

Parameters
dimensions -- The new minimum dimension for the window.

abstract set_position(*position: Vector2 | Tuple[float, float]*)

Method to directly set the absolute screen rect position of an element.

Parameters
position -- The new position to set.

abstract set_relative_position(*position: Vector2 | Tuple[float, float]*)

Method to directly set the relative rect position of an element.

Parameters
position -- The new position to set.

abstract should_use_window_edge_resize_cursor() → bool

Returns true if this window is in a state where we should display one of the resizing cursors

Returns
True if a resizing cursor is needed.

abstract update(*time_delta: float*)

A method called every update cycle of our application. Designed to be overridden by derived classes but also has a little functionality to make sure the window's layer 'thickness' is accurate and to handle window resizing.

Parameters
time_delta -- time passed in seconds between one call to this method and the next.

pygame_gui.core.text package

Submodules

pygame_gui.core.text.horiz_rule_layout_rect module

class pygame_gui.core.text.horiz_rule_layout_rect.**HorizRuleLayoutRect**(*height: int*, *colour_or_gradient: Color*, *rule_dimensions: Tuple[int, int] = (-1, 1)*, *has_shade: bool = True*, *alignment=0*)

Bases: *TextLayoutRect*

Represents a horizontal rule in the HTML style. This is normally a line across the width of the layout block area, but styling options can provide some variation on that theme.

Parameters

- **height** -- the current line height of the layout/font we are using when invoking the rule.
- **colour_or_gradient** -- the colour or gradient of the rule.
- **rule_dimensions** -- the dimensions of the rule itself, normally it is 1 pixel tall and the width of the text block layout wide.
- **has_shade** -- whether the rule has 'shading' which by default is just another alpha'd line beneath it to add some depth. Doesn't work great if the line has more height to it.
- **alignment** -- ALIGN_CENTER, ALIGN_LEFT or ALIGN_RIGHT. ALIGN_CENTER is the default.

finalise(*target_surface: Surface*, *target_area: Rect*, *row_chunk_origin: int*, *row_chunk_height: int*, *row_bg_height: int*, *x_scroll_offset: int = 0*, *letter_end: int | None = None*)

Bake the contents of this layout rect onto a surface.

Parameters

- **target_surface** --
- **target_area** --
- **row_chunk_origin** --
- **row_chunk_height** --
- **row_bg_height** --
- **x_scroll_offset** --
- **letter_end** --

pygame_gui.core.text.html_parser module

class pygame_gui.core.text.html_parser.**HTMLParser**(*ui_theme: IUIAppearanceThemeInterface*, *combined_ids: List[str]*, *link_style: Dict[str, Any]*, *line_spacing: float = 1.0*, *text_direction: int = 0*)

Bases: HTMLParser

Parses a subset of HTML styled text so it is usable as text in pygame GUI. There are lots of text markup languages and this would be the class to swap in and out if you wanted to support them (Though this might need some refactoring to have a generic markup parser base class).

Parameters

- **ui_theme** -- The UI theme we are using - for colour and fonts.
- **combined_ids** -- The IDs for the UI element this parser instance belongs to.
- **line_spacing** -- The line spacing we use when the text is on multiple lines - defaults to 1.2.

create_styled_text_chunk(*text: str*)

Create a styled text chunk from the input text string and the current style.

Parameters
text -- The text to style up into a chunk.

Returns
A text 'chunk' all in the same style.

empty_layout_queue()

Clear out the layout queue.

error(*message*)

Feeds any parsing errors up the chain to the warning system.

Parameters
message -- The message to warn about.

handle_data(*data: str*)

Handles parsed HTML that is not a tag of any kind, ordinary text basically.

Parameters
data -- Some string data.

handle_endtag(*tag: str*)

Handles encountering an HTML end tag. Usually this will involve us popping a style off our stack of styles.

Parameters
tag -- The end tag to handle.

handle_starttag(*tag: str*, *attrs: List[Tuple[str, str]]*)

Process an HTML 'start tag' (e.g. 'b' - tags are stripped of their angle brackets) where we have a start and an end tag enclosing a range of text this is the first one of those and controls where we add the 'styling' thing to our styling stack.

Eventually we will want to expand this to handle tags like <img>.

Parameters

- **tag** -- The tag itself
- **attrs** -- Attributes of the tag.

pop_style(*key: str*)

Remove a styling element/dictionary from the stack by it's identifying key name.

Parameters
key -- The identifier.

push_style(*key: str*, *styles: Dict[str, Any]*)

Add a new styling element onto the style stack. These are single styles generally (i.e. a font size change, or a bolding of text) rather than a load of different styles all at once. The eventual style of a character/bit of text is built up by evaluating all styling elements currently on the stack when we parse that bit of text.

Styles on top of the stack will be evaluated last, so they can overwrite elements earlier in the stack (i.e. a later 'font_size' of 5 will overwrite an earlier 'font_size' of 3).

Parameters

- **key** -- Name for this styling element so, we can identify when to remove it
- **styles** -- The styling dictionary that contains the actual styling.

pygame_gui.core.text.hyperlink_text_chunk module

class pygame_gui.core.text.hyperlink_text_chunk.**HyperlinkTextChunk**(*href: str*, *text: str*, *font: IGUIFontInterface*, *underlined: bool*, *colour: Color*, *bg_colour: Color*, *hover_colour: Color*, *active_colour: Color*, *hover_underline: bool*, *text_shadow_data: Tuple[int, int, int] | None = None*, *effect_id: str | None = None*)

Bases: *TextLineChunkFTFont*

Represents a hyperlink to the layout system..

on_hovered()

Handles hovering over this text chunk with the mouse. Used for links.

on_unhovered()

Handles hovering over this text chunk with the mouse. Used for links.

set_active()

Handles clicking on this text chunk with the mouse. Used for links.

set_inactive()

Handles clicking on this text chunk with the mouse. Used for links.

pygame_gui.core.text.image_layout_rect module

class pygame_gui.core.text.image_layout_rect.**ImageLayoutRect**(*image_path*, *float_position*, *padding: Padding*)

Bases: *TextLayoutRect*

Represents an image that sits in the text.

finalise(*target_surface: Surface*, *target_area: Rect*, *row_chunk_origin: int*, *row_chunk_height: int*, *row_bg_height: int*, *x_scroll_offset: int = 0*, *letter_end: int | None = None*)

Bake the contents of this layout rect onto a surface.

Parameters

- **target_surface** --
- **target_area** --
- **row_chunk_origin** --
- **row_chunk_height** --
- **row_bg_height** --
- **x_scroll_offset** --
- **letter_end** --

pygame_gui.core.text.line_break_layout_rect module

class pygame_gui.core.text.line_break_layout_rect.**LineBreakLayoutRect**(*dimensions: Tuple[int, int]*, *font*)

Bases: *TextLayoutRect*

Represents a line break, or new line, instruction in the text to the text layout system.

Parameters

dimensions -- The dimensions of the 'line break', the height is the important thing so the new lines are spaced correctly for the last active font.

finalise(*target_surface: Surface*, *target_area: Rect*, *row_chunk_origin: int*, *row_chunk_height: int*, *row_bg_height: int*, *x_scroll_offset: int = 0*, *letter_end: int | None = None*)

Bake the contents of this layout rect onto a surface.

Parameters

- **target_surface** --

- **target_area** --
- **row_chunk_origin** --
- **row_chunk_height** --
- **row_bg_height** --
- **x_scroll_offset** --
- **letter_end** --

pygame_gui.core.text.text_box_layout module

class pygame_gui.core.text.text_box_layout.**TextBoxLayout**(*input_data_queue: Deque[TextLayoutRect], layout_rect: Rect, view_rect: Rect, line_spacing: float, default_font_data: Dict[str, Any], allow_split_dashes: bool = True, text_direction: int = 0, text_x_scroll_enabled: bool = False, editable: bool = False*)

Bases: object

Class to layout multiple lines of text to fit in a defined column.

The base of the 'column' rectangle is set once the data supplied has been laid out to fit in the width provided.

add_chunks_to_hover_group(*link_hover_chunks: List[TextLayoutRect]*)

Pass in a list of layout rectangles to add to a hover-able group. Usually used for hyperlinks.

Parameters
link_hover_chunks --

align_left_all_rows(*x_padding*)

Align all rows to the left hand side of the layout.

Parameters
x_padding -- the amount of padding to insert to on the left before the text starts.

align_right_all_rows(*x_padding*)

Align all rows to the right hand side of the layout.

Parameters
x_padding -- the amount of padding to insert to on the right before the text starts.

append_layout_rects(*new_queue*)

Add some LayoutRects on to the end of the current layout. This should be relatively fast as we don't have to rejig everything before the additions, and some of the time don't need to redraw everything either.

This is new so there may still be some bugs to iron out.

Parameters
new_queue --

backspace_at_cursor()

Deletes a single character behind the edit cursor. Mimics a standard word processor 'backspace' operation.

blit_finalised_text_to_surf(*surface: Surface*)

Lets us blit a finalised text surface to an arbitrary surface. Useful for doing stuff with text effects.

Parameters
surface -- the target surface to blit onto.

clear_effects()

Clear text layout level text effect parameters.

clear_final_surface()

Clears the finalised surface.

delete_at_cursor()

Deletes a single character in front of the edit cursor. Mimics a standard word processor 'delete' operation.

delete_selected_text()

Delete the currently selected text.

finalise_to_new()

Finalises our layout to a brand-new surface that this method creates.

finalise_to_surf(*surface: Surface*)

Take this layout, with everything positioned in the correct place and finalise it to a surface.

May be called again after changes to the layout? Update surf?

Parameters
surface -- The surface we are going to blit the contents of this layout onto.

find_cursor_position_from_click_pos(*click_pos*) → int

Find an edit text cursor position in the text from a click.

Here we don't set it, we just find it and return it.

Parameters
click_pos -- This is the pixel position we want to find the nearest cursor spot to.

Returns
an integer representing the character index position in the text

get_cursor_colour() → Color

Get the current colour of the editing carat/text cursor.

Returns
a pygame.Color object containing the current colour.

get_cursor_index()

Get the current character index, in the text layout's text, of the current edit cursor position.

Essentially the reverse of 'set_cursor_position()'.

get_cursor_pos_move_down_one_row(*last_cursor_horiz_index*)

Returns a cursor character position in the row directly above the current cursor position if possible.

get_cursor_pos_move_up_one_row(*last_cursor_horiz_index*)

Returns a cursor character position in the row directly above the last horizontal cursor position if possible.

horiz_center_all_rows(*method='rect'*)

Horizontally center all rows of text in the layout. This uses 'rectangular' centering by default, which could also be called mathematical centering. Sometimes this type of centering looks wrong - e.g. for arrows, so we instead have an option to use a 'center of mass' style centering for right facing and left facing triangles.

Parameters

method -- this is an ID for the method of centering to use, for almost all cases this will be the default 'rect' style basic centering. However, if you are trying to center an arrow you might try 'right_triangle' or 'left_triangle'

insert_layout_rects(*layout_rects: Deque[TextLayoutRect]*, *row_index: int*, *item_index: int*, *chunk_index: int*)

Insert some new layout rectangles from a queue at specific place in the current layout. Hopefully this means we only need to redo the layout after this point... we shall see.

Warning: this is a test function, it may not be up-to-date with current text layout features

Parameters

- **layout_rects** -- the new TextLayoutRects to insert.
- **row_index** -- which row we are sticking them on.
- **item_index** -- which chunk we are sticking them into.
- **chunk_index** -- where in the chunk we are sticking them.

insert_line_break(*layout_index: int*, *parser: HTMLParser | None*)

Insert a line break into the text layout at a given point.

Parameters

- **layout_index** -- the character index at which to insert the line break.
- **parser** -- An optional HTML parser for text styling data

insert_text(*text: str*, *layout_index: int*, *parser: HTMLParser | None = None*)

Insert some text into the text layout at a given point. Handy when e.g. pasting a chunk of text into an existing layout.

Parameters

- **text** -- the text to insert.
- **layout_index** -- the character index at which to insert the text.
- **parser** -- An optional HTML parser for text styling data

redraw_other_chunks(*not_these_chunks*)

Useful for text effects. TODO: no idea how this will play with images? Probably badly.

Parameters

not_these_chunks -- The chunks not to redraw

Returns

reprocess_layout_queue(*layout_rect*)

Re-lays out already parsed text data. Useful to call if the layout requirements have changed but the text data hasn't.

Parameters

layout_rect -- The new layout rectangle.

set_alpha(*alpha: int*)

Set the overall alpha level of this text box from 0 to 255. This allows us to fade text in and out of view.

Parameters

alpha -- integer from 0 to 255.

`set_cursor_colour`(*colour: Color*)

Set the colour of the editing carat/text cursor for this text layout.

Parameters
`colour` -- The colour to set it to.

`set_cursor_from_click_pos`(*click_pos*)

Set the edit cursor position in the text layout from a pixel position. Generally used to set the text editing cursor position from a mouse click.

Parameters
`click_pos` -- This is the pixel position we want the cursor to appear near to.

`set_cursor_position`(*cursor_pos*)

Set the edit cursor position in the text layout.

Parameters
`cursor_pos` -- This is the index of the character the cursor should appear before.

`set_cursor_to_end_of_current_row`()

Set the edit cursor position in the text layout to the end of the current row and returns the overall position in the text

Returns
the overall position of the cursor in the text layout, after setting it to the end of the current row

`set_cursor_to_start_of_current_row`()

Set the edit cursor position in the text layout to the end of the current row and returns the overall position in the text

Returns
the overall position of the cursor in the text layout, after setting it to the end of the current row

`set_default_text_colour`(*colour*)

Set the default text colour, used when no other colour is set for a portion of the text.

Parameters
`colour` -- the colour to use as the default text colour.

`set_default_text_shadow_colour`(*colour*)

Set the default text shadow colour, used when no other colour is set for the shadow of a portion of the text.

Parameters
`colour` -- the colour to use as the default text shadow colour.

`set_text_selection`(*start_index*, *end_index*)

Set a portion of the text layout as 'selected'. This is useful when editing chunks of text all at once (e.g. copying to a memory 'clipboard', deleting a block of text).

Parameters

- `start_index` -- the character index to start the selection area at.
- `end_index` -- the character index to end the selection area at.

`toggle_cursor`()

Toggle the visibility of the edit cursor.

Used routinely by editable text boxes to make the cursor flash to catch user attention.

turn_off_cursor()

Makes the edit test cursor invisible.

turn_on_cursor()

Makes the edit test cursor visible.

update_text_with_new_text_end_pos(*new_end_pos: int*)

Sets a new end position for the text in this block and redraws it, so we can display a 'typing' type effect. The text will only be displayed up to the index position set here.

Parameters
new_end_pos -- The new ending index for the text string.

vert_align_bottom_all_rows(*y_padding*)

Align all rows to the bottom of the layout.

Parameters
y_padding -- the amount of padding to insert below before the text starts.

vert_align_top_all_rows(*y_padding*)

Align all rows to the top of the layout.

Parameters
y_padding -- the amount of padding to insert above before the text starts.

vert_center_all_rows()

Vertically center all rows of text in the layout.

TODO: I expect we should have the arrow centering methods in here too.

pygame_gui.core.text.text_box_layout_row module

class pygame_gui.core.text.text_box_layout_row.**TextBoxLayoutRow**(*row_start_x*, *row_start_y*, *row_index*, *line_spacing*, *layout*)

Bases: Rect

A single line of text-like stuff to be used in a text box type layout.

add_item(*item:* TextLayoutRect)

Add a new item to the row. Items are added left to right.

If you wanted to build a right to left writing system layout, changing this might be a good place to start.

Parameters
item -- The new item to add to the text row

align_left_row(*start_x: int*, *floating_rects: List[*TextLayoutRect*]*)

Align this row to the left.

Parameters

- **start_x** -- Effectively the padding. Indicates how many pixels from the edge to start this row.
- **floating_rects** -- Floating rectangles we need to align around

align_right_row(*start_x: int*, *floating_rects*)

Align this row to the right.

Parameters

- **floating_rects** -- Any floating rects in the row
- **start_x** -- Effectively the padding. Indicates how many pixels from the right edge of the layout to start this row.

at_start()

Returns true if this row has no items in it.

Returns
True if we are at the start of the row.

clear()

'Clears' the current row from its target surface by setting the
area taken up by this row to transparent black.

Hopefully the target surface is supposed to be transparent black when empty.

finalise(*surface: Surface*, *current_end_pos: int | None = None*, *cumulative_letter_count: int | None = None*)

Finalise this row, turning it into pixels on a pygame surface. Generally done once we are finished applying styles and laying out the text.

Parameters

- **surface** -- The surface we are finalising this row on to.
- **current_end_pos** -- Optional parameter indicating the current end position of the visible text. This lets us do the 'typewriter' effect.
- **cumulative_letter_count** -- A count of how many letters we have already finalised. Also helps with the 'typewriter' effect.

find_cursor_pos_from_click_pos(*click_pos: Tuple[int, int]*, *num_rows: int*)

Find an edit cursor position from a pixel position - usually originating from a mouse click.

Parameters

- **num_rows** --
- **click_pos** -- The pixel position to use.

get_cursor_index() → int

Get the current character index of the cursor

Returns
the character index the edit cursor currently occupies.

horiz_center_row(*floating_rects*, *method='rect'*)

Horizontally center this row of text.

This uses 'rectangular' centering by default, which could also be called mathematical centering. Sometimes this type of centering looks wrong - e.g. for arrows, so we instead have an option to use a 'center of mass' style centering for right facing and left facing triangles.

Parameters

- **floating_rects** -- Any floating rects in the row.
- **method** -- this is an ID for the method of centering to use, for almost all cases this will be the default 'rect' style basic centering. However, if you are trying to center an arrow you might try 'right_triangle' or 'left_triangle'

insert_text(*text: str*, *letter_row_index: int*, *parser: HTMLParser | None = None*)

Insert the provided text into this row at the given location.

Parameters

- **text** -- the text to insert.
- **letter_row_index** -- the index in the row at which to insert this text.
- **parser** -- An optional HTML parser for text styling data

merge_adjacent_compatible_chunks()

Merge chunks of text next to each other in this row that have identical styles.

Should leave the minimum possible chunks in a row given the set styles for text.

rewind_row(*layout_rect_queue*)

Use this to add items from the row back onto a layout queue, useful if we've added something to the layout that means that this row needs to be re-run through the layout code.

Parameters

layout_rect_queue -- A layout queue that contains items to be laid out in order.

set_cursor_from_click_pos(*click_pos: Tuple[int, int]*, *num_rows: int*)

Set the current edit cursor position from a pixel position - usually originating from a mouse click.

Parameters

- **num_rows** --
- **click_pos** -- The pixel position to use.

set_cursor_position(*cursor_pos*)

Set the edit cursor position by a character index.

Parameters

cursor_pos -- the character index in this row to put the edit cursor after.

set_default_text_colour(*colour*)

Set the default colour of the text.

Parameters

colour -- The colour to set.

set_default_text_shadow_colour(*colour*)

Set the default colour of the text shadow.

Parameters

colour -- The colour to set.

toggle_cursor()

Toggles the visibility of the edit cursor/carat.

Generally used to make it flash on and off to catch the attention of the user.

turn_off_cursor()

Makes the edit test cursor invisible.

turn_on_cursor()

Makes the edit test cursor visible.

vert_align_items_to_row()

Align items in this row to the row's vertical position.

pygame_gui.core.text.text_layout_rect module

class pygame_gui.core.text.text_layout_rect.**Padding**(*top*, *right*, *bottom*, *left*)

Bases: tuple

bottom

Alias for field number 2

left

Alias for field number 3

right

Alias for field number 1

top

Alias for field number 0

class pygame_gui.core.text.text_layout_rect.**TextFloatPosition**(*value*)

Bases: Enum

An enumeration covering the three possible 'float' positions of a layout rect in a text layout.

'NONE' is the normal case. This rectangle will be treated as one in a row of rectangles

'LEFT' will allow other rectangles to flow around this one on the right hand side with
potentially multiple rows of shorter rectangles taking up its vertical height.

'RIGHT' the same as left except on the other side.

Generally we use this to embed images into the flow of a text box.

class pygame_gui.core.text.text_layout_rect.**TextLayoutRect**(*dimensions: Tuple[int, int]*, ***, *can_split=False*, *float_pos: TextFloatPosition = TextFloatPosition.NONE*, *should_span=False*)

Bases: Rect

A base class for use in Layouts.

can_split()

Return True if this rectangle can be split into smaller rectangles by a layout.

Returns
True if splittable, False otherwise.

clear(*optional_rect: Rect | None = None*)

Clear the space this rect takes up on the finalised surface.

Parameters
optional_rect -- A rect to clear instead of the default self rect, useful for effects.

abstract finalise(*target_surface: Surface*, *target_area: Rect*, *row_chunk_origin: int*, *row_chunk_height: int*, *row_bg_height: int*, *x_scroll_offset: int = 0*, *letter_end: int | None = None*)

Bake the contents of this layout rect onto a surface.

Parameters

- **target_surface** --
- **target_area** --

- **row_chunk_origin** --
- **row_chunk_height** --
- **row_bg_height** --
- **x_scroll_offset** --
- **letter_end** --

float_pos() → *TextFloatPosition*

Return the 'floating' status of this rectangle. can be TextFloatPosition.left, TextFloatPosition.right or TextFloatPosition.none and the default, used by most rectangles, is TextFloatPosition.none.

Used to make text flow around images.

Returns
returns the float status of this rectangle.

should_span()

Return True if this rectangle should be expanded/shrunk to fit the available width in a layout.

Returns
True if should span the width, False otherwise.

split(*requested_x: int*, *line_width: int*, *row_start_x: int*, *allow_split_dashes: bool = True*) → *TextLayoutRect* | None

Try to perform a split operation on this rectangle. Often rectangles will be split at the nearest point that is still less than the request (i.e. to the left of the request in the common left-to-right text layout case) .

Parameters

- **requested_x** -- the requested place to split this rectangle along it's width.
- **line_width** -- the width of the current line.
- **row_start_x** -- the x start position of the row.
- **allow_split_dashes** -- whether we allow text to be split with dashes either side. allowing this makes direct text editing more annoying.

vertical_overlap(*other_rect: Rect*) → bool

Test if two rectangles overlap one another in the y axis.

Parameters
other_rect --

Returns
True if they overlap.

pygame_gui.core.text.text_line_chunk module

A bit of renderable text that fits on a single line and all in the same style.

Blocks of text are made up of many text chunks. At the simplest level there would be one per line.

On creation a text chunk calculates how much space it will take up when rendered to a surface and stores this size information in a rectangle. These rectangles can then be used in layout calculations.

Once a layout for the text chunk is finalised the chunk's render function can be called to add the chunk onto it's final destination.

class pygame_gui.core.text.text_line_chunk.**TextLineChunkFTFont**(*text: str*, *font: IGUIFontInterface*, *underlined: bool*, *colour: Color | ColourGradient*, *using_default_text_colour: bool*, *bg_colour: Color | ColourGradient*, *text_shadow_data: Tuple[int, int, int, Color, bool] | None = None*, *max_dimensions: List[int] | None = None*, *effect_id: str | None = None*)

Bases: *TextLayoutRect*

A Text line chunk (text on the same horizontal line in the same style)

add_text(*input_text: str*)

Adds text to the end of this text chunk. Theoretically it should be faster to add text on the end of a text layout than sticking it in the middle.

Parameters
input_text -- The text to add.

backspace_letter_at_index(*index*)

Removes the letter before the supplied character index.

Parameters
index -- the index of the character to 'backspace' from.

can_split()

Return True if this rectangle can be split into smaller rectangles by a layout.

Returns
True if splittable, False otherwise.

clear(*optional_rect: Rect | None = None*)

Clear the finalised/rendered text in this chunk to an invisible/empty surface.

clear_effects()

Clear any effect parameters stored on the text chunk back to default values.

delete_letter_at_index(*index*)

Removes the letter after the supplied character index.

Parameters
index -- the index of the character to 'delete' from.

finalise(*target_surface: Surface*, *target_area: Rect*, *row_chunk_origin: int*, *row_chunk_height: int*, *row_bg_height: int*, *x_scroll_offset: int = 0*, *letter_end: int | None = None*)

Bake the contents of this layout rect onto a surface.

Parameters

- **target_surface** --
- **target_area** --
- **row_chunk_origin** --
- **row_chunk_height** --
- **row_bg_height** --

- **x_scroll_offset** --
- **letter_end** --

grab_pre_effect_surface()

Grab the 'pre effect' surface, used to get a 'normal' pre-effect surface to apply effects to.

insert_text(*input_text: str*, *index: int*)

Insert a new string of text into this chunk and update the chunk's data.

NOTE: We don't redraw the text immediately here as this size of this chunk changing may affect the position of other chunks later in the layout.

Parameters

- **input_text** -- the new text to insert.
- **index** -- the index we are sticking the new text at.

redraw()

Redraw a surface that has already been finalised once before.

set_alpha(*alpha: int*)

Set the overall alpha level of this text chunk from 0 to 255. This allows us to fade text in and out of view.

Parameters
alpha -- integer from 0 to 255.

set_offset_pos(*offset_pos: Tuple[int, int]*)

Set an offset that lets us move the text around a bit for effects

Parameters
offset_pos -- integers that offset the x & y position of this text chunk

set_rotation(*rotation: int*)

Set a rotation that lets us move the text around a bit for effects

Parameters
rotation -- a rotation in degrees

set_scale(*scale: float*)

Set a scale that lets us move the text around a bit for effects

Parameters
scale -- a rotation in degrees

split(*requested_x: int*, *line_width: int*, *row_start_x: int*, *allow_split_dashes: bool = True*) → *TextLayoutRect* | None

Try to perform a split operation on this chunk at the requested pixel position.

Often rectangles will be split at the nearest point that is still less than the request (i.e. to the left of the request in the common left-to-right text layout case) .

Parameters

- **requested_x** -- the requested place to split this rectangle along its width.
- **line_width** -- the width of the current line.
- **row_start_x** -- the x start position of the row.
- **allow_split_dashes** -- whether we allow text to be split with dashes either side. allowing this makes direct text editing more annoying.

split_index(*index*)

Try to perform a split operation on this chunk at the requested character index.

Parameters
index -- the requested index at which to split this rectangle along it's width.

style_match(*other_text_chunk:* TextLineChunkFTFont)

Do two layout rectangles have matching styles (generally applies only to actual text).

x_pos_to_letter_index(*x_pos: int*)

Convert a horizontal, or 'x' pixel position into a letter/character index in this text chunk. Commonly used for converting mouse clicks into letter positions for positioning the text editing cursor/carat.

Module contents

class pygame_gui.core.text.**BounceEffect**(*text_owner:* IUITextOwnerInterface, *params: Dict[str, Any]* | *None* = *None*, *text_sub_chunk:* TextLineChunkFTFont | *None* = *None*)

Bases: *TextEffect*

A bounce effect

apply_effect()

Apply the effect to the text

has_text_changed() → bool

Lets us know when the effect has changed enough to warrant us redrawing the text.

Returns
True if it is is time to redraw our text.

update(*time_delta: float*)

Updates the effect with amount of time passed since the last call to update.

Parameters
time_delta -- time in seconds since last frame.

class pygame_gui.core.text.**ExpandContractEffect**(*text_owner:* IUITextOwnerInterface, *params: Dict[str, Any]* | *None* = *None*, *text_sub_chunk:* TextLineChunkFTFont | *None* = *None*)

Bases: *TextEffect*

Expands to a specified scale and then returns

apply_effect()

Apply the effect to the text

has_text_changed() → bool

Lets us know when the effect has changed enough to warrant us redrawing the text.

Returns
True if it is is time to redraw our text.

update(*time_delta: float*)

Updates the effect with amount of time passed since the last call to update.

Parameters
time_delta -- time in seconds since last frame.

class pygame_gui.core.text.**FadeInEffect**(*text_owner:* IUITextOwnerInterface, *params: Dict[str, Any]* | *None* = *None*, *text_sub_chunk:* TextLineChunkFTFont | *None* = *None*)

Bases: *TextEffect*

A fade in effect for the text box. Allows us to fade the text, though this class just takes care of fading up an alpha value over time.

apply_effect()

Apply the effect to the text

get_final_alpha() → int

Returns the current alpha value of the fade.

Returns
The alpha value, between 0 and 255

has_text_changed() → bool

Lets us know when the fade alpha has changed enough (i.e. by a whole int) to warrant us redrawing the text box with the new alpha value.

Returns
True if it is is time to redraw our text.

update(*time_delta: float*)

Updates the fade with amount of time passed since the last call to update.

Parameters
time_delta -- time in seconds since last frame.

class pygame_gui.core.text.**FadeOutEffect**(*text_owner:* IUITextOwnerInterface, *params: Dict[str, Any]* | *None* = *None*, *text_sub_chunk:* TextLineChunkFTFont | *None* = *None*)

Bases: *TextEffect*

A fade out effect for the text box. Allows us to fade the text, though this class just takes care of fading out an alpha value over time.

apply_effect()

Apply the effect to the text

get_final_alpha() → int

Returns the current alpha value of the fade.

Returns
The alpha value, between 0 and 255

has_text_changed() → bool

Lets us know when the fade alpha has changed enough (i.e. by a whole int) to warrant us redrawing the text box with the new alpha value.

Returns
True if it is is time to redraw our text.

update(*time_delta: float*)

Updates the fade with amount of time passed since the last call to update.

Parameters
time_delta -- time in seconds since last frame.

class pygame_gui.core.text.HTMLParser(*ui_theme: IUIAppearanceThemeInterface*, *combined_ids: List[str]*, *link_style: Dict[str, Any]*, *line_spacing: float = 1.0*, *text_direction: int = 0*)

Bases: HTMLParser

Parses a subset of HTML styled text so it is usable as text in pygame GUI. There are lots of text markup languages and this would be the class to swap in and out if you wanted to support them (Though this might need some refactoring to have a generic markup parser base class).

Parameters

- **ui_theme** -- The UI theme we are using - for colour and fonts.
- **combined_ids** -- The IDs for the UI element this parser instance belongs to.
- **line_spacing** -- The line spacing we use when the text is on multiple lines - defaults to 1.2.

create_styled_text_chunk(*text: str*)

Create a styled text chunk from the input text string and the current style.

Parameters

text -- The text to style up into a chunk.

Returns

A text 'chunk' all in the same style.

empty_layout_queue()

Clear out the layout queue.

error(*message*)

Feeds any parsing errors up the chain to the warning system.

Parameters

message -- The message to warn about.

handle_data(*data: str*)

Handles parsed HTML that is not a tag of any kind, ordinary text basically.

Parameters

data -- Some string data.

handle_endtag(*tag: str*)

Handles encountering an HTML end tag. Usually this will involve us popping a style off our stack of styles.

Parameters

tag -- The end tag to handle.

handle_starttag(*tag: str*, *attrs: List[Tuple[str, str]]*)

Process an HTML 'start tag' (e.g. 'b' - tags are stripped of their angle brackets) where we have a start and an end tag enclosing a range of text this is the first one of those and controls where we add the 'styling' thing to our styling stack.

Eventually we will want to expand this to handle tags like <img>.

Parameters

- **tag** -- The tag itself
- **attrs** -- Attributes of the tag.

pop_style(*key: str*)

Remove a styling element/dictionary from the stack by it's identifying key name.

Parameters
key -- The identifier.

push_style(*key: str*, *styles: Dict[str, Any]*)

Add a new styling element onto the style stack. These are single styles generally (i.e. a font size change, or a bolding of text) rather than a load of different styles all at once. The eventual style of a character/bit of text is built up by evaluating all styling elements currently on the stack when we parse that bit of text.

Styles on top of the stack will be evaluated last, so they can overwrite elements earlier in the stack (i.e. a later 'font_size' of 5 will overwrite an earlier 'font_size' of 3).

Parameters

- **key** -- Name for this styling element so, we can identify when to remove it
- **styles** -- The styling dictionary that contains the actual styling.

class pygame_gui.core.text.**HorizRuleLayoutRect**(*height: int*, *colour_or_gradient: Color*, *rule_dimensions: Tuple[int, int] = (-1, 1)*, *has_shade: bool = True*, *alignment=0*)

Bases: *TextLayoutRect*

Represents a horizontal rule in the HTML style. This is normally a line across the width of the layout block area, but styling options can provide some variation on that theme.

Parameters

- **height** -- the current line height of the layout/font we are using when invoking the rule.
- **colour_or_gradient** -- the colour or gradient of the rule.
- **rule_dimensions** -- the dimensions of the rule itself, normally it is 1 pixel tall and the width of the text block layout wide.
- **has_shade** -- whether the rule has 'shading' which by default is just another alpha'd line beneath it to add some depth. Doesn't work great if the line has more height to it.
- **alignment** -- ALIGN_CENTER, ALIGN_LEFT or ALIGN_RIGHT. ALIGN_CENTER is the default.

finalise(*target_surface: Surface*, *target_area: Rect*, *row_chunk_origin: int*, *row_chunk_height: int*, *row_bg_height: int*, *x_scroll_offset: int = 0*, *letter_end: int | None = None*)

Bake the contents of this layout rect onto a surface.

Parameters

- **target_surface** --
- **target_area** --
- **row_chunk_origin** --
- **row_chunk_height** --
- **row_bg_height** --
- **x_scroll_offset** --
- **letter_end** --

class pygame_gui.core.text.**HyperlinkTextChunk**(*href: str*, *text: str*, *font: IGUIFontInterface*, *underlined: bool*, *colour: Color*, *bg_colour: Color*, *hover_colour: Color*, *active_colour: Color*, *hover_underline: bool*, *text_shadow_data: Tuple[int, int, int] | None = None*, *effect_id: str | None = None*)

Bases: *TextLineChunkFTFont*

Represents a hyperlink to the layout system..

on_hovered()

Handles hovering over this text chunk with the mouse. Used for links.

on_unhovered()

Handles hovering over this text chunk with the mouse. Used for links.

set_active()

Handles clicking on this text chunk with the mouse. Used for links.

set_inactive()

Handles clicking on this text chunk with the mouse. Used for links.

class pygame_gui.core.text.**ImageLayoutRect**(*image_path*, *float_position*, *padding: Padding*)

Bases: *TextLayoutRect*

Represents an image that sits in the text.

finalise(*target_surface: Surface*, *target_area: Rect*, *row_chunk_origin: int*, *row_chunk_height: int*, *row_bg_height: int*, *x_scroll_offset: int = 0*, *letter_end: int | None = None*)

Bake the contents of this layout rect onto a surface.

Parameters

- **target_surface** --
- **target_area** --
- **row_chunk_origin** --
- **row_chunk_height** --
- **row_bg_height** --
- **x_scroll_offset** --
- **letter_end** --

class pygame_gui.core.text.**LineBreakLayoutRect**(*dimensions: Tuple[int, int]*, *font*)

Bases: *TextLayoutRect*

Represents a line break, or new line, instruction in the text to the text layout system.

Parameters

dimensions -- The dimensions of the 'line break', the height is the important thing so the new lines are spaced correctly for the last active font.

finalise(*target_surface: Surface*, *target_area: Rect*, *row_chunk_origin: int*, *row_chunk_height: int*, *row_bg_height: int*, *x_scroll_offset: int = 0*, *letter_end: int | None = None*)

Bake the contents of this layout rect onto a surface.

Parameters

- **target_surface** --

- **target_area** --
- **row_chunk_origin** --
- **row_chunk_height** --
- **row_bg_height** --
- **x_scroll_offset** --
- **letter_end** --

class pygame_gui.core.text.**ShakeEffect**(*text_owner: IUITextOwnerInterface*, *params: Dict[str, Any] | None = None*, *text_sub_chunk: TextLineChunkFTFont | None = None*)

Bases: *TextEffect*

A shake effect

apply_effect()

Apply the effect to the text

has_text_changed() → bool

Lets us know when the effect has changed enough to warrant us redrawing the text.

Returns
True if it is is time to redraw our text.

update(*time_delta: float*)

Updates the effect with amount of time passed since the last call to update.

Parameters
time_delta -- time in seconds since last frame.

class pygame_gui.core.text.**SimpleTestLayoutRect**(*dimensions: Tuple[int, int]*, ***, *create_split_points=True*, *float_pos=TextFloatPosition.NONE*)

Bases: *TextLayoutRect*

Useful class for testing layout generation. Multi coloured boxes make it easy to distinguish different layout Rects from one another and it's possible to set all the layout options in the constructor/initializer to represent different types of layout rect.

finalise(*target_surface: Surface*, *target_area: Rect*, *row_chunk_origin: int*, *row_chunk_height: int*, *row_bg_height: int*, *x_scroll_offset: int = 0*, *letter_end: int | None = None*)

Bake the contents of this layout rect onto a surface.

Parameters

- **target_surface** --
- **target_area** --
- **row_chunk_origin** --
- **row_chunk_height** --
- **row_bg_height** --
- **x_scroll_offset** --
- **letter_end** --

static gen_random_colour()

Creates a random colour using the golden ratio method.

Helps make the test layout rects reasonably distinctive from each other.

split(*requested_x: int*, *line_width: int*, *row_start_x: int*, *allow_split_dashes: bool = True*)

Try to perform a split operation on this rectangle. Often rectangles will be split at the nearest point that is still less than the request (i.e. to the left of the request in the common left-to-right text layout case) .

Parameters

- **requested_x** -- the requested place to split this rectangle along it's width.
- **line_width** -- the width of the current line.
- **row_start_x** -- the x start position of the row.
- **allow_split_dashes** -- whether we allow text to be split with dashes either side. allowing this makes direct text editing more annoying.

class pygame_gui.core.text.**TextBoxLayout**(*input_data_queue: Deque[*TextLayoutRect*]*, *layout_rect: Rect*, *view_rect: Rect*, *line_spacing: float*, *default_font_data: Dict[str, Any]*, *allow_split_dashes: bool = True*, *text_direction: int = 0*, *text_x_scroll_enabled: bool = False*, *editable: bool = False*)

Bases: object

Class to layout multiple lines of text to fit in a defined column.

The base of the 'column' rectangle is set once the data supplied has been laid out to fit in the width provided.

add_chunks_to_hover_group(*link_hover_chunks: List[*TextLayoutRect*]*)

Pass in a list of layout rectangles to add to a hover-able group. Usually used for hyperlinks.

Parameters
link_hover_chunks --

align_left_all_rows(*x_padding*)

Align all rows to the left hand side of the layout.

Parameters
x_padding -- the amount of padding to insert to on the left before the text starts.

align_right_all_rows(*x_padding*)

Align all rows to the right hand side of the layout.

Parameters
x_padding -- the amount of padding to insert to on the right before the text starts.

append_layout_rects(*new_queue*)

Add some LayoutRects on to the end of the current layout. This should be relatively fast as we don't have to rejig everything before the additions, and some of the time don't need to redraw everything either.

This is new so there may still be some bugs to iron out.

Parameters
new_queue --

backspace_at_cursor()

Deletes a single character behind the edit cursor. Mimics a standard word processor 'backspace' operation.

blit_finalised_text_to_surf(*surface: Surface*)

Lets us blit a finalised text surface to an arbitrary surface. Useful for doing stuff with text effects.

Parameters
surface -- the target surface to blit onto.

clear_effects()

Clear text layout level text effect parameters.

clear_final_surface()

Clears the finalised surface.

delete_at_cursor()

Deletes a single character in front of the edit cursor. Mimics a standard word processor 'delete' operation.

delete_selected_text()

Delete the currently selected text.

finalise_to_new()

Finalises our layout to a brand-new surface that this method creates.

finalise_to_surf(*surface: Surface*)

Take this layout, with everything positioned in the correct place and finalise it to a surface.

May be called again after changes to the layout? Update surf?

Parameters
surface -- The surface we are going to blit the contents of this layout onto.

find_cursor_position_from_click_pos(*click_pos*) → int

Find an edit text cursor position in the text from a click.

Here we don't set it, we just find it and return it.

Parameters
click_pos -- This is the pixel position we want to find the nearest cursor spot to.

Returns
an integer representing the character index position in the text

get_cursor_colour() → Color

Get the current colour of the editing carat/text cursor.

Returns
a pygame.Color object containing the current colour.

get_cursor_index()

Get the current character index, in the text layout's text, of the current edit cursor position.

Essentially the reverse of 'set_cursor_position()'.

get_cursor_pos_move_down_one_row(*last_cursor_horiz_index*)

Returns a cursor character position in the row directly above the current cursor position if possible.

get_cursor_pos_move_up_one_row(*last_cursor_horiz_index*)

Returns a cursor character position in the row directly above the last horizontal cursor position if possible.

horiz_center_all_rows(*method='rect'*)

Horizontally center all rows of text in the layout. This uses 'rectangular' centering by default, which could also be called mathematical centering. Sometimes this type of centering looks wrong - e.g. for arrows, so we instead have an option to use a 'center of mass' style centering for right facing and left facing triangles.

Parameters

method -- this is an ID for the method of centering to use, for almost all cases this will be the default 'rect' style basic centering. However, if you are trying to center an arrow you might try 'right_triangle' or 'left_triangle'

insert_layout_rects(*layout_rects: Deque[TextLayoutRect]*, *row_index: int*, *item_index: int*, *chunk_index: int*)

Insert some new layout rectangles from a queue at specific place in the current layout. Hopefully this means we only need to redo the layout after this point... we shall see.

Warning: this is a test function, it may not be up-to-date with current text layout features

Parameters

- **layout_rects** -- the new TextLayoutRects to insert.
- **row_index** -- which row we are sticking them on.
- **item_index** -- which chunk we are sticking them into.
- **chunk_index** -- where in the chunk we are sticking them.

insert_line_break(*layout_index: int*, *parser: HTMLParser | None*)

Insert a line break into the text layout at a given point.

Parameters

- **layout_index** -- the character index at which to insert the line break.
- **parser** -- An optional HTML parser for text styling data

insert_text(*text: str*, *layout_index: int*, *parser: HTMLParser | None = None*)

Insert some text into the text layout at a given point. Handy when e.g. pasting a chunk of text into an existing layout.

Parameters

- **text** -- the text to insert.
- **layout_index** -- the character index at which to insert the text.
- **parser** -- An optional HTML parser for text styling data

redraw_other_chunks(*not_these_chunks*)

Useful for text effects. TODO: no idea how this will play with images? Probably badly.

Parameters

not_these_chunks -- The chunks not to redraw

Returns

reprocess_layout_queue(*layout_rect*)

Re-lays out already parsed text data. Useful to call if the layout requirements have changed but the text data hasn't.

Parameters

layout_rect -- The new layout rectangle.

set_alpha(*alpha: int*)

Set the overall alpha level of this text box from 0 to 255. This allows us to fade text in and out of view.

Parameters

alpha -- integer from 0 to 255.

set_cursor_colour(*colour: Color*)

Set the colour of the editing carat/text cursor for this text layout.

Parameters
colour -- The colour to set it to.

set_cursor_from_click_pos(*click_pos*)

Set the edit cursor position in the text layout from a pixel position. Generally used to set the text editing cursor position from a mouse click.

Parameters
click_pos -- This is the pixel position we want the cursor to appear near to.

set_cursor_position(*cursor_pos*)

Set the edit cursor position in the text layout.

Parameters
cursor_pos -- This is the index of the character the cursor should appear before.

set_cursor_to_end_of_current_row()

Set the edit cursor position in the text layout to the end of the current row and returns the overall position in the text

Returns
the overall position of the cursor in the text layout, after setting it to the end of the current row

set_cursor_to_start_of_current_row()

Set the edit cursor position in the text layout to the end of the current row and returns the overall position in the text

Returns
the overall position of the cursor in the text layout, after setting it to the end of the current row

set_default_text_colour(*colour*)

Set the default text colour, used when no other colour is set for a portion of the text.

Parameters
colour -- the colour to use as the default text colour.

set_default_text_shadow_colour(*colour*)

Set the default text shadow colour, used when no other colour is set for the shadow of a portion of the text.

Parameters
colour -- the colour to use as the default text shadow colour.

set_text_selection(*start_index*, *end_index*)

Set a portion of the text layout as 'selected'. This is useful when editing chunks of text all at once (e.g. copying to a memory 'clipboard', deleting a block of text).

Parameters

- **start_index** -- the character index to start the selection area at.
- **end_index** -- the character index to end the selection area at.

toggle_cursor()

Toggle the visibility of the edit cursor.

Used routinely by editable text boxes to make the cursor flash to catch user attention.

turn_off_cursor()

Makes the edit test cursor invisible.

turn_on_cursor()

Makes the edit test cursor visible.

update_text_with_new_text_end_pos(*new_end_pos: int*)

Sets a new end position for the text in this block and redraws it, so we can display a 'typing' type effect. The text will only be displayed up to the index position set here.

Parameters

new_end_pos -- The new ending index for the text string.

vert_align_bottom_all_rows(*y_padding*)

Align all rows to the bottom of the layout.

Parameters

y_padding -- the amount of padding to insert below before the text starts.

vert_align_top_all_rows(*y_padding*)

Align all rows to the top of the layout.

Parameters

y_padding -- the amount of padding to insert above before the text starts.

vert_center_all_rows()

Vertically center all rows of text in the layout.

TODO: I expect we should have the arrow centering methods in here too.

class pygame_gui.core.text.**TextBoxLayoutRow**(*row_start_x*, *row_start_y*, *row_index*, *line_spacing*, *layout*)

Bases: Rect

A single line of text-like stuff to be used in a text box type layout.

add_item(*item:* TextLayoutRect)

Add a new item to the row. Items are added left to right.

If you wanted to build a right to left writing system layout, changing this might be a good place to start.

Parameters

item -- The new item to add to the text row

align_left_row(*start_x: int*, *floating_rects: List[*TextLayoutRect*]*)

Align this row to the left.

Parameters

- **start_x** -- Effectively the padding. Indicates how many pixels from the edge to start this row.
- **floating_rects** -- Floating rectangles we need to align around

align_right_row(*start_x: int*, *floating_rects*)

Align this row to the right.

Parameters

- **floating_rects** -- Any floating rects in the row
- **start_x** -- Effectively the padding. Indicates how many pixels from the right edge of the layout to start this row.

`at_start`()

Returns true if this row has no items in it.

Returns
True if we are at the start of the row.

`clear`()

'Clears' the current row from its target surface by setting the
area taken up by this row to transparent black.

Hopefully the target surface is supposed to be transparent black when empty.

`finalise`(*surface: Surface*, *current_end_pos: int | None = None*, *cumulative_letter_count: int | None = None*)

Finalise this row, turning it into pixels on a pygame surface. Generally done once we are finished applying styles and laying out the text.

Parameters

- **surface** -- The surface we are finalising this row on to.
- **current_end_pos** -- Optional parameter indicating the current end position of the visible text. This lets us do the 'typewriter' effect.
- **cumulative_letter_count** -- A count of how many letters we have already finalised. Also helps with the 'typewriter' effect.

`find_cursor_pos_from_click_pos`(*click_pos: Tuple[int, int]*, *num_rows: int*)

Find an edit cursor position from a pixel position - usually originating from a mouse click.

Parameters

- **num_rows** --
- **click_pos** -- The pixel position to use.

`get_cursor_index`() → int

Get the current character index of the cursor

Returns
the character index the edit cursor currently occupies.

`horiz_center_row`(*floating_rects*, *method='rect'*)

Horizontally center this row of text.

This uses 'rectangular' centering by default, which could also be called mathematical centering. Sometimes this type of centering looks wrong - e.g. for arrows, so we instead have an option to use a 'center of mass' style centering for right facing and left facing triangles.

Parameters

- **floating_rects** -- Any floating rects in the row.
- **method** -- this is an ID for the method of centering to use, for almost all cases this will be the default 'rect' style basic centering. However, if you are trying to center an arrow you might try 'right_triangle' or 'left_triangle'

`insert_text`(*text: str*, *letter_row_index: int*, *parser: HTMLParser | None = None*)

Insert the provided text into this row at the given location.

Parameters

- **text** -- the text to insert.

- **letter_row_index** -- the index in the row at which to insert this text.
- **parser** -- An optional HTML parser for text styling data

merge_adjacent_compatible_chunks()

Merge chunks of text next to each other in this row that have identical styles.

Should leave the minimum possible chunks in a row given the set styles for text.

rewind_row(*layout_rect_queue*)

Use this to add items from the row back onto a layout queue, useful if we've added something to the layout that means that this row needs to be re-run through the layout code.

Parameters
layout_rect_queue -- A layout queue that contains items to be laid out in order.

set_cursor_from_click_pos(*click_pos: Tuple[int, int]*, *num_rows: int*)

Set the current edit cursor position from a pixel position - usually originating from a mouse click.

Parameters

- **num_rows** --
- **click_pos** -- The pixel position to use.

set_cursor_position(*cursor_pos*)

Set the edit cursor position by a character index.

Parameters
cursor_pos -- the character index in this row to put the edit cursor after.

set_default_text_colour(*colour*)

Set the default colour of the text.

Parameters
colour -- The colour to set.

set_default_text_shadow_colour(*colour*)

Set the default colour of the text shadow.

Parameters
colour -- The colour to set.

toggle_cursor()

Toggles the visibility of the edit cursor/carat.

Generally used to make it flash on and off to catch the attention of the user.

turn_off_cursor()

Makes the edit test cursor invisible.

turn_on_cursor()

Makes the edit test cursor visible.

vert_align_items_to_row()

Align items in this row to the row's vertical position.

class pygame_gui.core.text.**TextEffect**

Bases: object

Base class for text effects

apply_effect()

Apply the effect to the text

get_final_alpha() → int

The alpha value to draw the text box with. By default it is 255.

Returns
The default alpha value for a text box.

has_text_changed() → bool

Stub that returns False

Returns
False

update(*time_delta: float*)

Stub for overriding.

Parameters
time_delta -- time in seconds since last frame.

class pygame_gui.core.text.**TextFloatPosition**(*value*)

Bases: Enum

An enumeration covering the three possible 'float' positions of a layout rect in a text layout.

'NONE' is the normal case. This rectangle will be treated as one in a row of rectangles

'LEFT' will allow other rectangles to flow around this one on the right hand side with
potentially multiple rows of shorter rectangles taking up its vertical height.

'RIGHT' the same as left except on the other side.

Generally we use this to embed images into the flow of a text box.

class pygame_gui.core.text.**TextLayoutRect**(*dimensions: Tuple[int, int]*, ***, *can_split=False*, *float_pos: TextFloatPosition = TextFloatPosition.NONE*, *should_span=False*)

Bases: Rect

A base class for use in Layouts.

can_split()

Return True if this rectangle can be split into smaller rectangles by a layout.

Returns
True if splittable, False otherwise.

clear(*optional_rect: Rect | None = None*)

Clear the space this rect takes up on the finalised surface.

Parameters
optional_rect -- A rect to clear instead of the default self rect, useful for effects.

abstract finalise(*target_surface: Surface*, *target_area: Rect*, *row_chunk_origin: int*, *row_chunk_height: int*, *row_bg_height: int*, *x_scroll_offset: int = 0*, *letter_end: int | None = None*)

Bake the contents of this layout rect onto a surface.

Parameters

- **target_surface** --
- **target_area** --

- **row_chunk_origin** --
- **row_chunk_height** --
- **row_bg_height** --
- **x_scroll_offset** --
- **letter_end** --

float_pos() → *TextFloatPosition*

Return the 'floating' status of this rectangle. can be TextFloatPosition.left, TextFloatPosition.right or TextFloatPosition.none and the default, used by most rectangles, is TextFloatPosition.none.

Used to make text flow around images.

Returns
returns the float status of this rectangle.

should_span()

Return True if this rectangle should be expanded/shrunk to fit the available width in a layout.

Returns
True if should span the width, False otherwise.

split(*requested_x: int*, *line_width: int*, *row_start_x: int*, *allow_split_dashes: bool = True*) → *TextLayoutRect* | None

Try to perform a split operation on this rectangle. Often rectangles will be split at the nearest point that is still less than the request (i.e. to the left of the request in the common left-to-right text layout case) .

Parameters

- **requested_x** -- the requested place to split this rectangle along it's width.
- **line_width** -- the width of the current line.
- **row_start_x** -- the x start position of the row.
- **allow_split_dashes** -- whether we allow text to be split with dashes either side. allowing this makes direct text editing more annoying.

vertical_overlap(*other_rect: Rect*) → bool

Test if two rectangles overlap one another in the y axis.

Parameters
other_rect --

Returns
True if they overlap.

class pygame_gui.core.text.**TextLineChunkFTFont**(*text: str*, *font: IGUIFontInterface*, *underlined: bool*, *colour: Color | ColourGradient*, *using_default_text_colour: bool*, *bg_colour: Color | ColourGradient*, *text_shadow_data: Tuple[int, int, int, Color, bool] | None = None*, *max_dimensions: List[int] | None = None*, *effect_id: str | None = None*)

Bases: *TextLayoutRect*

A Text line chunk (text on the same horizontal line in the same style)

add_text(*input_text: str*)

Adds text to the end of this text chunk. Theoretically it should be faster to add text on the end of a text layout than sticking it in the middle.

Parameters
input_text -- The text to add.

backspace_letter_at_index(*index*)

Removes the letter before the supplied character index.

Parameters
index -- the index of the character to 'backspace' from.

can_split()

Return True if this rectangle can be split into smaller rectangles by a layout.

Returns
True if splittable, False otherwise.

clear(*optional_rect: Rect | None = None*)

Clear the finalised/rendered text in this chunk to an invisible/empty surface.

clear_effects()

Clear any effect parameters stored on the text chunk back to default values.

delete_letter_at_index(*index*)

Removes the letter after the supplied character index.

Parameters
index -- the index of the character to 'delete' from.

finalise(*target_surface: Surface*, *target_area: Rect*, *row_chunk_origin: int*, *row_chunk_height: int*, *row_bg_height: int*, *x_scroll_offset: int = 0*, *letter_end: int | None = None*)

Bake the contents of this layout rect onto a surface.

Parameters

- **target_surface** --
- **target_area** --
- **row_chunk_origin** --
- **row_chunk_height** --
- **row_bg_height** --
- **x_scroll_offset** --
- **letter_end** --

grab_pre_effect_surface()

Grab the 'pre effect' surface, used to get a 'normal' pre-effect surface to apply effects to.

insert_text(*input_text: str*, *index: int*)

Insert a new string of text into this chunk and update the chunk's data.

NOTE: We don't redraw the text immediately here as this size of this chunk changing may affect the position of other chunks later in the layout.

Parameters

- **input_text** -- the new text to insert.
- **index** -- the index we are sticking the new text at.

redraw()

Redraw a surface that has already been finalised once before.

set_alpha(*alpha: int*)

Set the overall alpha level of this text chunk from 0 to 255. This allows us to fade text in and out of view.

Parameters
alpha -- integer from 0 to 255.

set_offset_pos(*offset_pos: Tuple[int, int]*)

Set an offset that lets us move the text around a bit for effects

Parameters
offset_pos -- integers that offset the x & y position of this text chunk

set_rotation(*rotation: int*)

Set a rotation that lets us move the text around a bit for effects

Parameters
rotation -- a rotation in degrees

set_scale(*scale: float*)

Set a scale that lets us move the text around a bit for effects

Parameters
scale -- a rotation in degrees

split(*requested_x: int*, *line_width: int*, *row_start_x: int*, *allow_split_dashes: bool = True*) → *TextLayoutRect* | None

Try to perform a split operation on this chunk at the requested pixel position.

Often rectangles will be split at the nearest point that is still less than the request (i.e. to the left of the request in the common left-to-right text layout case) .

Parameters

- **requested_x** -- the requested place to split this rectangle along its width.
- **line_width** -- the width of the current line.
- **row_start_x** -- the x start position of the row.
- **allow_split_dashes** -- whether we allow text to be split with dashes either side. allowing this makes direct text editing more annoying.

split_index(*index*)

Try to perform a split operation on this chunk at the requested character index.

Parameters
index -- the requested index at which to split this rectangle along it's width.

style_match(*other_text_chunk: TextLineChunkFTFont*)

Do two layout rectangles have matching styles (generally applies only to actual text).

x_pos_to_letter_index(*x_pos: int*)

Convert a horizontal, or 'x' pixel position into a letter/character index in this text chunk. Commonly used for converting mouse clicks into letter positions for positioning the text editing cursor/carat.

class pygame_gui.core.text.**TiltEffect**(*text_owner: IUITextOwnerInterface*, *params: Dict[str, Any] | None = None*, *text_sub_chunk: TextLineChunkFTFont | None = None*)

Bases: *TextEffect*

Tilts to an angle and then returns

apply_effect()

Apply the effect to the text

has_text_changed() → bool

Lets us know when the effect has changed enough to warrant us redrawing the text.

Returns
True if it is is time to redraw our text.

update(*time_delta: float*)

Updates the effect with amount of time passed since the last call to update.

Parameters
time_delta -- time in seconds since last frame.

class pygame_gui.core.text.**TypingAppearEffect**(*text_owner:* IUITextOwnerInterface, *params: Dict[str, Any]* | *None* = *None*, *text_sub_chunk:* TextLineChunkFTFont | *None* = *None*)

Bases: *TextEffect*

Does that 'typewriter' effect where the text in a box appears one character at a time as if they were being typed by an invisible hand.

apply_effect()

Apply the effect to the text

has_text_changed() → bool

Test if we should redraw the text.

Returns
True if we should redraw, False otherwise.

update(*time_delta: float*)

Updates the effect with amount of time passed since the last call to update. Adds a new letter to the progress every self.time_per_letter seconds.

Parameters
time_delta -- time in seconds since last frame.

Submodules

pygame_gui.core.colour_gradient module

class pygame_gui.core.colour_gradient.**ColourGradient**(*angle_direction: int*, *colour_1: Color*, *colour_2: Color*, *colour_3: Color* | *None* = *None*)

Bases: *IColourGradientInterface*

Creates a small surface containing a smooth gradient between two or three colours.

Parameters

- **angle_direction** -- Angle direction of the gradient in degrees.

- **colour_1** -- The first colour of the gradient.
- **colour_2** -- The second colour of the gradient.
- **colour_3** -- An optional third colour for the gradient.

apply_gradient_to_surface(*input_surface: Surface*, *rect: Rect | None = None*)

Applies this gradient to a specified input surface using blending multiplication. As a result this method works best when the input surface is a mostly white, stencil shape type surface.

Parameters

- **input_surface** --
- **rect** -- The rectangle on the surface to apply the gradient to. If None, applies to the whole surface.

pygame_gui.core.colour_parser module

pygame_gui.core.colour_parser.py A Functional Based Module for the parsing of Colour Strings in pygame_gui

Use Notes:

Mostly all that one will ever need is these 5 functions

- **parse_colour_or_gradient_string**: Attempt to parse a string that may be a colour or gradient into its respective value
- **is_valid_colour_string**: Check if a string represents a valid pygame Color
- **is_valid_gradient_string**: Check if a string represents a valid pygame_gui.core.colour_gradient.ColourGradient
- **parse_colour_string**: Parse a string into a pygame Color
- **parse_gradient_string**: Parse a string into a pygame_gui.core.colour_gradient.ColourGradient

The documentation for what counts as a 'Valid' colour and gradient string can be found in the Theme Guide of the pygame_gui documentation at https://pygame-gui.readthedocs.io/en/v_065/theme_guide.html

Developer Notes:

How it Works:

This module works through pairs of validating and parsing functions at the value, colour, and gradient levels Hopefully everything should be generic enough that any new colour models or representations can be easily added without much bloat or *magic*

Parsing A Colour:

Simply, The parsing of a colour is done by simply checking if there is any valid function cashed in the _colourParsers dictionary, then calling the paired parsing function to get its value Therefore, this system is not concrete at all, and should be extremely extensible to add **any 2 functions that can validate and parse a developer-determined schema**

Parsing A Gradient:

As of now, the gradient parser is implemented in such a way where it assumes that all commas outside an enclosing glyph (Any comma not inside a (), [], or {}) is a separator in a gradient list Generally, if creating new colour string schemas, this will break if there is a new colour which uses commas not enclosed in a glyph. This shouldn't be a problem right now, but it is worth noting as a warning in case of any additions to this parser TL,DR: Dev life will be easier if it is ensured that commas in colour schemas are inside of parentheses, brackets, or curly braces (like "rgb(20, 20, 20)")

class pygame_gui.core.colour_parser.**ColourValueParserData**(**args*, ***kwargs*)

Bases: dict

class pygame_gui.core.colour_parser.**NumParserType**(*value*)

Bases: Enum

Enum for the supported value types in colour strings

pygame_gui.core.colour_parser.**expand_shorthand_hex**(*strdata: str*) → str

Expand a Shorthand Hex Color

Example:
#FA2 -> #FFAA22

Parameters
strdata (*str*) -- the hex string to expand

Returns
An expanded hex string

Return type
str

pygame_gui.core.colour_parser.**get_commas_outside_enclosing_glyphs**(*strdata: str*) → List[int]

In the colour_parser module, This function is used to determine where to split gradient strings in order to get the full list of colours and the final degrees

Developer Notes:

- Used to determine which top level commas should be used to separate gradients
- Used with the assumption that colour values themselves do not have glyphs that are left opened
- Used with the assumption that top level commas used to separate gradients are not within any sort of enclosing glyph
- An **enclosing glyph** is like a parentheses, bracket, curly brace, or something of the sort

Parameters
strdata (*str*) -- the string to check

Returns
A list of the indices of the commas determined to be outside any enclosing parentheses

Return type
list[int]

pygame_gui.core.colour_parser.**is_valid_cmy_string**(*strdata: str*) → bool

Validate CMY Color string in the format "cmy(percentage, percentage, percentage)"

Value Parameter Descriptions:

- **percentage**: either an integer value from 0 to 100 with "%" appended at the end or a float value ranging from 0 to 1

Examples:

- "cmy(.4, .7, 80%)"

Parameters
strdata (*str*) -- the cmy string to validate

Returns
A boolean determining whether the string fits the determined cmy schema

Return type
bool

pygame_gui.core.colour_parser.**is_valid_colour_name**(*strdata: str*)

Validate Colour name string to be recognizable by pygame_gui as a valid colour_name

As of the writing of these documentations, all colour names defined are from the CSS colours given by all major browsers, which can be found here: https://w3schools.sinsixx.com/css/css_colornames.asp.htm

All colour names are not case-sensitive, so RED, Red, and red all represent the same value

Parameters
strdata (*str*) -- the colour name to validate

Returns
A boolean determining whether strdata is recognized as a colour name and has a locatable colour value

Return type
bool

pygame_gui.core.colour_parser.**is_valid_colour_string**(*strdata: str*) → bool

Validate a colour string using the available colour string validator and parsers in pygame_gui

Parameters
strdata -- the colour string data to validate

Returns
A boolean on whether any valid parser could be found for the string data

Return type
bool

pygame_gui.core.colour_parser.**is_valid_gradient_string**(*strdata: str*) → bool

Validate a gradient string A gradient string should consist of a 3 or 4 length comma separated list, with the first values being any valid colour strings and the last value representing a degree angle for the direction of the gradient. Examples: - "red,blue,40deg" - "#f23,rgb(30, 70, 230),hsv(50, 70%, 90%),50"

Parameters
strdata (*str*) -- the gradient string to validate

Returns
A boolean indicating whether the gradient string is valid or not

Return type
bool

pygame_gui.core.colour_parser.**is_valid_hex_string**(*strdata: str*) → bool

Validate Hex Color string in the format "#FFF", "#FFFF", "#FFFFFF", or "#FFFFFFFF"

Value Parameter Descriptions:
F: any hexadecimal digit between 0 and F inclusive

Examples:

- "#A3F" (Shorthand)
- "#98C4" (Shorthand with Alpha)
- "#F3FAFF" (Full)
- "#BD24A017" (Full, with Alpha)

Parameters
strdata (*str*) -- the hex string to validate

Returns
A boolean determining whether the string fits the determined hex schema

Return type
bool

pygame_gui.core.colour_parser.**is_valid_hsl_string**(*strdata: str*) → bool

Validate HSL Color string in the format "hsl(degree, percentage, percentage)"

Value Parameter Descriptions:

- **percentage**: either an integer value from 0 to 100 with "%" appended at the end or a float value ranging from 0 to 1
- **degree**: a value between 0 and 360 with the "deg" unit optionally appended to the end

Examples:

- "hsl(30, 0.6, 0.7)"
- "hsl(30deg, 40%, .5)"

Parameters
strdata (*str*) -- the hsl string to validate

Returns
A boolean determining whether the string fits the determined hsl schema

Return type
bool

pygame_gui.core.colour_parser.**is_valid_hsla_string**(*strdata: str*) → bool

Validate HSLA Color string in the format "hsla(degree, percentage, percentage, percentage)"

Value Parameter Descriptions:

- **percentage**: either an integer value from 0 to 100 with "%" appended at the end or a float value ranging from 0 to 1
- **degree**: a value between 0 and 360 with the "deg" unit optionally appended to the end

Examples:

- "hsla(30, 0.6, 0.7, 80%)"
- "hsla(30deg, 40%, .5, 40%)"

Parameters
strdata (*str*) -- the hsla string to validate

Returns
A boolean determining whether the string fits the determined hsla schema

Return type
bool

pygame_gui.core.colour_parser.**is_valid_hsv_string**(*strdata: str*) → bool

Validate HSLA Color string in the format "hsv(degree, percentage, percentage)"

Value Parameter Descriptions:

- **percentage**: either an integer value from 0 to 100 with "%" appended at the end or a float value ranging from 0 to 1
- **degree**: a value between 0 and 360 with the "deg" unit optionally appended to the end

Examples:

- "hsv(50, 30%, .4)"
- "hsv(60deg, 40%, 12%)"

Parameters
strdata (*str*) -- the hsv string to validate

Returns
A boolean determining whether the string fits the determined hsv schema

Return type
bool

pygame_gui.core.colour_parser.**is_valid_hsva_string**(*strdata: str*) → bool

Validate HSVA Color string in the format "*hsva(degree, percentage, percentage, percentage)*"

Value Parameter Descriptions:

- **percentage**: either an integer value from 0 to 100 with "%" appended at the end or a float value ranging from 0 to 1
- **degree**: a value between 0 and 360 with the "deg" unit optionally appended to the end

Examples:

- "hsva(50, .3, 40%, .75)"
- "hsva(60deg, 40%, 12%, .94)"

Parameters
strdata (*str*) -- the hsva string to validate

Returns
A boolean determining whether the string fits the determined hsva schema or not

Return type
bool

pygame_gui.core.colour_parser.**is_valid_rgb_string**(*strdata: str*)

Validate RGB Color string in the format "rgb(uint8, uint8, uint8)"

Value Parameter Descriptions:

- **uint8**: is an integer value bounded between 0 and 255

Examples:

- "rgb(20, 40, 80)"

Parameters
strdata (*str*) -- the rgb string to validate

Returns
A boolean determining whether the string fits the determined rgb schema

Return type
bool

pygame_gui.core.colour_parser.**is_valid_rgba_string**(*strdata: str*) → bool

Validate RGBA Color string in the format "rgba(uint8, uint8, uint8, uint8)"

Value Parameter Descriptions:

- **uint8**: is an integer value bounded between 0 and 255

Examples:

- "rgba(30, 241, 174, 232)"

Parameters
strdata (*str*) -- the rgba string to validate

Returns
A boolean determining whether the string fits the determined rgba schema

Return type
bool

pygame_gui.core.colour_parser.**may_be_gradient_string**(*strdata: str*) → bool

Determine if a string should even be considered for validation as a gradient

Developer Notes:

- Determines this by determining if there are 2 or 3 commas outside any enclosing glyph and if the enclosing glyphs in the colour string are validly closed and opened.

Parameters
strdata (*str*) -- the possible gradient string to check

Returns
If the gradient string passes the starting check

Return type
bool

pygame_gui.core.colour_parser.**parse_cmy_string**(*strdata: str*) → Color

Parse CMY Color string in the format "cmy(percentage, percentage, percentage)"

Value Parameter Descriptions:

- **percentage**: either an integer value from 0 to 100 with "%" appended at the end or a float value ranging from 0 to 1

Examples:

- "cmy(.4, .7, 80%)"

Parameters
strdata (*str*) -- the cmy string to parse

Returns
A pygame Color from the cmy data parsed from strdata

Return type
pygame.Color

pygame_gui.core.colour_parser.**parse_colour_model**(*strdata: str*, *name: str*, *types: List[NumParserType]*) → List[T]

Use a colour model's name and types (generally denoted by its name in the _colourModelSchemas dictionary) to return a tuple of its containing values

Developer Notes:

- This function depends on the parser of each value type being present in the _valueParsers dictionary
- This function assumes that the strdata parameter has **already been validated** against the validate_colour_model function
- This function assumes that the colour model also takes up the form "name(value, value...)"

Parameters

- **strdata** (*str*) -- the colour model string to parse
- **name** (*str*) -- the name of the colour model
- **types** (*List[NumParserType]*) -- the types of the values in the colour model in order, used to parse each value individually

Returns
A tuple containing all the parsed values in the colour model

Return type
tuple

Raises

- **ValueError** -- if the colour model does not meet the standard <name(value, ...)> schema
- **KeyError** -- if the colour model cannot find the parser for a given type or the given type is not defined to have a parser

pygame_gui.core.colour_parser.**parse_colour_name**(*strdata: str*) → Color

Parse Colour name string into its corresponding colour value

As of the writing of these documentations, all colour names defined are from the CSS colours given by all major browsers, which can be found here: https://w3schools.sinsixx.com/css/css_colornames.asp.htm

All colour names are not case-sensitive, so RED, Red, and red all represent the same value

Parameters
strdata (*str*) -- The colour name to parse

Returns
A pygame Color from the colour name given by strdata

Return type
pygame.Color

Raises
KeyError -- If the colour corresponding to the strdata passed in could not be found (always call is_valid_colour_name first to check)

pygame_gui.core.colour_parser.**parse_colour_or_gradient_string**(*strdata: str*) → Color | *ColourGradient* | None

Parse a colour or gradient string into a pygame Color or a pygame_gui.core.colour_gradient.ColourGradient Object

The documentation for what counts as a 'Valid' colour and gradient string, including the supported colour formats by pygame_gui, can be found in the Theme Guide of the pygame_gui documentation at https://pygame-gui.readthedocs.io/en/v_065/theme_guide.html

Parameters
strdata (*str*) -- the string data to parse into a colour or gradient

Returns
The colour or gradient generated from the string data, or none

Return type
pygame.Color, *pygame_gui.core.colour_gradient.ColourGradient*, or None

pygame_gui.core.colour_parser.**parse_colour_string**(*strdata: str*) → Color | None

Parse a Color String

Developer Notes:

- This function uses the implemented colour parsing and colour validating functions available through _colourParsers in order to determine a proper colour data
- Additionally, named colour strings are taken into account firstly when determining the data of the colour
- Note that this function returns the first valid occurrence that they find

Parameters
strdata (*str*) -- The string to parse into a Colour

Returns
A Pygame Color Object from the colour data parsed from strdata or None if the strdata is an invalid colour string

Return type
pygame.Color or None

pygame_gui.core.colour_parser.**parse_gradient_string**(*strdata: str*) → *ColourGradient* | None

Parse a gradient string A gradient string should consist of a 3 or 4 length comma separated list, with the first values being any valid colour strings and the last value representing a degree angle for the direction of the gradient Examples: - "red,blue,40deg" - "#f23,rgb(30, 70, 230),hsv(50, 70%, 90%),50"

Parameters
strdata (*str*) -- the gradient string to validate

Returns
A pygame_gui.core.colour_gradient.ColourGradient object if strdata is a valid gradient string, otherwise None

Return type
bool or None

pygame_gui.core.colour_parser.**parse_hex_string**(*strdata: str*) → Color

Parse Hex Color string in the formats "#FFF", "#FFFF", "#FFFFFF", or "#FFFFFFFF"

Value Parameter Descriptions:

- **F**: any hexadecimal digit between 0 and F inclusive

Examples:

- "#A3F" (Shorthand)
- "#98C4" (Shorthand with Alpha)
- "#F3FAFF" (Full)
- "#BD24A017" (Full, with Alpha)

Parameters
strdata (*str*) -- the hex string to parse

Returns
A pygame Color from the hex data parsed from strdata

Return type
pygame.Color

pygame_gui.core.colour_parser.**parse_hsl_string**(*strdata: str*) → Color

Parse HSL Color string in the format "hsl(degree, percentage, percentage)"

Value Parameter Descriptions:

- **percentage**: either an integer value from 0 to 100 with "%" appended at the end or a float value ranging from 0 to 1
- **degree**: a value between 0 and 360 with the "deg" unit optionally appended to the end

Examples:

- "hsl(30, 0.6, 0.7)"
- "hsl(30deg, 40%, .5)"

Parameters
strdata (*str*) -- the hsl string to parse

Returns
A pygame Color from the hsl data parsed from strdata

Return type
pygame.Color

pygame_gui.core.colour_parser.**parse_hsla_string**(*strdata: str*) → Color

Parse HSLA Color string in the format "hsla(degree, percentage, percentage, percentage)"

Value Parameter Descriptions:

- **percentage**: either an integer value from 0 to 100 with "%" appended at the end or a float value ranging from 0 to 1
- **degree**: a value between 0 and 360 with the "deg" unit optionally appended to the end

Examples:

- "hsla(30, 0.6, 0.7, 80%)"
- "hsla(30deg, 40%, .5, 40%)"

Parameters
strdata (*str*) -- the hsla string to parse

Returns
A pygame Color from the hsla data parsed from strdata

Return type
pygame.Color

pygame_gui.core.colour_parser.**parse_hsv_string**(*strdata: str*) → Color

Parse HSV Color string in the format "hsv(degree, percentage, percentage)"

Value Parameter Descriptions:

- **percentage**: either an integer value from 0 to 100 with "%" appended at the end or a float value ranging from 0 to 1
- **degree**: a value between 0 and 360 with the "deg" unit optionally appended to the end

Examples:

- "hsv(50, 30%, .4)"
- "hsv(60deg, 40%, 12%)"

Parameters
strdata (*str*) -- the hsv string to parse

Returns
A pygame Color from the hsv data parsed from strdata

Return type
pygame.Color

pygame_gui.core.colour_parser.**parse_hsva_string**(*strdata: str*) → Color

Parse HSVA Color string in the format "hsva(degree, percentage, percentage, percentage)"

Value Parameter Descriptions:

- **percentage**: either an integer value from 0 to 100 with "%" appended at the end or a float value ranging from 0 to 1
- **degree**: a value between 0 and 360 with the "deg" unit optionally appended to the end

Examples:

- "hsva(50, .3, 40%, .75)"
- "hsva(60deg, 40%, 12%, .94)"

Parameters
strdata (*str*) -- the hsva string to parse

Returns
A pygame Color from the hsva data parsed from strdata

Return type
pygame.Color

pygame_gui.core.colour_parser.**parse_rgb_string**(*strdata: str*) → Color

Parse RGB Color string in the format "rgb(uint8, uint8, uint8)"

Value Parameter Descriptions:

- **uint8**: is an integer value bounded between 0 and 255

Examples:

- "rgb(20, 40, 80)"

Parameters
strdata (*str*) -- the rgb string to parse

Returns
A pygame Color from the data parsed from strdata

Return type
pygame.Color

pygame_gui.core.colour_parser.**parse_rgba_string**(*strdata: str*) → Color

Parse RGBA Color string in the format "rgba(uint8, uint8, uint8, uint8)"

Value Parameter Descriptions:

- **uint8**: is an integer value bounded between 0 and 255

Examples:

- "rgba(30, 241, 174, 232)"

Parameters
strdata (*str*) -- the rgba string to parse

Returns
A pygame Color from the rgba data parsed from strdata

Return type
pygame.Color

pygame_gui.core.colour_parser.**split_string_at_indices**(*strdata: str*, *indices: List[int] | Set[int] | Tuple[int]*) → List[str]

Works similarly to the built-in string split function, where the split data is discarded from the resultant array

Developer Notes:

- Used in colour_parser module to split gradient strings into their respective colour and angle components

Parameters

- **strdata** (*str*) -- the string to split
- **indices** (*Iterable[int]*) -- the indices to split the string at

Returns
A list of the split strings after cutting at the specified indices given by the 'indices' parameter

Return type
List[str]

pygame_gui.core.colour_parser.**valid_enclosing_glyphs**(*strdata: str*) → bool

Find if each opening parenthesis in a string has a valid closing parenthesis and vice versa

Developer Notes:

- Used to determine which top level commas should be used to separate gradients
- Used with the assumption that colour values themselves do not have glyphs that are left opened

Parameters
strdata (*str*) -- the string to check

Returns
A boolean determining whether each opening parenthesis has a closing parenthesis and each closing parenthesis has an open parenthesis

Return type
bool

pygame_gui.core.colour_parser.**validate_colour_model**(*strdata: str*, *name: str*, *types: List[NumParserType]*) → bool

Use a colour model's name and types (generally denoted by its name in the _colourModelSchemas dictionary) to validate its use

Developer Notes:

- This function depends on the validator of each value type being present in the _valueParsers dictionary
- This function assumes that the colour model also takes up the form "name(value, value...)"

Parameters

- **strdata** (*str*) -- the colour model string to validate
- **name** (*str*) -- the name of the colour model
- **types** (*List[NumParserType]*) -- the types of the values in the colour model in order, used to validate each value individually

Returns
A boolean on whether the colour model could find a working validator

Return type
bool

pygame_gui.core.gui_font_pygame module

class pygame_gui.core.gui_font_pygame.**GUIFontPygame**(*file: str | bytes | PathLike | IO | None*, *size: int | float*, *force_style: bool = False*, *style: Dict[str, bool] | None = None*)

Bases: *IGUIFontInterface*

get_direction() → int

Returns

get_metrics(*text: str*)

Parameters
text --

Returns

get_padding_height()

Returns

get_point_size()

get_rect(*text: str*) → Rect

Not sure if we want this. :return:

render_premul(*text: str*, *text_color: Color*) → Surface

Draws text to a surface ready for pre-multiplied alpha-blending

render_premul_to(*text: str*, *text_colour: Color*, *surf_size: Tuple[int, int]*, *surf_position: Tuple[int, int]*) → Surface

Parameters

- **text** --
- **text_colour** --
- **surf_size** --
- **surf_position** --

Returns

size(*text: str*) → Tuple[int, int]

Return the pixel size of a given text string in this font

Parameters
text -- the text to check.

Returns
the width & height in pixels.

property **underline**: bool

Returns

property **underline_adjustment**: float

Returns

pygame_gui.core.layered_gui_group module

class pygame_gui.core.layered_gui_group.**GUISprite**(**groups*)

Bases: object

A sprite class specifically designed for the GUI. Very similar to pygame's DirtySprite but without the Dirty flag.

add(**groups*)

add the sprite to groups

Parameters
groups -- sprite groups to add to.

Any number of Group instances can be passed as arguments. The Sprite will be added to the Groups it is not already a member of.

add_internal(*group*)

For adding this sprite to a group internally.

Parameters
group -- The group we are adding to.

alive()

does the sprite belong to any groups

Sprite.alive(): return bool

Returns True when the Sprite belongs to one or more Groups.

property blendmode

Layer property can only be set before the sprite is added to a group, after that it is read only and a sprite's layer in a group should be set via the group's change_layer() method.

Overwrites dynamic property from sprite class for speed.

groups()

list of Groups that contain this Sprite

Sprite.groups(): return group_list

Returns a list of all the Groups that contain this Sprite.

property image

Layer property can only be set before the sprite is added to a group, after that it is read only and a sprite's layer in a group should be set via the group's change_layer() method.

Overwrites dynamic property from sprite class for speed.

kill()

remove the Sprite from all Groups

Sprite.kill(): return None

The Sprite is removed from all the Groups that contain it. This won't change anything about the state of the Sprite. It is possible to continue to use the Sprite after this method has been called, including adding it to Groups.

property layer

Layer property can only be set before the sprite is added to a group, after that it is read only and a sprite's layer in a group should be set via the group's change_layer() method.

Overwrites dynamic property from sprite class for speed.

property rect

Layer property can only be set before the sprite is added to a group, after that it is read only and a sprite's layer in a group should be set via the group's change_layer() method.

Overwrites dynamic property from sprite class for speed.

remove(**groups*)

remove the sprite from groups

Parameters
groups -- sprite groups to remove from.

Any number of Group instances can be passed as arguments. The Sprite will be removed from the Groups it is currently a member of.

remove_internal(*group*)

For removing this sprite from a group internally.

Parameters
group -- The group we are removing from.

abstract update(*time_delta: float*)

A stub to override.

Parameters
time_delta -- the time passed in seconds between calls to this function.

property visible

You can make this sprite disappear without removing it from the group assign 0 for invisible and 1 for visible

class pygame_gui.core.layered_gui_group.**LayeredGUIGroup**(**sprites*)

Bases: LayeredUpdates

A sprite group specifically for the GUI. Similar to pygame's LayeredDirty group but with the dirty flag stuff removed for simplicity and speed. TODO: sever this entirely from LayeredUpdates at some point to fix the type hinting

add_internal(*sprite: GUISprite*, *layer=None*)

Do not use this method directly.

It is used by the group to add a sprite internally.

change_layer(*sprite: GUISprite*, *new_layer*)

change the layer of the sprite

LayeredUpdates.change_layer(sprite, new_layer): return None

The sprite must have been added to the renderer already. This is not checked.

draw(*surface*)

draw all sprites in the right order onto the given surface

remove_internal(*sprite*)

Do not use this method directly.

The group uses it to add a sprite.

update(**args*, ***kwargs*) → None

Parameters

- **args** --
- **kwargs** --

update_visibility()

Update the list of what is currently visible.

Called when we add or remove elements from the group or when an element is hidden or shown.

pygame_gui.core.object_id module

class pygame_gui.core.object_id.**ObjectID**(*object_id*, *class_id*)

Bases: tuple

class_id

Alias for field number 1

object_id

Alias for field number 0

pygame_gui.core.resource_loaders module

class pygame_gui.core.resource_loaders.**BlockingThreadedResourceLoader**

Bases: *ThreadedLoader*, *IResourceLoader*

This loader is designed to have it's update function called once, after which it will block the main thread until all it's assigned loading is complete.

update() → Tuple[bool, float]

Updates the load process. Blocks until it is completed.

Returns

A Boolean indicating whether the load has finished, and a float indicating the load's progress (between 0.0 and 1.0).

class pygame_gui.core.resource_loaders.**IResourceLoader**

Bases: object

Interface for a resource loader class. Resource loaders should inherit this interface.

abstract add_resource(*resource: FontResource | ImageResource | SurfaceResource*)

Adds a resource to be loaded.

Parameters

resource -- Either an ImageResource, SurfaceResource or a FontResource.

abstract start()

Kicks off the loading process. No more resources can be added to the loader at this point.

abstract started() → bool

Tells us if the loader has already begun or finished loading.

Returns
Returns True when it's too late to add anything to the load queues.

abstract update() → Tuple[bool, float]

Updates the load process.

Returns
A Boolean indicating whether the load has finished, and a float indicating the load's progress (between 0.0 and 1.0).

class pygame_gui.core.resource_loaders.**IncrementalThreadedResourceLoader**

Bases: *ThreadedLoader*, *IResourceLoader*

This loader is designed to have it's update function called repeatedly until it is finished.

It's useful if you want to display a loading progress bar for the UI - Though you will have to be careful not to use any assets that are still being loaded.

set_update_time_budget(*budget: float*)

Set the minimum amount of time to spend loading, per update loop.

Actual time spent may be somewhat over this budget as a long file load may start while we are within the budget.

NOTE: This only affects sequentially loading resources.

Parameters
budget -- A time budget in seconds. The default is 0.02 seconds.

update() → Tuple[bool, float]

Updates the load process will try to spend only as much time in here as allocated by the time budget.

Returns
A Boolean indicating whether the load has finished, and a float indicating the load's progress (between 0.0 and 1.0).

class pygame_gui.core.resource_loaders.**ThreadedLoader**

Bases: object

A loader that uses threads to try and load data faster.

Defaults to using five threads. Mess with it before starting the loader if you want to see if you can get any better loading performance with a different number.

add_resource(*resource: FontResource | ImageResource | SurfaceResource*)

Adds a resource to be loaded.

Currently Fonts & Images are loaded with threads. Surfaces load sequentially after the images are finished because they rely on their image being loaded and it is difficult to guarantee that with threads.

Parameters
resource -- Either an ImageResource, SurfaceResource or a FontResource.

set_finished()

Called when loading is done.

start()

Kicks off the loading process. No more resources can be added to the loader at this point.

started() → bool

Tells us if the loader has already begun or finished loading.

Returns

Returns True when it's too late to add anything to the load queues.

pygame_gui.core.surface_cache module

class pygame_gui.core.surface_cache.**SurfaceCache**

Bases: object

A cache for surfaces that we estimate the UI may want to reuse to save constantly remaking almost identical drawable shapes.

add_surface_to_cache(*surface: Surface*, *string_id: str*)

Adds a surface to the cache. There are two levels to the cache, the short term level just keeps hold of the surface until we have time to add it to the long term level.

Parameters

- **surface** -- The surface to add to the cache.
- **string_id** -- An ID to store the surface under to make it easy to recall later.

add_surface_to_long_term_cache(*cached_item: List[Surface | int]*, *string_id: str*)

Move a surface from the short term cache into the long term one.

Parameters

- **cached_item** -- The surface to move into the long term cache.
- **string_id** -- The ID of the surface in the cache.

static build_cache_id(*shape: str*, *size: Tuple[int, int]*, *shadow_width: int*, *border_width: int*, *border_colour: Color*, *bg_colour: Color*, *corner_radius: List[int] | None = None*) → str

Create an ID string for a surface based on it's dimensions and parameters. The idea is that any surface in the cache with the same values in this ID should be identical.

Parameters

- **shape** -- A string for the overall shape of the surface (rounded rectangle, rectangle, etc).
- **size** -- The dimensions of the surface.
- **shadow_width** -- The thickness of the shadow around the shape.
- **border_width** -- The thickness of the border around the shape.
- **border_colour** -- The colour of the border.
- **bg_colour** -- The background, or main colour of the surface.
- **corner_radius** -- Optional corner radius parameter, only used for rounded rectangles.

Returns

A assembled string ID from the provided data.

find_surface_in_cache(*lookup_id: str*) → Surface | None

Looks for a surface in the cache by an ID and returns it if found.

Parameters

lookup_id -- ID of the surface to look for in the cache.

:return The found surface, or None.

remove_user_and_request_clean_up_of_cached_item(*string_id: str*)

If we are certain that a cached surface won't be used again anytime soon we can request it is removed from the cache directly.

Parameters
string_id -- the ID of the cached surface to remove from the cache.

remove_user_from_cache_item(*string_id: str*)

Deduct a 'user' from a particular cache surface. The number of users of a cache surface over the lifetime of a program would be a decent measure of how 'valuable' it is to keep a surface in the cache.

Parameters
string_id -- The ID of the cached surface to deduct a user from.

static split_rect(*found_rectangle_to_split: Rect*, *dividing_rect: Rect*, *free_space_rectangles: List[Rect]*)

Takes an existing free space rectangle that we are placing a new surface inside of and then divides up the remaining space into new, smaller free space rectangles.

Parameters

- **found_rectangle_to_split** -- The rectangle we are splitting.
- **dividing_rect** -- The rectangle dividing up the split rectangle.
- **free_space_rectangles** -- A list of all free space rectangles for a particular surface.

update()

Takes care of steadily moving surfaces from the short term cache into the long term. Long term caching takes a while so we limit it to adding one surface a frame.

We also purge some lesser used surfaces from the long term cache when we run out of space.

pygame_gui.core.ui_appearance_theme module

class pygame_gui.core.ui_appearance_theme.**UIAppearanceTheme**(*resource_loader: IResourceLoader*, *locale: str*)

Bases: *IUIAppearanceThemeInterface*

The Appearance Theme class handles all the data that styles and generally dictates the appearance of UI elements across the whole UI.

The styling is split into four general areas:

- colours - spelled in the British English fashion with a 'u'.
- font - specifying a font to use for a UIElement where that is a relevant consideration.
- images - describing any images to be used in a UIElement.
- misc - covering all other types of data and stored as strings.

To change the theming for the UI you normally specify a theme file when creating the UIManager. For more information on theme files see the specific documentation elsewhere.

build_all_combined_ids(*element_base_ids: None | List[str | None]*, *element_ids: None | List[str]*, *class_ids: None | List[str | None]*, *object_ids: None | List[str | None]*) → List[str]

Construct a list of combined element ids from the element's various accumulated ids.

Parameters

- **element_base_ids** -- when an element is also another element (e.g. a file dialog is also a window)
- **element_ids** -- All the ids of elements this element is contained within.
- **class_ids** -- All the ids of 'classes' that this element is contained within.
- **object_ids** -- All the ids of objects this element is contained within.

Returns
A list of IDs that reference this element in order of decreasing specificity.

check_need_to_reload() → bool

Check if we need to reload our theme file because it's been modified. If so, trigger a reload and return True so that the UIManager can trigger elements to rebuild from the theme data.

Return bool
True if we need to reload elements because the theme data has changed.

get_colour(*colour_id: str*, *combined_element_ids: List[str] | None = None*) → Color

Uses data about a UI element and a specific ID to find a colour from our theme.

Parameters

- **combined_element_ids** -- A list of IDs representing an element's location in a hierarchy of elements.
- **colour_id** -- The id for the specific colour we are looking for.

Return pygame.Color
A pygame colour.

get_colour_or_gradient(*colour_id: str*, *combined_ids: List[str] | None = None*) → Color | *IColourGradientInterface*

Uses data about a UI element and a specific ID to find a colour, or a gradient, from our theme. Use this function if the UIElement can handle either type.

Parameters

- **combined_ids** -- A list of IDs representing an element's location in a hierarchy of elements.
- **colour_id** -- The id for the specific colour we are looking for.

Return pygame.Color or ColourGradient
A colour or a gradient object.

get_font(*combined_element_ids: List[str]*) → *IGUIFontInterface*

Uses some data about a UIElement to get a font object.

Parameters
combined_element_ids -- A list of IDs representing an element's location in an interleaved hierarchy of elements.

Return IGUIFontInterface
An interface to a pygame font object wrapper.

get_font_dictionary() → *IUIFontDictionaryInterface*

Lets us grab the font dictionary, which is created by the theme object, so we can access it directly.

Returns
The font dictionary.

get_font_info(*combined_element_ids: List[str]*) → Dict[str, Any]

Uses some data about a UIElement to get font data as dictionary

Parameters
combined_element_ids -- A list of IDs representing an element's location in an interleaved hierarchy of elements.

Return dictionary
Data about the font requested

get_image(*image_id: str*, *combined_element_ids: List[str]*) → Surface

Will raise an exception if no image with the ids specified is found. UI elements that have an optional image display will need to handle the exception.

Parameters

- **combined_element_ids** -- A list of IDs representing an element's location in a hierarchy of elements.
- **image_id** -- The id identifying the particular image spot in the UI we are looking for an image to add to.

Returns
A pygame.surface.Surface

get_misc_data(*misc_data_id: str*, *combined_element_ids: List[str]*) → str | Dict

Uses data about a UI element and a specific ID to try and find a piece of miscellaneous theming data. Raises an exception if it can't find the data requested, UI elements requesting optional data will need to handle this exception.

Parameters

- **combined_element_ids** -- A list of IDs representing an element's location in a hierarchy of elements.
- **misc_data_id** -- The id for the specific piece of miscellaneous data we are looking for.

Return Any
Returns a string or a Dict

load_theme(*file_path: str | PathLike | StringIO | PackageResource | dict*)

Loads a theme, and currently, all associated data like fonts and images required by the theme.

Parameters
file_path -- The location of the theme, or the theme data we want to load.

reload_theming()

We need to load our theme file to see if anything expensive has changed, if so trigger it to reload/rebuild.

set_locale(*locale: str*)

Set the locale used in the appearance theme.

Parameters
locale -- a two-letter ISO country code.

update_caching(*time_delta: float*)

Updates the various surface caches.

update_single_element_theming(*element_name: str*, *new_theming_data: str | dict*)

Update theming data, via string - for a single element. :param element_name: :param new_theming_data: :return:

update_theming(*new_theming_data: str | dict*, *rebuild_all: bool = True*)

Update theming data, via string - for the whole UI.

Parameters

- **new_theming_data** --
- **rebuild_all** --

Returns

pygame_gui.core.ui_container module

class pygame_gui.core.ui_container.**UIContainer**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *manager: IUIManagerInterface*, ***, *starting_height: int = 1*, *is_window_root_container: bool = False*, *container: IContainerLikeInterface | None = None*, *parent_element: UIElement | None = None*, *object_id: ObjectID | str | None = None*, *element_id: List[str] | None = None*, *anchors: Dict[str, str] | None = None*, *visible: int = 1*)

Bases: *UIElement*, *IUIContainerInterface*, *IContainerLikeInterface*

A UI Container holds any number of other UI elements inside a rectangle. When we move the UIContainer all the UI elements contained within it can be moved as well.

This class helps us make UI Windows, but likely will have wider uses as well as the GUI system develops.

Parameters

- **relative_rect** -- A pygame.Rect whose position is relative to whatever UIContainer it is inside, if any.
- **manager** -- The UIManager that manages this UIElement.
- **starting_height** -- The starting layer height for this element above its container.
- **is_window_root_container** -- True/False flag for whether this container is the root container for a UI window.
- **container** -- The UIContainer that this UIElement is contained within.
- **parent_element** -- The element this element 'belongs to' in the theming hierarchy.
- **object_id** -- A custom defined ID for fine-tuning of theming.
- **anchors** -- A dictionary describing what this element's relative_rect is relative to.
- **visible** -- Whether the container and its children are visible by default. Warning - it's parent container visibility may override this.

add_element(*element: IUIElementInterface*)

Add a UIElement to the container. The UI's relative_rect parameter will be relative to this container.

Parameters
element -- A UIElement to add to this container.

are_contents_hovered() → bool

Are any of the elements in the container hovered? Used for handling mousewheel events.

Returns
True if one of the elements is hovered, False otherwise.

calc_add_element_changes_thickness(*element:* IUIElementInterface)

This function checks if a single added element will increase the containers thickness and if so updates containers recursively.

Parameters
element -- the element to check.

change_layer(*new_layer: int*)

Change the layer of this container. Layers are used by the GUI to control the order in which things are drawn and which things should currently be interactive (so you can't interact with things behind other things).

This particular method is most often used to shift the visible contents of a window in front of any others when it is moved to the front of the window stack.

Parameters
new_layer -- The layer to move our container to.

check_hover(*time_delta: float*, *hovered_higher_element: bool*) → bool

A method that helps us to determine which, if any, UI Element is currently being hovered by the mouse.

Parameters

- **time_delta** -- A float, the time in seconds between the last call to this function and now (roughly).
- **hovered_higher_element** -- A boolean, representing whether we have already hovered a 'higher' element.

Returns
A boolean that is true if we have hovered a UI element, either just now or before this method.

clear()

Removes and kills all the UI elements inside this container.

disable()

Disables all elements in the container, so they are no longer interactive.

enable()

Enables all elements in the container, so they are interactive again.

expand_left(*width_increase: int*) → None

Increases the width of the container, but instead of expanding the right edge, it expands the left edge. This is achieved by setting the new dimensions and updating the anchors of all the elements anchored to the left of the container.

Parameters
width_increase -- The width to increase by. Pass in negative values to decrease the size

Returns
None

expand_top(*height_increase: int*) → None

Increases the height of the container, but instead of expanding the bottom edge, it expands the top edge. This is achieved by setting the new dimensions and updating the anchors of all the elements anchored to the top of the container.

Parameters
height_increase -- The height to increase by. Pass in negative values to decrease the size

Returns
None

get_container() → *IUIContainerInterface*

Implements the container interface. In this case we just return this since it is a container.

Returns
This container.

get_rect() → Rect

Access to the container's rect

Returns
a pygame rectangle

get_size() → Tuple[int, int]

Get the container's pixel size.

Returns
the pixel size as tuple [x, y]

get_thickness() → int

Get the container's layer thickness.

Returns
the thickness as an integer.

get_top_layer() → int

Assuming we have correctly calculated the 'thickness' of this container, this method will return the 'highest' layer in the LayeredDirty UI Group.

Returns
An integer representing the current highest layer being used by this container.

hide()

Hides the container, which means the container will not get drawn and will not process events. Should also hide all the children elements and containers. If the container was hidden before - ignore.

kill()

Overrides the standard kill method of UI Elements (and pygame sprites beyond that) to also call the kill method on all contained UI Elements.

on_contained_elements_changed(*target:* IUIElementInterface) → None

Update the positioning of the contained elements of this container. To be called when one of the contained elements may have moved, been resized or changed its anchors.

Parameters
target -- the UI element that has been benn moved resized or changed its anchors.

recalculate_container_layer_thickness()

This function will iterate through the elements in our container and determine the maximum 'height' that they reach in the 'layer stack'. We then use that to determine the overall 'thickness' of this container. The thickness value is used to determine where to place overlapping windows in the layers

remove_element(*element:* IUIElementInterface)

Remove a UIElement from this container.

Parameters
element -- A UIElement to remove from this container.

set_dimensions(*dimensions: Vector2 | Tuple[float, float], clamp_to_container: bool = False*)

Set the dimension of this container and update the positions of elements within it accordingly.

Parameters

- **dimensions** -- the new dimensions.
- **clamp_to_container** -- Whether we should clamp the dimensions to the dimensions of the container or not.

set_position(*position: Vector2 | Tuple[float, float]*)

Set the absolute position of this container - it is usually less chaotic to deal with setting relative positions.

Parameters
position -- the new absolute position to set.

set_relative_position(*position: Vector2 | Tuple[float, float]*)

Set the position of this container, relative to the container it is within.

Parameters
position -- the new relative position to set.

show()

Shows the container, which means the container will get drawn and will process events. Should also show all the children elements and containers. If the container was visible before - ignore.

update_containing_rect_position()

This function is called when we move the container to update all the contained UI Elements to move as well.

pygame_gui.core.ui_element module

class pygame_gui.core.ui_element.**UIElement**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *manager:* IUIManagerInterface | *None*, *container:* IContainerLikeInterface | *None*, *, *starting_height: int*, *layer_thickness: int*, *anchors: Dict[str, str | IUIElementInterface]* | *None = None*, *visible: int = 1*, *parent_element:* IUIElementInterface | *None = None*, *object_id:* ObjectID | *str* | *None = None*, *element_id: List[str]* | *None = None*, *ignore_shadow_for_initial_size_and_pos: bool = False*)

Bases: *GUISprite*, *IUIElementInterface*

A base class for UI elements. You shouldn't create UI Element objects, instead all UI Element classes should derive from this class. Inherits from pygame.sprite.Sprite.

Parameters

- **relative_rect** -- A rectangle shape of the UI element, the position is relative to the element's container.
- **manager** -- The UIManager that manages this UIElement.
- **container** -- A container that this element is contained in.
- **starting_height** -- Used to record how many layers above its container this element should be. Normally 1.
- **layer_thickness** -- Used to record how 'thick' this element is in layers. Normally 1.
- **anchors** -- A dictionary describing what this element's relative_rect is relative to.

- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.
- **parent_element** -- Element that this element 'belongs to' in theming. Elements inherit colours from parents.
- **object_id** -- An optional set of IDs to help distinguish this element from other elements.
- **element_id** -- A list of string ID representing this element's class.

can_hover() → bool

A stub method to override. Called to test if this method can be hovered.

change_layer(*new_layer: int*)

Changes the layer this element is on.

Parameters
new_layer -- The layer to change this element to.

change_object_id(*new_object_id:* ObjectID | *str* | *None*)

Allows for easy switching of an element's ObjectID used for theming and events.

Will rebuild the element after switching the ID

Parameters
new_object_id -- The new ID to use for this element.

Returns

check_hover(*time_delta: float*, *hovered_higher_element: bool*) → bool

A method that helps us to determine which, if any, UI Element is currently being hovered by the mouse.

Parameters

- **time_delta** -- A float, the time in seconds between the last call to this function and now (roughly).
- **hovered_higher_element** -- A boolean, representing whether we have already hovered a 'higher' element.

Return bool
A boolean that is true if we have hovered a UI element, either just now or before this method.

disable()

Disables elements so they are no longer interactive. This is just a default fallback implementation for elements that don't define their own.

Elements should handle their own enabling and disabling.

enable()

Enables elements so they are interactive again. This is just a default fallback implementation for elements that don't define their own.

Elements should handle their own enabling and disabling.

focus()

A stub to override. Called when we focus this UI element.

get_abs_rect() → Rect

The absolute positioning rect.

Returns
A pygame rect.

`get_anchor_targets()` → list

Get any anchor targets this element has, so we can update them when their targets change :return: the list of anchor targets.

`get_anchors()` → Dict[str, str | *IUIElementInterface*]

A dictionary containing all the anchors defining what the relative rect is relative to

Returns
A dictionary containing all the anchors defining what the relative rect is relative to

`get_class_ids()` → List[str]

A list of all the class IDs in this element's theming/event hierarchy.

Returns
a list of strings, one for each element in the hierarchy.

`get_element_base_ids()` → List[str]

A list of all the element base IDs in this element's theming/event hierarchy.

Returns
a list of strings, one for each element in the hierarchy.

`get_element_ids()` → List[str]

A list of all the element IDs in this element's theming/event hierarchy.

Returns
a list of strings, one for each element in the hierarchy.

`get_focus_set()` → Set[*IUIElementInterface*]

Return the set of elements to focus when we focus this element.

`get_image_clipping_rect()` → Rect | None

Obtain the current image clipping rect.

Returns
The current clipping rect. Maybe None.

`get_object_ids()` → List[str]

A list of all the object IDs in this element's theming/event hierarchy.

Returns
a list of strings, one for each element in the hierarchy.

`get_relative_rect()` → Rect

The relative positioning rect.

Returns
A pygame rect.

`get_starting_height()` → int

Get the starting layer height of this element. (i.e. the layer we start placing it on *above* its container, it may use more layers above this layer)

Returns
an integer representing the starting layer height.

`get_top_layer()` → int

Assuming we have correctly calculated the 'thickness' of it, this method will return the top of this element.

Return int
An integer representing the current highest layer being used by this element.

`hide`()

Hides the widget, which means the widget will not get drawn and will not process events. Clear hovered state.

`hover_point`(*hover_x: float*, *hover_y: float*) → bool

Test if a given point counts as 'hovering' this UI element. Normally that is a straightforward matter of seeing if a point is inside the rectangle. Occasionally it will also check if we are in a wider zone around a UI element once it is already active, this makes it easier to move scroll bars and the like.

Parameters

- **hover_x** -- The x (horizontal) position of the point.
- **hover_y** -- The y (vertical) position of the point.

Returns

Returns True if we are hovering this element.

`property hovered: bool`

Are we hovering over this element with the mouse pointer or other input highlighting method.

Returns

True if hovered.

`join_focus_sets`(*element:* IUIElementInterface)

Join this element's focus set with another element's focus set.

Parameters

element -- The other element whose focus set we are joining with.

`kill`()

Overriding regular sprite kill() method to remove the element from its container.

`on_fresh_drawable_shape_ready`()

Called when our drawable shape has finished rebuilding the active surface. This is needed because sometimes we defer rebuilding until a more advantageous (read quieter) moment.

`on_hovered`()

Called when this UI element first enters the 'hovered' state.

`on_locale_changed`()

Called for each element when the locale is changed on their UIManager

`on_unhovered`()

Called when this UI element leaves the 'hovered' state.

`process_event`(*event: Event*) → bool

A stub to override. Gives UI Elements access to pygame events.

Parameters

event -- The event to process.

Returns

Should return True if this element makes use of this event.

`rebuild`()

Takes care of rebuilding this element. Most derived elements are going to override this, and hopefully call the super() class method.

rebuild_from_changed_theme_data()

A stub to override when we want to rebuild from theme data.

remove_element_from_focus_set(*element*)

remove an element from this sets focus group.

Parameters
element -- The element to remove.

set_anchors(*anchors: Dict[str, str | IUIElementInterface] | None*) → None

Wraps the setting of the anchors with some validation

Parameters
anchors -- A dictionary of anchors defining what the relative rect is relative to

Returns
None

set_container(*container: None | IContainerLikeInterface*)

Switch the element to new container. Remove the element from the old container and add it to the new container.

Parameters
container -- The new container to add.

set_dimensions(*dimensions: Vector2 | Tuple[float, float]*, *clamp_to_container: bool = False*)

Method to directly set the dimensions of an element. And set whether the elements are dynamic.

NOTE: Using this on elements inside containers with non-default anchoring arrangements may make a mess of them.

Parameters

- **dimensions** -- The new dimensions to set. If it is a negative value, the element will become dynamically sized, otherwise it will become statically sized.
- **clamp_to_container** -- Whether we should clamp the dimensions to the dimensions of the container or not.

set_focus_set(*focus_set: Set[IUIElementInterface] | None*)

Set the focus set to a specific set of elements.

Parameters
focus_set -- The focus set to set.

set_image(*new_image: Surface | None*)

This used to be the way to set the proper way to set the .image property of a UIElement (inherited from pygame.Sprite), but it is intended for internal use in the library - not for adding actual images/pictures on UIElements. As such I've renamed the original function to make it protected and not part of the interface and deprecated this one for most elements.

Returns

set_minimum_dimensions(*dimensions: Vector2 | Tuple[float, float]*)

If this window is resizable, then the dimensions we set here will be the minimum that users can change the window to. They are also used as the minimum size when 'set_dimensions' is called.

Parameters
dimensions -- The new minimum dimension for the window.

set_position(*position: Vector2 | Tuple[float, float]*)

Method to directly set the absolute screen rect position of an element.

Parameters
position -- The new position to set.

set_relative_position(*position: Vector2 | Tuple[float, float]*)

Method to directly set the relative rect position of an element.

Parameters
position -- The new position to set.

set_visual_debug_mode(*activate_mode: bool*)

Enables a debug mode for the element which displays layer information on top of it in a tiny font.

Parameters
activate_mode -- True or False to enable or disable the mode.

show()

Shows the widget, which means the widget will get drawn and will process events.

unfocus()

A stub to override. Called when we stop focusing this UI element.

update(*time_delta: float*)

Updates this element's drawable shape, if it has one.

Parameters
time_delta -- The time passed between frames, measured in seconds.

update_containing_rect_position()

Updates the position of this element based on the position of its container. Usually called when the container has moved.

update_theming(*new_theming_data: str*)

Update the theming for this element using the most specific ID assigned to it.

If you have not given this element a unique ID, this function will also update the theming of other elements of this theming class or of this element type.

Parameters
new_theming_data -- the new theming data in a json string

while_hovering(*time_delta: float*, *mouse_pos: Vector2 | Tuple[int, int] | Tuple[float, float]*)

Called while we are in the hover state. It will create a tool tip if we've been in the hover state for a while, the text exists to create one, and we haven't created one already.

Parameters

- **time_delta** -- Time in seconds between calls to update.
- **mouse_pos** -- The current position of the mouse.

pygame_gui.core.ui_font_dictionary module

class pygame_gui.core.ui_font_dictionary.**DefaultFontData**(*size: int*, *name: str*, *style: str*, *regular_file_name: str*, *bold_file_name: str*, *italic_file_name: str*, *bold_italic_file_name: str*, *script: str = 'Latn'*, *direction: int = 0*)

Bases: object

Data class to wrap up all the data for a default font. Used now that we have multiple default fonts for different locales.

class pygame_gui.core.ui_font_dictionary.**UIFontDictionary**(*resource_loader: IResourceLoader*, *locale: str*)

Bases: *IUIFontDictionaryInterface*

The font dictionary is used to store all the fonts that have been loaded into the UI system.

add_font_path(*font_name: str*, *font_path: str | PackageResource*, *bold_path: str | PackageResource | None = None*, *italic_path: str | PackageResource | None = None*, *bold_italic_path: str | PackageResource | None = None*)

Adds paths to different font files for a font name.

Parameters

- **font_name** -- The name to assign to these font files.
- **font_path** -- The path to the font's file with no particular style.
- **bold_path** -- The path to the font's file with a bold style.
- **italic_path** -- The path to the font's file with an italic style.
- **bold_italic_path** -- The path to the font's file with a bold and an italic style.

check_font_preloaded(*font_id: str*) → bool

Check if a font is already preloaded or not.

Parameters
font_id -- The ID of the font to check for

Returns
True or False.

convert_html_to_point_size(*html_size: float*) → int

Takes in an HTML style font size and converts it into a point font size.

Parameters
html_size -- Size in HTML style.

Return int
A 'point' font size.

create_font_id(*font_size: int*, *font_name: str*, *bold: bool*, *italic: bool*, *antialiased: bool = True*) → str

Create an id for a particularly styled and sized font from those characteristics.

Parameters

- **font_size** -- The size of the font.
- **font_name** -- The name of the font.

- **bold** -- Whether the font is bold styled or not.
- **italic** -- Whether the font is italic styled or not.
- **antialiased** -- Whether the font is antialiased or not.

Return str
The finished font id.

ensure_debug_font_loaded()

Ensure the font we use for debugging purposes is loaded. Generally called after we start a debugging mode.

find_font(*font_size: int*, *font_name: str*, *bold: bool = False*, *italic: bool = False*, *antialiased: bool = True*, *script: str = 'Latn'*, *direction: int = 0*) → *IGUIFontInterface*

Find a loaded font from the font dictionary. Will load a font if it does not already exist, and we have paths to the needed files, however it will issue a warning after doing so because dynamic file loading is normally a bad idea as you will get frame rate hitches while the running program waits for the font to load.

Instead, it's best to preload all your needed files at another time in your program when you have more control over the user experience.

Parameters

- **font_size** -- The size of the font to find.
- **font_name** -- The name of the font to find.
- **bold** -- Whether the font is bold or not.
- **italic** -- Whether the font is italic or not.
- **antialiased** -- Whether the font is antialiased or not.
- **script** -- The ISO 15924 script code used for text shaping as a string.
- **direction** -- the direction of text e.g. left to right or right to left. An integer.

Return IGUIFontInterface
Returns either the font we asked for, or the default font.

find_font_resource(*font_size: int*, *font_name: str*, *bold: bool = False*, *italic: bool = False*, *antialiased: bool = True*, *script: str = 'Latn'*, *direction: int = 0*) → FontResource

Find a loaded font resource from the font dictionary. Will load a font if it does not already exist, and we have paths to the needed files, however it will issue a warning after doing so because dynamic file loading is normally a bad idea as you will get frame rate hitches while the running program waits for the font to load.

Instead, it's best to preload all your needed files at another time in your program when you have more control over the user experience.

Parameters

- **font_size** -- The size of the font to find.
- **font_name** -- The name of the font to find.
- **bold** -- Whether the font is bold or not.
- **italic** -- Whether the font is italic or not.
- **antialiased** -- Whether the font is antialiased or not.
- **script** -- The ISO 15924 script code used for text shaping as a string.
- **direction** -- the direction of text e.g. left to right or right to left. An integer.

Return FontResource
Returns either the font resource we asked for, or the default font.

`get_default_font`() → *IGUIFontInterface*

Grab the default font.

Returns
The default font.

`preload_font`(*font_size: int*, *font_name: str*, *bold: bool = False*, *italic: bool = False*, *force_immediate_load: bool = False*, *antialiased: bool = True*, *script: str = 'Latn'*, *direction: int = 0*)

Lets us load a font at a particular size and style before we use it. While you can get away with relying on dynamic font loading during development, it is better to eventually preload all your font data at a controlled time, which is where this method comes in.

Parameters

- **font_size** -- The size of the font to load.
- **font_name** -- The name of the font to load.
- **bold** -- Whether the font is bold styled or not.
- **italic** -- Whether the font is italic styled or not.
- **force_immediate_load** -- resource loading setup to immediately load the font on the main thread.
- **antialiased** -- Whether the font is antialiased or not.
- **script** -- The ISO 15924 script code used for text shaping as a string.
- **direction** -- the direction of text e.g. left to right or right to left. An integer.

`print_unused_loaded_fonts`()

Can be called to check if the UI is loading any fonts that we haven't used by the point this function is called. If a font is truly unused then we can remove it from our loading and potentially speed up the overall loading of the program.

This is not a foolproof check because this function could easily be called before we have explored all the code paths in a project that may use fonts.

`set_locale`(*new_locale: str*)

This may change the default font.

Parameters
new_locale -- The new locale to set, a two-letter country code ISO 639-1

pygame_gui.core.ui_shadow module

`class pygame_gui.core.ui_shadow.ShadowGenerator`

Bases: `object`

A class to generate surfaces that work as a 'shadow' for rectangular UI elements. Base shadow surface are generated with an algorithm, then when one is requested at a specific size the closest pre-generated shadow surface is picked and then scaled to the exact size requested.

By default it creates a four base shadows in a small range of sizes. If you find the shadow appearance unsatisfactory then it is possible to create more closer to the size of the elements you are having trouble with.

clear_short_term_caches()

Empties short term caches so we aren't hanging on to so many surfaces.

create_new_ellipse_shadow(*width: int*, *height: int*, *shadow_width_param: int*, *aa_amount: int = 4*) → Surface

Creates a ellipse shaped shadow surface at the specified size and stores it for later use.

Parameters

- **width** -- The width of the shadow to create.
- **height** -- The height of the shadow to create.
- **shadow_width_param** -- The width of the shadowed edge.
- **aa_amount** -- The amount of anti-aliasing to use, defaults to 4.

create_new_rectangle_shadow(*width: int*, *height: int*, *shadow_width_param: int*, *corner_radii: List[int]*) → Surface | None

Creates a rectangular shadow surface at the specified size and stores it for later use.

Parameters

- **width** -- The width of the base shadow to create.
- **height** -- The height of the base shadow to create.
- **shadow_width_param** -- The width of the shadowed edge.
- **corner_radii** -- The radius of the rectangular shadow's corners.

create_shadow_corners(*shadow_width_param: int*, *corner_radii: List[int]*, *aa_amount=4*) → Dict[str, Surface]

Create corners for our rectangular shadows. These can be used across many sizes of shadow with the same shadow width and corner radius.

Parameters

- **shadow_width_param** -- Width of the shadow.
- **corner_radii** -- Corner radii of the shadow.
- **aa_amount** -- Antialiasing amount. Defaults to 4x.

find_closest_shadow_scale_to_size(*size: Tuple[int, int]*, *shadow_width: int = 2*, *shape: str = 'rectangle'*, *corner_radius: List[int] | None = None*) → Surface | None

This function searches through our dictionary of created shadows, grabs the closest one to the size we request and then scales that shadow to the exact size we need.

Parameters

- **size** -- The size of the element we are finding a shadow for.
- **shadow_width** -- The width of the shadow to find.
- **shape** -- The shape of the shadow to find.
- **corner_radius** -- The radius of the corners if this is a rectangular shadow.

Returns

The shadow surface we asked for scaled to the size we requested, or None if no shadows exist.

pygame_gui.core.ui_window_stack module

class pygame_gui.core.ui_window_stack.**UIWindowStack**(*window_resolution: Tuple[int, int]*, *root_container:* IUIContainerInterface)

Bases: *IUIWindowStackInterface*

A class for managing a stack of GUI windows so that only one is 'in front' at a time and the rest are sorted based on the last time they were interacted with/created.

Parameters

- **window_resolution** -- The resolution of the OS window.
- **root_container** -- The root container for the whole UI.

add_new_window(*window:* IWindowInterface)

Adds a new window to the top of the stack.

Parameters
window -- The window to add.

clear()

Empties the whole stack removing and killing all windows.

get_full_stack() → List[*IWindowInterface*]

Returns the full stack of normal and always on top windows.

Returns
a list of Windows

is_window_at_top(*window:* IWindowInterface) → bool

Checks if a window is at the top of the normal window stack or not.

Parameters
window -- The window to check.

Returns
returns True if this window is at the top of the stack.

is_window_at_top_of_top(*window:* IWindowInterface) → bool

Checks if a window is at the top of the top window stack or not.

Parameters
window -- The window to check.

Returns
returns True if this window is at the top of the stack.

move_window_to_front(*window_to_front:* IWindowInterface)

Moves the passed in window to the top of its stack and resorts the other windows to deal with the change.

Parameters
window_to_front -- the window to move to the front.

remove_window(*window_to_remove:* IWindowInterface)

Removes a window from the stack and resorts the remaining windows to adjust for its absence.

Parameters
window_to_remove -- the window to remove.

Module contents

class pygame_gui.core.**BlockingThreadedResourceLoader**

Bases: *ThreadedLoader*, *IResourceLoader*

This loader is designed to have it's update function called once, after which it will block the main thread until all it's assigned loading is complete.

update() → Tuple[bool, float]

Updates the load process. Blocks until it is completed.

Returns

A Boolean indicating whether the load has finished, and a float indicating the load's progress (between 0.0 and 1.0).

class pygame_gui.core.**ColourGradient**(*angle_direction: int*, *colour_1: Color*, *colour_2: Color*, *colour_3: Color | None = None*)

Bases: *IColourGradientInterface*

Creates a small surface containing a smooth gradient between two or three colours.

Parameters

- **angle_direction** -- Angle direction of the gradient in degrees.
- **colour_1** -- The first colour of the gradient.
- **colour_2** -- The second colour of the gradient.
- **colour_3** -- An optional third colour for the gradient.

apply_gradient_to_surface(*input_surface: Surface*, *rect: Rect | None = None*)

Applies this gradient to a specified input surface using blending multiplication. As a result this method works best when the input surface is a mostly white, stencil shape type surface.

Parameters

- **input_surface** --
- **rect** -- The rectangle on the surface to apply the gradient to. If None, applies to the whole surface.

class pygame_gui.core.**IContainerLikeInterface**

Bases: object

A metaclass that defines the interface for containers used by elements.

This interface lets us treat classes like UIWindows and UIPanels like containers for elements even though they actually pass this functionality off to the proper UIContainer class.

are_contents_hovered() → bool

Are any of the elements in the container hovered? Used for handling mousewheel events.

Returns

True if one of the elements is hovered, False otherwise.

abstract get_container() → *IUIContainerInterface*

Gets an actual container from this container-like UI element.

abstract hide()

Hides the container, which means the container will not get drawn and will not process events. Should also hide all the children elements. If the container was hidden before - ignore.

abstract show()

Shows the container, which means the container will get drawn and will process events. Should also show all the children elements. If the container was visible before - ignore.

class pygame_gui.core.**IWindowInterface**

Bases: object

A metaclass that defines the interface that the window stack uses to interface with the UIWindow class.

Interfaces like this help us evade cyclical import problems by allowing us to define the actual window class later on and have it make use of the window stack.

abstract property always_on_top: bool

Whether the window is always above normal windows or not. :return:

abstract can_hover() → bool

Called to test if this window can be hovered.

abstract change_layer(*layer: int*)

Change the drawing layer of this window.

Parameters
layer -- the new layer to move to.

abstract check_clicked_inside_or_blocking(*event: Event*) → bool

A quick event check outside the normal event processing so that this window is brought to the front of the window stack if we click on any of the elements contained within it.

Parameters
event -- The event to check.

Returns
returns True if the event represents a click inside this window or the window is blocking.

abstract check_hover(*time_delta: float*, *hovered_higher_element: bool*) → bool

For the window the only hovering we care about is the edges if this is a resizable window.

Parameters

- **time_delta** -- time passed in seconds between one call to this method and the next.
- **hovered_higher_element** -- Have we already hovered an element/window above this one?

abstract get_hovering_edge_id() → str

Gets the ID of the combination of edges we are hovering for use by the cursor system.

Returns
a string containing the edge combination ID (e.g. xy,yx,xl,xr,yt,yb)

get_layer_thickness() → int

The layer 'thickness' of this window/ :return: an integer

abstract get_top_layer() → int

Returns the 'highest' layer used by this window so that we can correctly place other windows on top of it.

Returns
The top layer for this window as a number (greater numbers are higher layers).

abstract kill()

Overrides the basic kill() method of a pygame sprite so that we also kill all the UI elements in this window, and remove if from the window stack.

abstract property layer: int

The layer of this window (read-only)

abstract on_moved_to_front()

Called when a window is moved to the front of the stack.

abstract process_event(*event: Event*) → bool

Handles resizing & closing windows. Gives UI Windows access to pygame events. Derived windows should super() call this class if they implement their own process_event method.

Parameters
event -- The event to process.

Return bool
Return True if this element should consume this event and not pass it to the rest of the UI.

abstract rebuild()

Rebuilds the window when the theme has changed.

abstract rebuild_from_changed_theme_data()

Called by the UIManager to check the theming data and rebuild whatever needs rebuilding for this element when the theme data has changed.

abstract set_blocking(*state: bool*)

Sets whether this window being open should block clicks to the rest of the UI or not. Defaults to False.

Parameters
state -- True if this window should block mouse clicks.

abstract set_dimensions(*dimensions: Vector2 | Tuple[float, float]*)

Set the size of this window and then re-sizes and shifts the contents of the windows container to fit the new size.

Parameters
dimensions -- The new dimensions to set.

abstract set_display_title(*new_title: str*)

Set the title of the window.

Parameters
new_title -- The title to set.

abstract set_minimum_dimensions(*dimensions: Vector2 | Tuple[float, float]*)

If this window is resizable, then the dimensions we set here will be the minimum that users can change the window to. They are also used as the minimum size when 'set_dimensions' is called.

Parameters
dimensions -- The new minimum dimension for the window.

abstract set_position(*position: Vector2 | Tuple[float, float]*)

Method to directly set the absolute screen rect position of an element.

Parameters
position -- The new position to set.

abstract set_relative_position(*position: Vector2 | Tuple[float, float]*)

Method to directly set the relative rect position of an element.

Parameters
position -- The new position to set.

abstract should_use_window_edge_resize_cursor() → bool

Returns true if this window is in a state where we should display one of the resizing cursors

Returns
True if a resizing cursor is needed.

abstract update(*time_delta: float*)

A method called every update cycle of our application. Designed to be overridden by derived classes but also has a little functionality to make sure the window's layer 'thickness' is accurate and to handle window resizing.

Parameters
time_delta -- time passed in seconds between one call to this method and the next.

class pygame_gui.core.**IncrementalThreadedResourceLoader**

Bases: *ThreadedLoader*, *IResourceLoader*

This loader is designed to have it's update function called repeatedly until it is finished.

It's useful if you want to display a loading progress bar for the UI - Though you will have to be careful not to use any assets that are still being loaded.

set_update_time_budget(*budget: float*)

Set the minimum amount of time to spend loading, per update loop.

Actual time spent may be somewhat over this budget as a long file load may start while we are within the budget.

NOTE: This only affects sequentially loading resources.

Parameters
budget -- A time budget in seconds. The default is 0.02 seconds.

update() → Tuple[bool, float]

Updates the load process will try to spend only as much time in here as allocated by the time budget.

Returns
A Boolean indicating whether the load has finished, and a float indicating the load's progress (between 0.0 and 1.0).

class pygame_gui.core.**ObjectID**(*object_id*, *class_id*)

Bases: tuple

class_id

Alias for field number 1

object_id

Alias for field number 0

class pygame_gui.core.**ShadowGenerator**

Bases: object

A class to generate surfaces that work as a 'shadow' for rectangular UI elements. Base shadow surface are generated with an algorithm, then when one is requested at a specific size the closest pre-generated shadow surface is picked and then scaled to the exact size requested.

By default it creates a four base shadows in a small range of sizes. If you find the shadow appearance unsatisfactory then it is possible to create more closer to the size of the elements you are having trouble with.

clear_short_term_caches()

Empties short term caches so we aren't hanging on to so many surfaces.

create_new_ellipse_shadow(*width: int*, *height: int*, *shadow_width_param: int*, *aa_amount: int = 4*) → Surface

Creates a ellipse shaped shadow surface at the specified size and stores it for later use.

Parameters

- **width** -- The width of the shadow to create.
- **height** -- The height of the shadow to create.
- **shadow_width_param** -- The width of the shadowed edge.
- **aa_amount** -- The amount of anti-aliasing to use, defaults to 4.

create_new_rectangle_shadow(*width: int*, *height: int*, *shadow_width_param: int*, *corner_radii: List[int]*) → Surface | None

Creates a rectangular shadow surface at the specified size and stores it for later use.

Parameters

- **width** -- The width of the base shadow to create.
- **height** -- The height of the base shadow to create.
- **shadow_width_param** -- The width of the shadowed edge.
- **corner_radii** -- The radius of the rectangular shadow's corners.

create_shadow_corners(*shadow_width_param: int*, *corner_radii: List[int]*, *aa_amount=4*) → Dict[str, Surface]

Create corners for our rectangular shadows. These can be used across many sizes of shadow with the same shadow width and corner radius.

Parameters

- **shadow_width_param** -- Width of the shadow.
- **corner_radii** -- Corner radii of the shadow.
- **aa_amount** -- Antialiasing amount. Defaults to 4x.

find_closest_shadow_scale_to_size(*size: Tuple[int, int]*, *shadow_width: int = 2*, *shape: str = 'rectangle'*, *corner_radius: List[int] | None = None*) → Surface | None

This function searches through our dictionary of created shadows, grabs the closest one to the size we request and then scales that shadow to the exact size we need.

Parameters

- **size** -- The size of the element we are finding a shadow for.
- **shadow_width** -- The width of the shadow to find.
- **shape** -- The shape of the shadow to find.
- **corner_radius** -- The radius of the corners if this is a rectangular shadow.

Returns

The shadow surface we asked for scaled to the size we requested, or None if no shadows exist.

class pygame_gui.core.**TextBoxLayout**(*input_data_queue: Deque[TextLayoutRect]*, *layout_rect: Rect*, *view_rect: Rect*, *line_spacing: float*, *default_font_data: Dict[str, Any]*, *allow_split_dashes: bool = True*, *text_direction: int = 0*, *text_x_scroll_enabled: bool = False*, *editable: bool = False*)

Bases: object

Class to layout multiple lines of text to fit in a defined column.

The base of the 'column' rectangle is set once the data supplied has been laid out to fit in the width provided.

add_chunks_to_hover_group(*link_hover_chunks: List[TextLayoutRect]*)

Pass in a list of layout rectangles to add to a hover-able group. Usually used for hyperlinks.

Parameters
link_hover_chunks --

align_left_all_rows(*x_padding*)

Align all rows to the left hand side of the layout.

Parameters
x_padding -- the amount of padding to insert to on the left before the text starts.

align_right_all_rows(*x_padding*)

Align all rows to the right hand side of the layout.

Parameters
x_padding -- the amount of padding to insert to on the right before the text starts.

append_layout_rects(*new_queue*)

Add some LayoutRects on to the end of the current layout. This should be relatively fast as we don't have to rejig everything before the additions, and some of the time don't need to redraw everything either.

This is new so there may still be some bugs to iron out.

Parameters
new_queue --

backspace_at_cursor()

Deletes a single character behind the edit cursor. Mimics a standard word processor 'backspace' operation.

blit_finalised_text_to_surf(*surface: Surface*)

Lets us blit a finalised text surface to an arbitrary surface. Useful for doing stuff with text effects.

Parameters
surface -- the target surface to blit onto.

clear_effects()

Clear text layout level text effect parameters.

clear_final_surface()

Clears the finalised surface.

delete_at_cursor()

Deletes a single character in front of the edit cursor. Mimics a standard word processor 'delete' operation.

delete_selected_text()

Delete the currently selected text.

finalise_to_new()

Finalises our layout to a brand-new surface that this method creates.

finalise_to_surf(*surface: Surface*)

Take this layout, with everything positioned in the correct place and finalise it to a surface.

May be called again after changes to the layout? Update surf?

Parameters
surface -- The surface we are going to blit the contents of this layout onto.

find_cursor_position_from_click_pos(*click_pos*) → int

Find an edit text cursor position in the text from a click.

Here we don't set it, we just find it and return it.

Parameters
click_pos -- This is the pixel position we want to find the nearest cursor spot to.

Returns
an integer representing the character index position in the text

get_cursor_colour() → Color

Get the current colour of the editing carat/text cursor.

Returns
a pygame.Color object containing the current colour.

get_cursor_index()

Get the current character index, in the text layout's text, of the current edit cursor position.

Essentially the reverse of 'set_cursor_position()'.

get_cursor_pos_move_down_one_row(*last_cursor_horiz_index*)

Returns a cursor character position in the row directly above the current cursor position if possible.

get_cursor_pos_move_up_one_row(*last_cursor_horiz_index*)

Returns a cursor character position in the row directly above the last horizontal cursor position if possible.

horiz_center_all_rows(*method='rect'*)

Horizontally center all rows of text in the layout. This uses 'rectangular' centering by default, which could also be called mathematical centering. Sometimes this type of centering looks wrong - e.g. for arrows, so we instead have an option to use a 'center of mass' style centering for right facing and left facing triangles.

Parameters
method -- this is an ID for the method of centering to use, for almost all cases this will be the default 'rect' style basic centering. However, if you are trying to center an arrow you might try 'right_triangle' or 'left_triangle'

insert_layout_rects(*layout_rects: Deque[TextLayoutRect]*, *row_index: int*, *item_index: int*, *chunk_index: int*)

Insert some new layout rectangles from a queue at specific place in the current layout. Hopefully this means we only need to redo the layout after this point... we shall see.

Warning: this is a test function, it may not be up-to-date with current text layout features

Parameters

- **layout_rects** -- the new TextLayoutRects to insert.
- **row_index** -- which row we are sticking them on.
- **item_index** -- which chunk we are sticking them into.
- **chunk_index** -- where in the chunk we are sticking them.

insert_line_break(*layout_index: int*, *parser: HTMLParser | None*)

Insert a line break into the text layout at a given point.

Parameters

- **layout_index** -- the character index at which to insert the line break.
- **parser** -- An optional HTML parser for text styling data

insert_text(*text: str*, *layout_index: int*, *parser: HTMLParser | None = None*)

Insert some text into the text layout at a given point. Handy when e.g. pasting a chunk of text into an existing layout.

Parameters

- **text** -- the text to insert.
- **layout_index** -- the character index at which to insert the text.
- **parser** -- An optional HTML parser for text styling data

redraw_other_chunks(*not_these_chunks*)

Useful for text effects. TODO: no idea how this will play with images? Probably badly.

Parameters
not_these_chunks -- The chunks not to redraw

Returns

reprocess_layout_queue(*layout_rect*)

Re-lays out already parsed text data. Useful to call if the layout requirements have changed but the text data hasn't.

Parameters
layout_rect -- The new layout rectangle.

set_alpha(*alpha: int*)

Set the overall alpha level of this text box from 0 to 255. This allows us to fade text in and out of view.

Parameters
alpha -- integer from 0 to 255.

set_cursor_colour(*colour: Color*)

Set the colour of the editing carat/text cursor for this text layout.

Parameters
colour -- The colour to set it to.

set_cursor_from_click_pos(*click_pos*)

Set the edit cursor position in the text layout from a pixel position. Generally used to set the text editing cursor position from a mouse click.

Parameters
click_pos -- This is the pixel position we want the cursor to appear near to.

set_cursor_position(*cursor_pos*)

Set the edit cursor position in the text layout.

Parameters
cursor_pos -- This is the index of the character the cursor should appear before.

`set_cursor_to_end_of_current_row()`

Set the edit cursor position in the text layout to the end of the current row and returns the overall position in the text

Returns
the overall position of the cursor in the text layout, after setting it to the end of the current row

`set_cursor_to_start_of_current_row()`

Set the edit cursor position in the text layout to the end of the current row and returns the overall position in the text

Returns
the overall position of the cursor in the text layout, after setting it to the end of the current row

`set_default_text_colour(colour)`

Set the default text colour, used when no other colour is set for a portion of the text.

Parameters
colour -- the colour to use as the default text colour.

`set_default_text_shadow_colour(colour)`

Set the default text shadow colour, used when no other colour is set for the shadow of a portion of the text.

Parameters
colour -- the colour to use as the default text shadow colour.

`set_text_selection(start_index, end_index)`

Set a portion of the text layout as 'selected'. This is useful when editing chunks of text all at once (e.g. copying to a memory 'clipboard', deleting a block of text).

Parameters

- **start_index** -- the character index to start the selection area at.
- **end_index** -- the character index to end the selection area at.

`toggle_cursor()`

Toggle the visibility of the edit cursor.

Used routinely by editable text boxes to make the cursor flash to catch user attention.

`turn_off_cursor()`

Makes the edit test cursor invisible.

`turn_on_cursor()`

Makes the edit test cursor visible.

`update_text_with_new_text_end_pos(new_end_pos: int)`

Sets a new end position for the text in this block and redraws it, so we can display a 'typing' type effect. The text will only be displayed up to the index position set here.

Parameters
new_end_pos -- The new ending index for the text string.

`vert_align_bottom_all_rows(y_padding)`

Align all rows to the bottom of the layout.

Parameters
y_padding -- the amount of padding to insert below before the text starts.

vert_align_top_all_rows(*y_padding*)

Align all rows to the top of the layout.

Parameters
y_padding -- the amount of padding to insert above before the text starts.

vert_center_all_rows()

Vertically center all rows of text in the layout.

TODO: I expect we should have the arrow centering methods in here too.

class pygame_gui.core.**UIAppearanceTheme**(*resource_loader:* IResourceLoader, *locale: str*)

Bases: *IUIAppearanceThemeInterface*

The Appearance Theme class handles all the data that styles and generally dictates the appearance of UI elements across the whole UI.

The styling is split into four general areas:

- colours - spelled in the British English fashion with a 'u'.
- font - specifying a font to use for a UIElement where that is a relevant consideration.
- images - describing any images to be used in a UIElement.
- misc - covering all other types of data and stored as strings.

To change the theming for the UI you normally specify a theme file when creating the UIManager. For more information on theme files see the specific documentation elsewhere.

build_all_combined_ids(*element_base_ids: None | List[str | None]*, *element_ids: None | List[str]*, *class_ids: None | List[str | None]*, *object_ids: None | List[str | None]*) → List[str]

Construct a list of combined element ids from the element's various accumulated ids.

Parameters

- **element_base_ids** -- when an element is also another element (e.g. a file dialog is also a window)
- **element_ids** -- All the ids of elements this element is contained within.
- **class_ids** -- All the ids of 'classes' that this element is contained within.
- **object_ids** -- All the ids of objects this element is contained within.

Returns
A list of IDs that reference this element in order of decreasing specificity.

check_need_to_reload() → bool

Check if we need to reload our theme file because it's been modified. If so, trigger a reload and return True so that the UIManager can trigger elements to rebuild from the theme data.

Return bool
True if we need to reload elements because the theme data has changed.

get_colour(*colour_id: str*, *combined_element_ids: List[str] | None = None*) → Color

Uses data about a UI element and a specific ID to find a colour from our theme.

Parameters

- **combined_element_ids** -- A list of IDs representing an element's location in a hierarchy of elements.

- **colour_id** -- The id for the specific colour we are looking for.

Return pygame.Color
A pygame colour.

get_colour_or_gradient(*colour_id: str*, *combined_ids: List[str] | None = None*) → Color | *IColourGradientInterface*

Uses data about a UI element and a specific ID to find a colour, or a gradient, from our theme. Use this function if the UIElement can handle either type.

Parameters

- **combined_ids** -- A list of IDs representing an element's location in a hierarchy of elements.
- **colour_id** -- The id for the specific colour we are looking for.

Return pygame.Color or ColourGradient
A colour or a gradient object.

get_font(*combined_element_ids: List[str]*) → *IGUIFontInterface*

Uses some data about a UIElement to get a font object.

Parameters
combined_element_ids -- A list of IDs representing an element's location in an interleaved hierarchy of elements.

Return IGUIFontInterface
An interface to a pygame font object wrapper.

get_font_dictionary() → *IUIFontDictionaryInterface*

Lets us grab the font dictionary, which is created by the theme object, so we can access it directly.

Returns
The font dictionary.

get_font_info(*combined_element_ids: List[str]*) → Dict[str, Any]

Uses some data about a UIElement to get font data as dictionary

Parameters
combined_element_ids -- A list of IDs representing an element's location in an interleaved hierarchy of elements.

Return dictionary
Data about the font requested

get_image(*image_id: str*, *combined_element_ids: List[str]*) → Surface

Will raise an exception if no image with the ids specified is found. UI elements that have an optional image display will need to handle the exception.

Parameters

- **combined_element_ids** -- A list of IDs representing an element's location in a hierarchy of elements.
- **image_id** -- The id identifying the particular image spot in the UI we are looking for an image to add to.

Returns
A pygame.surface.Surface

get_misc_data(*misc_data_id: str*, *combined_element_ids: List[str]*) → str | Dict

Uses data about a UI element and a specific ID to try and find a piece of miscellaneous theming data. Raises an exception if it can't find the data requested, UI elements requesting optional data will need to handle this exception.

Parameters

- **combined_element_ids** -- A list of IDs representing an element's location in a hierarchy of elements.
- **misc_data_id** -- The id for the specific piece of miscellaneous data we are looking for.

Return Any
Returns a string or a Dict

load_theme(*file_path: str | PathLike | StringIO | PackageResource | dict*)

Loads a theme, and currently, all associated data like fonts and images required by the theme.

Parameters
file_path -- The location of the theme, or the theme data we want to load.

reload_theming()

We need to load our theme file to see if anything expensive has changed, if so trigger it to reload/rebuild.

set_locale(*locale: str*)

Set the locale used in the appearance theme.

Parameters
locale -- a two-letter ISO country code.

update_caching(*time_delta: float*)

Updates the various surface caches.

update_single_element_theming(*element_name: str*, *new_theming_data: str | dict*)

Update theming data, via string - for a single element. :param element_name: :param new_theming_data: :return:

update_theming(*new_theming_data: str | dict*, *rebuild_all: bool = True*)

Update theming data, via string - for the whole UI.

Parameters

- **new_theming_data** --
- **rebuild_all** --

Returns

class pygame_gui.core.**UIContainer**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *manager:* IUIManagerInterface, ***, *starting_height: int = 1*, *is_window_root_container: bool = False*, *container:* IContainerLikeInterface | *None = None*, *parent_element:* UIElement | *None = None*, *object_id:* ObjectID | *str* | *None = None*, *element_id: List[str]* | *None = None*, *anchors: Dict[str, str]* | *None = None*, *visible: int = 1*)

Bases: *UIElement*, *IUIContainerInterface*, *IContainerLikeInterface*

A UI Container holds any number of other UI elements inside a rectangle. When we move the UIContainer all the UI elements contained within it can be moved as well.

This class helps us make UI Windows, but likely will have wider uses as well as the GUI system develops.

Parameters

- **relative_rect** -- A pygame.Rect whose position is relative to whatever UIContainer it is inside, if any.
- **manager** -- The UIManager that manages this UIElement.
- **starting_height** -- The starting layer height for this element above its container.
- **is_window_root_container** -- True/False flag for whether this container is the root container for a UI window.
- **container** -- The UIContainer that this UIElement is contained within.
- **parent_element** -- The element this element 'belongs to' in the theming hierarchy.
- **object_id** -- A custom defined ID for fine-tuning of theming.
- **anchors** -- A dictionary describing what this element's relative_rect is relative to.
- **visible** -- Whether the container and its children are visible by default. Warning - it's parent container visibility may override this.

add_element(*element:* IUIElementInterface)

Add a UIElement to the container. The UI's relative_rect parameter will be relative to this container.

Parameters
element -- A UIElement to add to this container.

are_contents_hovered() → bool

Are any of the elements in the container hovered? Used for handling mousewheel events.

Returns
True if one of the elements is hovered, False otherwise.

calc_add_element_changes_thickness(*element:* IUIElementInterface)

This function checks if a single added element will increase the containers thickness and if so updates containers recursively.

Parameters
element -- the element to check.

change_layer(*new_layer: int*)

Change the layer of this container. Layers are used by the GUI to control the order in which things are drawn and which things should currently be interactive (so you can't interact with things behind other things).

This particular method is most often used to shift the visible contents of a window in front of any others when it is moved to the front of the window stack.

Parameters
new_layer -- The layer to move our container to.

check_hover(*time_delta: float*, *hovered_higher_element: bool*) → bool

A method that helps us to determine which, if any, UI Element is currently being hovered by the mouse.

Parameters

- **time_delta** -- A float, the time in seconds between the last call to this function and now (roughly).
- **hovered_higher_element** -- A boolean, representing whether we have already hovered a 'higher' element.

Returns
A boolean that is true if we have hovered a UI element, either just now or before this method.

`clear()`

Removes and kills all the UI elements inside this container.

`disable()`

Disables all elements in the container, so they are no longer interactive.

`enable()`

Enables all elements in the container, so they are interactive again.

`expand_left(width_increase: int) → None`

Increases the width of the container, but instead of expanding the right edge, it expands the left edge. This is achieved by setting the new dimensions and updating the anchors of all the elements anchored to the left of the container.

Parameters
width_increase -- The width to increase by. Pass in negative values to decrease the size

Returns
None

`expand_top(height_increase: int) → None`

Increases the height of the container, but instead of expanding the bottom edge, it expands the top edge. This is achieved by setting the new dimensions and updating the anchors of all the elements anchored to the top of the container.

Parameters
height_increase -- The height to increase by. Pass in negative values to decrease the size

Returns
None

`get_container() → IUIContainerInterface`

Implements the container interface. In this case we just return this since it is a container.

Returns
This container.

`get_rect() → Rect`

Access to the container's rect

Returns
a pygame rectangle

`get_size() → Tuple[int, int]`

Get the container's pixel size.

Returns
the pixel size as tuple [x, y]

`get_thickness() → int`

Get the container's layer thickness.

Returns
the thickness as an integer.

get_top_layer() → int

Assuming we have correctly calculated the 'thickness' of this container, this method will return the 'highest' layer in the LayeredDirty UI Group.

Returns
An integer representing the current highest layer being used by this container.

hide()

Hides the container, which means the container will not get drawn and will not process events. Should also hide all the children elements and containers. If the container was hidden before - ignore.

kill()

Overrides the standard kill method of UI Elements (and pygame sprites beyond that) to also call the kill method on all contained UI Elements.

on_contained_elements_changed(*target:* IUIElementInterface) → None

Update the positioning of the contained elements of this container. To be called when one of the contained elements may have moved, been resized or changed its anchors.

Parameters
target -- the UI element that has been benn moved resized or changed its anchors.

recalculate_container_layer_thickness()

This function will iterate through the elements in our container and determine the maximum 'height' that they reach in the 'layer stack'. We then use that to determine the overall 'thickness' of this container. The thickness value is used to determine where to place overlapping windows in the layers

remove_element(*element:* IUIElementInterface)

Remove a UIElement from this container.

Parameters
element -- A UIElement to remove from this container.

set_dimensions(*dimensions: Vector2 | Tuple[float, float]*, *clamp_to_container: bool = False*)

Set the dimension of this container and update the positions of elements within it accordingly.

Parameters

- **dimensions** -- the new dimensions.
- **clamp_to_container** -- Whether we should clamp the dimensions to the dimensions of the container or not.

set_position(*position: Vector2 | Tuple[float, float]*)

Set the absolute position of this container - it is usually less chaotic to deal with setting relative positions.

Parameters
position -- the new absolute position to set.

set_relative_position(*position: Vector2 | Tuple[float, float]*)

Set the position of this container, relative to the container it is within.

Parameters
position -- the new relative position to set.

show()

Shows the container, which means the container will get drawn and will process events. Should also show all the children elements and containers. If the container was visible before - ignore.

update_containing_rect_position()

This function is called when we move the container to update all the contained UI Elements to move as well.

class pygame_gui.core.**UIElement**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *manager:* IUIManagerInterface | *None*, *container:* IContainerLikeInterface | *None*, *, *starting_height: int*, *layer_thickness: int*, *anchors: Dict[str, str | IUIElementInterface]* | *None = None*, *visible: int = 1*, *parent_element:* IUIElementInterface | *None = None*, *object_id:* ObjectID | *str* | *None = None*, *element_id: List[str]* | *None = None*, *ignore_shadow_for_initial_size_and_pos: bool = False*)

Bases: *GUISprite*, *IUIElementInterface*

A base class for UI elements. You shouldn't create UI Element objects, instead all UI Element classes should derive from this class. Inherits from pygame.sprite.Sprite.

Parameters

- **relative_rect** -- A rectangle shape of the UI element, the position is relative to the element's container.
- **manager** -- The UIManager that manages this UIElement.
- **container** -- A container that this element is contained in.
- **starting_height** -- Used to record how many layers above its container this element should be. Normally 1.
- **layer_thickness** -- Used to record how 'thick' this element is in layers. Normally 1.
- **anchors** -- A dictionary describing what this element's relative_rect is relative to.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.
- **parent_element** -- Element that this element 'belongs to' in theming. Elements inherit colours from parents.
- **object_id** -- An optional set of IDs to help distinguish this element from other elements.
- **element_id** -- A list of string ID representing this element's class.

can_hover() → bool

A stub method to override. Called to test if this method can be hovered.

change_layer(*new_layer: int*)

Changes the layer this element is on.

Parameters

new_layer -- The layer to change this element to.

change_object_id(*new_object_id:* ObjectID | *str* | *None*)

Allows for easy switching of an element's ObjectID used for theming and events.

Will rebuild the element after switching the ID

Parameters

new_object_id -- The new ID to use for this element.

Returns

check_hover(*time_delta: float*, *hovered_higher_element: bool*) → bool

A method that helps us to determine which, if any, UI Element is currently being hovered by the mouse.

Parameters

- **time_delta** -- A float, the time in seconds between the last call to this function and now (roughly).
- **hovered_higher_element** -- A boolean, representing whether we have already hovered a 'higher' element.

Return bool
A boolean that is true if we have hovered a UI element, either just now or before this method.

disable()

Disables elements so they are no longer interactive. This is just a default fallback implementation for elements that don't define their own.

Elements should handle their own enabling and disabling.

enable()

Enables elements so they are interactive again. This is just a default fallback implementation for elements that don't define their own.

Elements should handle their own enabling and disabling.

focus()

A stub to override. Called when we focus this UI element.

get_abs_rect() → Rect

The absolute positioning rect.

Returns
A pygame rect.

get_anchor_targets() → list

Get any anchor targets this element has, so we can update them when their targets change :return: the list of anchor targets.

get_anchors() → Dict[str, str | *IUIElementInterface*]

A dictionary containing all the anchors defining what the relative rect is relative to

Returns
A dictionary containing all the anchors defining what the relative rect is relative to

get_class_ids() → List[str]

A list of all the class IDs in this element's theming/event hierarchy.

Returns
a list of strings, one for each element in the hierarchy.

get_element_base_ids() → List[str]

A list of all the element base IDs in this element's theming/event hierarchy.

Returns
a list of strings, one for each element in the hierarchy.

get_element_ids() → List[str]

A list of all the element IDs in this element's theming/event hierarchy.

Returns
a list of strings, one for each element in the hierarchy.

get_focus_set() → Set[*IUIElementInterface*]

Return the set of elements to focus when we focus this element.

get_image_clipping_rect() → Rect | None

Obtain the current image clipping rect.

Returns
The current clipping rect. Maybe None.

get_object_ids() → List[str]

A list of all the object IDs in this element's theming/event hierarchy.

Returns
a list of strings, one for each element in the hierarchy.

get_relative_rect() → Rect

The relative positioning rect.

Returns
A pygame rect.

get_starting_height() → int

Get the starting layer height of this element. (i.e. the layer we start placing it on *above* its container, it may use more layers above this layer)

Returns
an integer representing the starting layer height.

get_top_layer() → int

Assuming we have correctly calculated the 'thickness' of it, this method will return the top of this element.

Return int
An integer representing the current highest layer being used by this element.

hide()

Hides the widget, which means the widget will not get drawn and will not process events. Clear hovered state.

hover_point(*hover_x: float*, *hover_y: float*) → bool

Test if a given point counts as 'hovering' this UI element. Normally that is a straightforward matter of seeing if a point is inside the rectangle. Occasionally it will also check if we are in a wider zone around a UI element once it is already active, this makes it easier to move scroll bars and the like.

Parameters

- **hover_x** -- The x (horizontal) position of the point.
- **hover_y** -- The y (vertical) position of the point.

Returns
Returns True if we are hovering this element.

property hovered: bool

Are we hovering over this element with the mouse pointer or other input highlighting method.

Returns
True if hovered.

join_focus_sets(*element:* IUIElementInterface)

Join this element's focus set with another element's focus set.

Parameters
element -- The other element whose focus set we are joining with.

kill()

Overriding regular sprite kill() method to remove the element from its container.

on_fresh_drawable_shape_ready()

Called when our drawable shape has finished rebuilding the active surface. This is needed because sometimes we defer rebuilding until a more advantageous (read quieter) moment.

on_hovered()

Called when this UI element first enters the 'hovered' state.

on_locale_changed()

Called for each element when the locale is changed on their UIManager

on_unhovered()

Called when this UI element leaves the 'hovered' state.

process_event(*event: Event*) → bool

A stub to override. Gives UI Elements access to pygame events.

Parameters
event -- The event to process.

Returns
Should return True if this element makes use of this event.

rebuild()

Takes care of rebuilding this element. Most derived elements are going to override this, and hopefully call the super() class method.

rebuild_from_changed_theme_data()

A stub to override when we want to rebuild from theme data.

remove_element_from_focus_set(*element*)

remove an element from this sets focus group.

Parameters
element -- The element to remove.

set_anchors(*anchors: Dict[str, str | IUIElementInterface] | None*) → None

Wraps the setting of the anchors with some validation

Parameters
anchors -- A dictionary of anchors defining what the relative rect is relative to

Returns
None

set_container(*container: None | IContainerLikeInterface*)

Switch the element to new container. Remove the element from the old container and add it to the new container.

Parameters
container -- The new container to add.

set_dimensions(*dimensions: Vector2 | Tuple[float, float]*, *clamp_to_container: bool = False*)

Method to directly set the dimensions of an element. And set whether the elements are dynamic.

NOTE: Using this on elements inside containers with non-default anchoring arrangements may make a mess of them.

Parameters

- **dimensions** -- The new dimensions to set. If it is a negative value, the element will become dynamically sized, otherwise it will become statically sized.
- **clamp_to_container** -- Whether we should clamp the dimensions to the dimensions of the container or not.

set_focus_set(*focus_set: Set[*IUIElementInterface*] | None*)

Set the focus set to a specific set of elements.

Parameters
focus_set -- The focus set to set.

set_image(*new_image: Surface | None*)

This used to be the way to set the proper way to set the .image property of a UIElement (inherited from pygame.Sprite), but it is intended for internal use in the library - not for adding actual images/pictures on UIElements. As such I've renamed the original function to make it protected and not part of the interface and deprecated this one for most elements.

Returns

set_minimum_dimensions(*dimensions: Vector2 | Tuple[float, float]*)

If this window is resizable, then the dimensions we set here will be the minimum that users can change the window to. They are also used as the minimum size when 'set_dimensions' is called.

Parameters
dimensions -- The new minimum dimension for the window.

set_position(*position: Vector2 | Tuple[float, float]*)

Method to directly set the absolute screen rect position of an element.

Parameters
position -- The new position to set.

set_relative_position(*position: Vector2 | Tuple[float, float]*)

Method to directly set the relative rect position of an element.

Parameters
position -- The new position to set.

set_visual_debug_mode(*activate_mode: bool*)

Enables a debug mode for the element which displays layer information on top of it in a tiny font.

Parameters
activate_mode -- True or False to enable or disable the mode.

show()

Shows the widget, which means the widget will get drawn and will process events.

unfocus()

A stub to override. Called when we stop focusing this UI element.

update(*time_delta: float*)

Updates this element's drawable shape, if it has one.

Parameters
time_delta -- The time passed between frames, measured in seconds.

update_containing_rect_position()

Updates the position of this element based on the position of its container. Usually called when the container has moved.

update_theming(*new_theming_data: str*)

Update the theming for this element using the most specific ID assigned to it.

If you have not given this element a unique ID, this function will also update the theming of other elements of this theming class or of this element type.

Parameters
new_theming_data -- the new theming data in a json string

while_hovering(*time_delta: float*, *mouse_pos: Vector2 | Tuple[int, int] | Tuple[float, float]*)

Called while we are in the hover state. It will create a tool tip if we've been in the hover state for a while, the text exists to create one, and we haven't created one already.

Parameters

- **time_delta** -- Time in seconds between calls to update.
- **mouse_pos** -- The current position of the mouse.

class pygame_gui.core.**UIFontDictionary**(*resource_loader: IResourceLoader*, *locale: str*)

Bases: *IUIFontDictionaryInterface*

The font dictionary is used to store all the fonts that have been loaded into the UI system.

add_font_path(*font_name: str*, *font_path: str | PackageResource*, *bold_path: str | PackageResource | None = None*, *italic_path: str | PackageResource | None = None*, *bold_italic_path: str | PackageResource | None = None*)

Adds paths to different font files for a font name.

Parameters

- **font_name** -- The name to assign to these font files.
- **font_path** -- The path to the font's file with no particular style.
- **bold_path** -- The path to the font's file with a bold style.
- **italic_path** -- The path to the font's file with an italic style.
- **bold_italic_path** -- The path to the font's file with a bold and an italic style.

check_font_preloaded(*font_id: str*) → bool

Check if a font is already preloaded or not.

Parameters
font_id -- The ID of the font to check for

Returns
True or False.

convert_html_to_point_size(*html_size: float*) → int

Takes in an HTML style font size and converts it into a point font size.

Parameters
html_size -- Size in HTML style.

Return int
A 'point' font size.

create_font_id(*font_size: int*, *font_name: str*, *bold: bool*, *italic: bool*, *antialiased: bool = True*) → str

Create an id for a particularly styled and sized font from those characteristics.

Parameters

- **font_size** -- The size of the font.
- **font_name** -- The name of the font.
- **bold** -- Whether the font is bold styled or not.
- **italic** -- Whether the font is italic styled or not.
- **antialiased** -- Whether the font is antialiased or not.

Return str
The finished font id.

ensure_debug_font_loaded()

Ensure the font we use for debugging purposes is loaded. Generally called after we start a debugging mode.

find_font(*font_size: int*, *font_name: str*, *bold: bool = False*, *italic: bool = False*, *antialiased: bool = True*, *script: str = 'Latn'*, *direction: int = 0*) → *IGUIFontInterface*

Find a loaded font from the font dictionary. Will load a font if it does not already exist, and we have paths to the needed files, however it will issue a warning after doing so because dynamic file loading is normally a bad idea as you will get frame rate hitches while the running program waits for the font to load.

Instead, it's best to preload all your needed files at another time in your program when you have more control over the user experience.

Parameters

- **font_size** -- The size of the font to find.
- **font_name** -- The name of the font to find.
- **bold** -- Whether the font is bold or not.
- **italic** -- Whether the font is italic or not.
- **antialiased** -- Whether the font is antialiased or not.
- **script** -- The ISO 15924 script code used for text shaping as a string.
- **direction** -- the direction of text e.g. left to right or right to left. An integer.

Return IGUIFontInterface
Returns either the font we asked for, or the default font.

find_font_resource(*font_size: int*, *font_name: str*, *bold: bool = False*, *italic: bool = False*, *antialiased: bool = True*, *script: str = 'Latn'*, *direction: int = 0*) → FontResource

Find a loaded font resource from the font dictionary. Will load a font if it does not already exist, and we have paths to the needed files, however it will issue a warning after doing so because dynamic file loading is normally a bad idea as you will get frame rate hitches while the running program waits for the font to load.

Instead, it's best to preload all your needed files at another time in your program when you have more control over the user experience.

Parameters

- **font_size** -- The size of the font to find.
- **font_name** -- The name of the font to find.
- **bold** -- Whether the font is bold or not.
- **italic** -- Whether the font is italic or not.
- **antialiased** -- Whether the font is antialiased or not.
- **script** -- The ISO 15924 script code used for text shaping as a string.
- **direction** -- the direction of text e.g. left to right or right to left. An integer.

Return FontResource
Returns either the font resource we asked for, or the default font.

`get_default_font()` → *IGUIFontInterface*

Grab the default font.

Returns
The default font.

`preload_font`(*font_size: int*, *font_name: str*, *bold: bool = False*, *italic: bool = False*, *force_immediate_load: bool = False*, *antialiased: bool = True*, *script: str = 'Latn'*, *direction: int = 0*)

Lets us load a font at a particular size and style before we use it. While you can get away with relying on dynamic font loading during development, it is better to eventually preload all your font data at a controlled time, which is where this method comes in.

Parameters

- **font_size** -- The size of the font to load.
- **font_name** -- The name of the font to load.
- **bold** -- Whether the font is bold styled or not.
- **italic** -- Whether the font is italic styled or not.
- **force_immediate_load** -- resource loading setup to immediately load the font on the main thread.
- **antialiased** -- Whether the font is antialiased or not.
- **script** -- The ISO 15924 script code used for text shaping as a string.
- **direction** -- the direction of text e.g. left to right or right to left. An integer.

`print_unused_loaded_fonts()`

Can be called to check if the UI is loading any fonts that we haven't used by the point this function is called. If a font is truly unused then we can remove it from our loading and potentially speed up the overall loading of the program.

This is not a foolproof check because this function could easily be called before we have explored all the code paths in a project that may use fonts.

`set_locale`(*new_locale: str*)

This may change the default font.

Parameters
new_locale -- The new locale to set, a two-letter country code ISO 639-1

class pygame_gui.core.**UIWindowStack**(*window_resolution: Tuple[int, int]*, *root_container:* IUIContainerInterface)

Bases: *IUIWindowStackInterface*

A class for managing a stack of GUI windows so that only one is 'in front' at a time and the rest are sorted based on the last time they were interacted with/created.

Parameters

- **window_resolution** -- The resolution of the OS window.
- **root_container** -- The root container for the whole UI.

add_new_window(*window:* IWindowInterface)

Adds a new window to the top of the stack.

Parameters
window -- The window to add.

clear()

Empties the whole stack removing and killing all windows.

get_full_stack() → List[*IWindowInterface*]

Returns the full stack of normal and always on top windows.

Returns
a list of Windows

is_window_at_top(*window:* IWindowInterface) → bool

Checks if a window is at the top of the normal window stack or not.

Parameters
window -- The window to check.

Returns
returns True if this window is at the top of the stack.

is_window_at_top_of_top(*window:* IWindowInterface) → bool

Checks if a window is at the top of the top window stack or not.

Parameters
window -- The window to check.

Returns
returns True if this window is at the top of the stack.

move_window_to_front(*window_to_front:* IWindowInterface)

Moves the passed in window to the top of its stack and resorts the other windows to deal with the change.

Parameters
window_to_front -- the window to move to the front.

remove_window(*window_to_remove:* IWindowInterface)

Removes a window from the stack and resorts the remaining windows to adjust for its absence.

Parameters
window_to_remove -- the window to remove.

pygame_gui.elements package

Submodules

pygame_gui.elements.ui_2d_slider module

class pygame_gui.elements.ui_2d_slider.**UI2DSlider**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *start_value_x: float | int*, *value_range_x: Tuple[float, float] | Tuple[int, int]*, *start_value_y: float | int*, *value_range_y: Tuple[float, float] | Tuple[int, int]*, *invert_y: bool = False*, *manager: IUIManagerInterface | None = None*, *container: IContainerLikeInterface | None = None*, *starting_height: int = 1*, *parent_element: UIElement | None = None*, *object_id: ObjectID | str | None = None*, *anchors: Dict[str, str | UIElement] | None = None*, *visible: int = 1*)

Bases: *UIElement*

A 2d slider is intended to help users adjust values within a range, for example a volume control.

Parameters

- **relative_rect** -- A rectangle describing the position and dimensions of the element.
- **start_value_x** -- The x value to start the slider at.
- **value_range_x** -- The full range of x values.
- **start_value_y** -- The y value to start the slider at.
- **value_range_y** -- The full range of y values.
- **invert_y** -- Should the y increase from bottom to top instead of the other way round?
- **manager** -- The UIManager that manages this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **container** -- The container that this element is within. If set to None will be the root window's container.
- **parent_element** -- The element this element "belongs to" in the theming hierarchy.
- **object_id** -- A custom defined ID for fine-tuning of theming.
- **anchors** -- A dictionary describing what this element's relative_rect is relative to.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.

disable()

Disable the slider. It should not be interactive and will use the disabled theme colours.

enable()

Enable the slider. It should become interactive and will use the normal theme colours.

get_current_value() → Tuple[float, int]

Gets the current value the slider is set to.

Returns

The current value recorded by the slider.

hide()

In addition to the base UIElement.hide() - hide the sliding button and hide the button_container which will propagate and hide the left and right buttons.

kill()

Overrides the normal sprite kill() method to also kill the button elements that help make up the slider.

rebuild()

Rebuild anything that might need rebuilding.

rebuild_from_changed_theme_data() → None

Called by the UIManager to check the theming data and rebuild whatever needs rebuilding for this element when the theme data has changed.

Returns
None

set_current_value(*value_x: float | int*, *value_y: float | int*, *warn: bool = True*) → None

Sets the value of the slider, which will move the position of the slider to match. Will issue a warning if the value set is not in the value range.

Parameters

- **value_x** -- The x value to set.
- **value_y** -- The y value to set.
- **warn** -- set to false in order to suppress the default warning, instead the value will be clamped.

Param
invert_y: Should the y value be inverted? If not passed then will use self.invert_value for this info.

Returns
None

set_dimensions(*dimensions: Vector2 | Tuple[float, float]*, *clamp_to_container: bool = False*)

Method to directly set the dimensions of an element.

Parameters

- **dimensions** -- The new dimensions to set.
- **clamp_to_container** -- Whether we should clamp the dimensions to the dimensions of the container or not.

set_position(*position: Vector2 | Tuple[float, float]*) → None

Sets the absolute screen position of this slider, updating all subordinate button elements at the same time.

Parameters
position -- The absolute screen position to set.

Returns
None

set_relative_position(*position: Vector2 | Tuple[float, float]*) → None

Sets the relative screen position of this slider, updating all subordinate button elements at the same time.

Parameters
position -- The relative screen position to set.

Returns
None

`show()`

In addition to the base UIElement.show() - show the sliding button and show the button_container which will propagate and show the left and right buttons.

`update(time_delta: float)`

Takes care of actually moving the slider based on interactions reported by the buttons or based on movement of the mouse if we are gripping the slider itself.

Parameters
`time_delta` -- the time in seconds between calls to update.

pygame_gui.elements.ui_auto_resizing_container module

class pygame_gui.elements.ui_auto_resizing_container.**UIAutoResizingContainer**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *min_edges_rect: Rect | None = None*, *max_edges_rect: Rect | None = None*, *resize_left: bool = True*, *resize_right: bool = True*, *resize_top: bool = True*, *resize_bottom: bool = True*, *manager: IUIManagerInterface | None = None*, ***, *starting_height: int = 1*, *container: IContainerLikeInterface | None = None*, *parent_element: UIElement | None = None*, *object_id: ObjectID | str | None = None*, *anchors: Dict[str, str | UIElement] | None = None*, *visible: int = 1*)

Bases: *UIContainer*

A container like UI element that updates its size as elements within it change size, or new elements are added

Parameters

- **relative_rect** -- The starting size and relative position of the container.
- **min_edges_rect** -- The Rect which defines the maximum values for the left and top, and the minimum values for the right and bottom edges of the container. Defaults to the None (current rect)
- **max_edges_rect** -- The Rect which defines the minimum values for the left and top, and the maximum values for the right and bottom edges of the container. Defaults to the None (unbounded)
- **resize_left** -- Should the left side be resized?

- **resize_right** -- Should the right side be resized?
- **resize_top** -- Should the top side be resized?
- **resize_bottom** -- Should the bottom side be resized?
- **manager** -- The UI manager for this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **starting_height** -- The starting layer height of this container above its container. Defaults to 1.
- **container** -- The container this container is within. Defaults to None (which is the root container for the UI)
- **parent_element** -- A parent element for this container. Defaults to None, or the container if you've set that.
- **object_id** -- An object ID for this element.
- **anchors** -- Layout anchors in a dictionary.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.

add_element(*element:* IUIElementInterface) → None

Add a UIElement to the container. The UIElement's relative_rect parameter will be relative to this container. Overridden to also update dimensions.

Parameters
element -- A UIElement to add to this container.

Returns
None

on_contained_elements_changed(*target:* UIElement) → None

Update the positioning of the contained elements of this container. To be called when one of the contained elements may have moved, been resized or changed its anchors.

Parameters
target -- the UI element that has been benn moved resized or changed its anchors.

Returns
None

recalculate_abs_edges_rect() → None

Used to recalculate the absolute rects from the min and max edges rect which control the minimum and maximum sizes of the container. Usually called when the container of this container has moved, or the minimum or maximum rects have changed.

Returns
None

remove_element(*element:* IUIElementInterface) → None

Remove a UIElement from this container.

Parameters
element -- A UIElement to remove from this container.

Returns
None

update(*time_delta: float*) → None

Updates the container's size based upon the elements inside.

Call this function if you have added or removed an element from this container and want to update the size in the same frame, otherwise it will update in the next frame.

Parameters
time_delta -- The time passed between frames, measured in seconds.

Returns
None

update_containing_rect_position() → None

Overridden to also recalculate the absolute rects which control the minimum and maximum sizes of the container

Returns
None

update_max_edges_rect(*new_rect: Rect*) → None

Updates the container's minimum values for the left and top, and the maximum value for the right and bottom edges based upon the edges of the new rect.

Call the update function if you want to update the elements contained in the same frame, otherwise the elements contained within will update in the next frame.

Parameters
new_rect -- Rect to update the max_edges_rect with

Returns
None

update_min_edges_rect(*new_rect: Rect*) → None

Updates the container's maximum values for the left and top, and the minimum value for the right and bottom edges based upon the edges of the new rect.

Call the update function if you want to update the elements contained in the same frame, otherwise the elements contained within will update in the next frame.

Parameters
new_rect -- Rect to update the min_edges_rect with

Returns
None

pygame_gui.elements.ui_button module

class pygame_gui.elements.ui_button.**UIButton**(*relative_rect: Rect | FRect | Tuple[float, float, float, float] | Vector2 | Tuple[float, float]*, *text: str*, *manager: IUIManagerInterface | None = None*, *container: IContainerLikeInterface | None = None*, *tool_tip_text: str | None = None*, *starting_height: int = 1*, *parent_element: UIElement | None = None*, *object_id: ObjectID | str | None = None*, *anchors: Dict[str, str | UIElement] | None = None*, *allow_double_clicks: bool = False*, *generate_click_events_from: Iterable[int] = frozenset({1})*, *visible: int = 1*, ***, *command: Callable | Dict[int, Callable] | None = None*, *tool_tip_object_id: ObjectID | None = None*, *text_kwargs: Dict[str, str] | None = None*, *tool_tip_text_kwargs: Dict[str, str] | None = None*, *max_dynamic_width: int | None = None*)

Bases: *UIElement*

A push button, a lot of the appearance of the button, including images to be displayed, is set up via the theme file. This button is designed to be pressed, do something, and then reset - rather than to be toggled on or off.

The button element is reused throughout the UI as part of other elements as it happens to be a very flexible interactive element.

Parameters

- **relative_rect** -- Normally a rectangle describing the position (relative to its container) and dimensions. Also accepts a position Coordinate where the dimensions will be dynamic depending on the text contents. Dynamic dimensions can be requested by setting the required dimension to -1.
- **text** -- Text for the button.
- **manager** -- The UIManager that manages this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **container** -- The container that this element is within. If not provided or set to None will be the root window's container.
- **tool_tip_text** -- Optional tool tip text, can be formatted with HTML. If supplied will appear on hover.
- **starting_height** -- The height in layers above its container that this element will be placed.
- **parent_element** -- The element this element 'belongs to' in the theming hierarchy.
- **object_id** -- A custom defined ID for fine-tuning of theming.
- **anchors** -- A dictionary describing what this element's relative_rect is relative to.
- **allow_double_clicks** -- Enables double-clicking on buttons which will generate a unique event.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.
- **command** -- Functions to be called when an event is triggered by this element.
- **text_kwargs** -- a dictionary of variable arguments to pass to the translated string useful when you have multiple translations that need variables inserted in the middle.

bind(*event: int*, *function: Callable | None = None*)

Bind a function to an element event.

Parameters

- **event** -- The event to bind.
- **function** -- The function to bind. None to unbind.

can_hover() → bool

Tests whether we can trigger the hover state for this button, other states take priority over it.

Returns
True if we are able to hover this button.

check_pressed() → bool

A direct way to check if this button has been pressed in the last update cycle.

Returns
True if the button has been pressed.

disable()

Disables the button so that it is no longer interactive.

enable()

Re-enables the button, so we can once again interact with it.

hide()

In addition to the base UIElement.hide() - Change the hovered state to a normal state.

hover_point(*hover_x: int*, *hover_y: int*) → bool

Tests if a position should be considered 'hovering' the button. Normally this just means our mouse pointer is inside the buttons rectangle, however if we are holding onto the button for a purpose(e.g. dragging a window around by its menu bar) the hover radius can be made to grow, so we don't keep losing touch with whatever we are moving.

Parameters

- **hover_x** -- horizontal pixel coordinate to test.
- **hover_y** -- vertical pixel coordinate to test

Returns
Returns True if we are hovering.

in_hold_range(*position: Vector2 | Tuple[int, int] | Tuple[float, float]*) → bool

Imagines a potentially larger rectangle around our button in which range we still grip hold of our button when moving the mouse. Makes it easier to use scrollbars.

Parameters
position -- The position we are testing.

Return bool
Returns True if our position is inside the hold range.

kill()

Overrides the standard sprite kill method to also kill any tooltips belonging to this button.

on_hovered()

Called when we enter the hover state, it sets the colours and image of the button to the appropriate values and redraws it.

on_locale_changed()

Called for each element when the locale is changed on their UIManager

on_self_event(*event: int*, *data: Dict[str, Any] | None = None*)

Called when an event is triggered by this element. Handles these events either by posting the event back to the event queue, or by running a function supplied by the user.

Parameters

- **event** -- The event triggered.
- **data** -- event data

on_unhovered()

Called when we leave the hover state. Resets the colours and images to normal and kills any tooltip that was created while we were hovering the button.

process_event(*event: Event*) → bool

Handles various interactions with the button.

Parameters
event -- The event to process.

Returns
Return True if we want to consume this event, so it is not passed on to the rest of the UI.

rebuild()

A complete rebuild of the drawable shape used by this button.

rebuild_from_changed_theme_data()

Checks if any theming parameters have changed, and if so triggers a full rebuild of the button's drawable shape

select()

Called when we select focus this element. Changes the colours and image to the appropriate ones for the new state then redraws the button.

set_hold_range(*xy_range: Tuple[int, int]*)

Set x and y values, in pixels, around our button to use as the hold range for time when we want to drag a button about but don't want it to slip out of our grasp too easily.

Imagine it as a large rectangle around our button, larger in all directions by whatever values we specify here.

Parameters
xy_range -- The x and y values used to create our larger 'holding' rectangle.

set_text(*text: str*, ***, *text_kwargs: Dict[str, str] | None = None*)

Sets the text on the button. The button will rebuild.

Parameters

- **text** -- The new text to set.
- **text_kwargs** -- a dictionary of variable arguments to pass to the translated string useful when you have multiple translations that need variables inserted in the middle.

unselect()

Called when we are no longer select focusing this element. Restores the colours and image to the default state then redraws the button.

update(*time_delta: float*)

Sets the pressed state for an update cycle if we've pressed this button recently.

Parameters
time_delta -- the time in seconds between one call to update and the next.

pygame_gui.elements.ui_drop_down_menu module

class pygame_gui.elements.ui_drop_down_menu.**UIClosedDropDownState**(*drop_down_menu_ui: UIDropDownMenu, selected_option: Tuple[str, str], base_position_rect: Rect | None, open_button_width: int, expand_direction: str | None, manager: IUIManagerInterface, container: IContainerLikeInterface, object_ids: List[str | None] | None, element_ids: List[str] | None, visible: int = 1, expand_on_option_click=True*)

Bases: object

The closed state of the drop-down just displays the currently chosen option and a button that will switch the menu to the expanded state.

Parameters

- **drop_down_menu_ui** -- The UIDropDownElement this state belongs to.
- **selected_option** -- The currently selected option.
- **base_position_rect** -- Position and dimensions rectangle.
- **open_button_width** -- Width of open button.
- **expand_direction** -- Direction of expansion, 'up' or 'down'.
- **manager** -- The UI Manager for the whole UI.
- **container** -- The container the element is within.
- **object_ids** -- The object IDs for the drop-down UI element.
- **element_ids** -- The element IDs for the drop-down UI element.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.

disable()

Disables the closed state so that it is no longer interactive.

enable()

Re-enables the closed state, so we can once again interact with it.

finish()

Called when we leave the closed state. Kills the open button and the selected option button.

hide()

Hide selected_option_button and open_button.

process_event(*event: Event*) → bool

Processes events for the closed state of the drop-down.

Parameters
event -- The event to process.

Returns
Return True if we want to consume this event, so it is not passed on to the rest of the UI.

rebuild()

Rebuild the closed state from theming parameters and dimensions.

show()

Show selected_option_button and open_button.

start(*should_rebuild: bool = True*)

Called each time we enter the closed state. It creates the necessary elements, the selected option and the open button.

update_dimensions()

Update the dimensions of all the button elements in the closed drop-down state.

Used when the dimensions of the drop-down have been altered.

update_position()

Update the position of all the button elements in the closed drop-down state.

Used when the position of the drop-down has been altered directly, rather than when it has been moved as a consequence of its container being moved.

class pygame_gui.elements.ui_drop_down_menu.**UIDropDownMenu**(*options_list: List[str] | List[Tuple[str, str]]*, *starting_option: str | Tuple[str, str]*, *relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *manager: IUIManagerInterface | None = None*, *container: IContainerLikeInterface | None = None*, *parent_element: UIElement | None = None*, *object_id: ObjectID | str | None = None*, *expansion_height_limit: int | None = None*, *anchors: Dict[str, str | UIElement] | None = None*, *visible: int = 1*, ***, *expand_on_option_click: bool = True*)

Bases: *UIContainer*

A drop-down menu lets us choose one text option from a list. That list of options can be expanded and hidden at the press of a button. While the element is called a drop-down, it can also be made to 'climb up' by changing the 'expand_direction' styling option to 'up' in the theme file.

The drop-down is implemented through two states, one representing the 'closed' menu state and one for when it has been 'expanded'.

Parameters

- **options_list** -- The list of options to choose from. They must be strings.
- **starting_option** -- The starting option, selected when the menu is first created.
- **relative_rect** -- The size and position of the element when not expanded.

- **manager** -- The UIManager that manages this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **container** -- The container that this element is within. If set to None will be the root window's container.
- **parent_element** -- The element this element 'belongs to' in the theming hierarchy.
- **object_id** -- A custom defined ID for fine-tuning of theming.
- **expansion_height_limit** -- Limit on the height that this will expand to, defaults to the container bounds.
- **anchors** -- A dictionary describing what this element's relative_rect is relative to.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.
- **expand_on_option_click** -- If this is set to False the drop-down will only expand when the open/close button is pressed and not when the selected option is pressed.

add_options(*new_options: List[str] | List[Tuple[str, str]]*) → None

Add new options to the drop-down. Will close the drop-down if it is currently open.

In many cases it may be easier just to recreate the drop-down with whatever the new options list is.

Parameters
new_options -- The list of new options to add.

disable()

Disables the button so that it is no longer interactive.

enable()

Re-enables the button so, we can once again interact with it.

hide()

In addition to the base UIElement.hide() - if the current state is 'expanded' call its hide() method, which begins a transition of the UIDropDownMenu to the 'closed' state, and call the hide() method of the 'closed' state which hides all it's children widgets.

kill()

Overrides the standard sprite kill to also properly kill/finish the current state of the drop-down. Depending on whether it is expanded or closed the drop-down menu will have different elements to clean up.

on_fresh_drawable_shape_ready()

Called by an element's drawable shape when it has a new image surface ready for use, normally after a rebuilding/redrawing of some kind.

process_event(*event: Event*) → bool

Handles various interactions with the drop-down menu by passing them along to the active state.

Parameters
event -- The event to process.

Returns
Return True if we want to consume this event, so it is not passed on to the rest of the UI.

rebuild()

A complete rebuild of the drawable parts of this element.

rebuild_from_changed_theme_data()

Triggers the element to rebuild if any of its theming data has changed, which involves a lot of checking and validating its theming data.

remove_options(*options_to_remove: List[str] | List[Tuple[str, str]]*) → None

Will remove all instances of the options provided.

Parameters
options_to_remove -- The list of new options to remove.

set_dimensions(*dimensions: Vector2 | Tuple[float, float]*, *clamp_to_container: bool = False*)

Sets the dimensions of this drop down, updating all subordinate button elements at the same time.

Parameters

- **dimensions** -- The new dimensions to set.
- **clamp_to_container** -- Whether we should clamp the dimensions to the dimensions of the container or not.

set_position(*position: Vector2 | Tuple[float, float]*)

Sets the absolute screen position of this drop down, updating all subordinate button elements at the same time.

Parameters
position -- The absolute screen position to set.

set_relative_position(*position: Vector2 | Tuple[float, float]*)

Sets the relative screen position of this drop down, updating all subordinate button elements at the same time.

Parameters
position -- The relative screen position to set.

show()

In addition to the base UIElement.show() - call show() on the closed state - showing its buttons.

unfocus()

A stub to override. Called when we stop focusing this UI element.

update(*time_delta: float*)

The update here deals with transitioning between the two states of the drop-down menu and then passes the rest of the work onto whichever state is active.

Parameters
time_delta -- The time in second between calls to update.

class pygame_gui.elements.ui_drop_down_menu.**UIExpandedDropDownState**(*drop_down_menu_ui: UIDropDownMenu*, *options_list: List[Tuple[str, str]]*, *selected_option: Tuple[str, str]*, *base_position_rect: Rect | None*, *close_button_width: int*, *expand_direction: str | None*, *manager: IUIManagerInterface*, *container: IContainerLikeInterface*, *object_ids: List[str | None] | None*, *element_ids: List[str] | None*, *expand_on_option_click*)

Bases: object

The expanded state of the drop-down displays the currently chosen option, all the available options and a button to close the menu and return to the closed state.

Picking an option will also close the menu.

Parameters

- **drop_down_menu_ui** -- The UIDropDownElement this state belongs to.
- **options_list** -- The list of options in this drop down.
- **selected_option** -- The currently selected option.
- **base_position_rect** -- Position and dimensions rectangle.
- **close_button_width** -- Width of close button.
- **expand_direction** -- Direction of expansion, 'up' or 'down'.
- **manager** -- The UI Manager for the whole UI.
- **container** -- The container the element is within.
- **object_ids** -- The object IDs for the drop-down UI element.
- **element_ids** -- The element IDs for the drop-down UI element.

finish()

cleans everything up upon exiting the expanded menu state.

hide()

Transition from expanded state to closed state.

process_event(*event: Event*) → bool

Processes events for the closed state of the drop-down.

Parameters
event -- The event to process.

Returns
Return True if we want to consume this event, so it is not passed on to the rest of the UI.

rebuild()

Rebuild the state from theming parameters and dimensions.

start(*should_rebuild: bool = True*)

Called each time we enter the expanded state. It creates the necessary elements, the selected option, all the other available options and the close button.

update_dimensions()

Update the dimensions of all the button elements in the closed drop-down state.

Used when the dimensions of the drop-down have been altered.

update_position()

Update the position of all the button elements in the open drop-down state.

Used when the position of the drop-down has been altered directly, rather than when it has been moved as a consequence of its container being moved.

pygame_gui.elements.ui_horizontal_scroll_bar module

class pygame_gui.elements.ui_horizontal_scroll_bar.**UIHorizontalScrollBar**(*relative_rect: Rect | FRect | Tuple[float, float, float, float], visible_percentage: float, manager: IUIManagerInterface | None = None, container: IContainerLikeInterface | None = None, parent_element: UIElement | None = None, object_id: ObjectID | str | None = None, anchors: Dict[str, str | UIElement] | None = None, visible: int = 1*)

Bases: *UIElement*

A horizontal scroll bar allows users to position a smaller visible area within a horizontally larger area.

Parameters

- **relative_rect** -- The size and position of the scroll bar.
- **visible_percentage** -- The horizontal percentage of the larger area that is visible, between 0.0 and 1.0.
- **manager** -- The UIManager that manages this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **container** -- The container that this element is within. If set to None will be the root window's container.
- **parent_element** -- The element this element 'belongs to' in the theming hierarchy.
- **object_id** -- A custom defined ID for fine-tuning of theming.
- **anchors** -- A dictionary describing what this element's relative_rect is relative to.

- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.

check_has_moved_recently() → bool

Returns True if the scroll bar was moved in the last call to the update function.

Returns
True if we've recently moved the scroll bar, False otherwise.

disable()

Disables the scroll bar, so it is no longer interactive.

enable()

Enables the scroll bar, so it is interactive once again.

hide()

In addition to the base UIElement.hide() - hide the self.button_container which will propagate and hide all the buttons.

kill()

Overrides the kill() method of the UI element class to kill all the buttons in the scroll bar and clear any of the parts of the scroll bar that are currently recorded as the 'last focused horizontal scroll bar element' on the ui manager.

NOTE: the 'last focused' state on the UI manager is used so that the mouse wheel will move whichever scrollbar we last fiddled with even if we've been doing other stuff. This seems to be consistent with the most common mousewheel/scrollbar interactions used elsewhere.

process_event(*event: Event*) → bool

Checks an event from pygame's event queue to see if the scroll bar needs to react to it. In this case it is just mousewheel events, mainly because the buttons that make up the scroll bar will handle the required mouse click events.

Parameters
event -- The event to process.

Returns
Returns True if we've done something with the input event.

rebuild()

Rebuild anything that might need rebuilding.

rebuild_from_changed_theme_data()

Called by the UIManager to check the theming data and rebuild whatever needs rebuilding for this element when the theme data has changed.

redraw_scrollbar()

Redraws the 'scrollbar' portion of the whole UI element. Called when we change the visible percentage.

reset_scroll_position()

Reset the current scroll position back to the top.

set_dimensions(*dimensions: Vector2 | Tuple[float, float]*, *clamp_to_container: bool = False*)

Method to directly set the dimensions of an element.

Parameters

- **dimensions** -- The new dimensions to set.
- **clamp_to_container** -- Whether we should clamp the dimensions to the dimensions of the container or not.

set_position(*position: Vector2 | Tuple[float, float]*)

Sets the absolute screen position of this scroll bar, updating all subordinate button elements at the same time.

Parameters
position -- The absolute screen position to set.

set_relative_position(*position: Vector2 | Tuple[float, float]*)

Sets the relative screen position of this scroll bar, updating all subordinate button elements at the same time.

Parameters
position -- The relative screen position to set.

set_scroll_from_start_percentage(*new_start_percentage: float*)

Set the scroll bar's scrolling position from a percentage between 0.0 and 1.0.

Parameters
new_start_percentage -- the percentage to set.

set_visible_percentage(*percentage: float*)

Sets the percentage of the total 'scrollable area' that is currently visible. This will affect the size of the scrollbar and should be called if the horizontal size of the 'scrollable area' or the horizontal size of the visible area change.

Parameters
percentage -- A float between 0.0 and 1.0 representing the percentage that is visible.

show()

In addition to the base UIElement.show() - show the self.button_container which will propagate and show all the buttons.

property start_percentage

turning start_percentage into a property, so we can round it to mitigate floating point errors

update(*time_delta: float*)

Called once per update loop of our UI manager. Deals largely with moving the scroll bar and updating the resulting 'start_percentage' variable that is then used by other 'scrollable' UI elements to control the point they start drawing.

Reacts to presses of the up and down arrow buttons, movement of the mouse wheel and dragging of the scroll bar itself.

Parameters
time_delta -- A float, roughly representing the time in seconds between calls to this method.

pygame_gui.elements.ui_horizontal_slider module

class pygame_gui.elements.ui_horizontal_slider.**UIHorizontalSlider**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *start_value: float | int*, *value_range: Tuple[float | int, float | int]*, *manager: IUIManagerInterface | None = None*, *container: IContainerLikeInterface | None = None*, *parent_element: UIElement | None = None*, *object_id: ObjectID | str | None = None*, *anchors: Dict[str, str | UIElement] | None = None*, *visible: int = 1*, *click_increment: float | int = 1*)

Bases: *UIElement*

A horizontal slider is intended to help users adjust values within a range, for example a volume control.

Parameters

- **relative_rect** -- A rectangle describing the position and dimensions of the element.
- **start_value** -- The value to start the slider at.
- **value_range** -- The full range of values.
- **manager** -- The UIManager that manages this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **container** -- The container that this element is within. If set to None will be the root window's container.
- **parent_element** -- The element this element 'belongs to' in the theming hierarchy.
- **object_id** -- A custom defined ID for fine-tuning of theming.
- **anchors** -- A dictionary describing what this element's relative_rect is relative to.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.
- **click_increment** -- the amount to increment by when clicking one of the arrow buttons.

disable()

Disable the slider. It should not be interactive and will use the disabled theme colours.

enable()

Enable the slider. It should become interactive and will use the normal theme colours.

get_current_value() → float | int

Gets the current value the slider is set to.

Returns

The current value recorded by the slider.

hide()

In addition to the base UIElement.hide() - hide the sliding button and hide the button_container which will propagate and hide the left and right buttons.

kill()

Overrides the normal sprite kill() method to also kill the button elements that help make up the slider.

process_event(*event: Event*) → bool

A stub to override. Gives UI Elements access to pygame events.

Parameters
event -- The event to process.

Returns
Should return True if this element makes use of this event.

rebuild()

Rebuild anything that might need rebuilding.

rebuild_from_changed_theme_data()

Called by the UIManager to check the theming data and rebuild whatever needs rebuilding for this element when the theme data has changed.

set_current_value(*value: float | int*, *warn: bool = True*)

Sets the value of the slider, which will move the position of the slider to match. Will issue a warning if the value set is not in the value range.

Parameters

- **value** -- The value to set.
- **warn** -- set to 'False' to suppress the default warning, instead the value will be clamped.

set_dimensions(*dimensions: Vector2 | Tuple[float, float]*, *clamp_to_container: bool = False*)

Method to directly set the dimensions of an element.

Parameters

- **dimensions** -- The new dimensions to set.
- **clamp_to_container** -- Whether we should clamp the dimensions to the dimensions of the container or not.

set_position(*position: Vector2 | Tuple[float, float]*)

Sets the absolute screen position of this slider, updating all subordinate button elements at the same time.

Parameters
position -- The absolute screen position to set.

set_relative_position(*position: Vector2 | Tuple[float, float]*)

Sets the relative screen position of this slider, updating all subordinate button elements at the same time.

Parameters
position -- The relative screen position to set.

show()

In addition to the base UIElement.show() - show the sliding button and show the button_container which will propagate and show the left and right buttons.

update(*time_delta: float*)

Takes care of actually moving the slider based on interactions reported by the buttons or based on movement of the mouse if we are gripping the slider itself.

Parameters
time_delta -- the time in seconds between calls to update.

pygame_gui.elements.ui_image module

class pygame_gui.elements.ui_image.**UIImage**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *image_surface: Surface*, *manager: IUIManagerInterface | None = None*, *image_is_alpha_premultiplied: bool = False*, *container: IContainerLikeInterface | None = None*, *parent_element: UIElement | None = None*, *object_id: ObjectID | str | None = None*, *anchors: Dict[str, str | UIElement] | None = None*, *visible: int = 1*, ***, *starting_height: int = 1*)

Bases: *UIElement*

Displays a pygame surface as a UI element, intended for an image, but it can serve other purposes.

Parameters

- **relative_rect** -- The rectangle that contains, positions and scales the image relative to its container.
- **image_surface** -- A pygame surface to display.
- **manager** -- The UIManager that manages this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **container** -- The container that this element is within. If not provided or set to None will be the root window's container.
- **parent_element** -- The element this element 'belongs to' in the theming hierarchy.
- **object_id** -- A custom defined ID for fine-tuning of theming.
- **anchors** -- A dictionary describing what this element's relative_rect is relative to.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.

rebuild_from_changed_theme_data()

A stub to override when we want to rebuild from theme data.

set_dimensions(*dimensions: Vector2 | Tuple[float, float]*, *clamp_to_container: bool = False*)

Set the dimensions of this image, scaling the image surface to match.

Parameters

- **dimensions** -- The new dimensions of the image.
- **clamp_to_container** -- Whether we should clamp the dimensions to the dimensions of the container or not.

set_image(*new_image: Surface | None*, *image_is_alpha_premultiplied: bool = False*) → None

Allows users to change the image displayed on a UIImage element during run time, without recreating the element.

GUI images are converted to the correct format for the GUI if the supplied image is not the dimensions of the UIImage element it will be scaled to fit. In this situation, an original size image is retained as well in case of future resizing events.

Parameters

- **new_image** -- the new image surface to use in the UIImage element.
- **image_is_alpha_premultiplied** -- set to True if the image is already in alpha multiplied colour format.

pygame_gui.elements.ui_label module

class pygame_gui.elements.ui_label.**UILabel**(*relative_rect: Rect | FRect | Tuple[float, float, float, float] | Vector2 | Tuple[float, float]*, *text: str*, *manager: IUIManagerInterface | None = None*, *container: IContainerLikeInterface | None = None*, *parent_element: UIElement | None = None*, *object_id: ObjectID | str | None = None*, *anchors: Dict[str, str | UIElement] | None = None*, *visible: int = 1*, ***, *text_kwargs: Dict[str, str] | None = None*)

Bases: *UIElement*, *IUITextOwnerInterface*

A label lets us display a single line of text with a single font style. It's a quick to rebuild and simple alternative to the text box element.

Parameters

- **relative_rect** -- Normally a rectangle describing the position (relative to its container) and dimensions. Also accepts a position Coordinate where the dimensions will be dynamic depending on the text contents. Dynamic dimensions can be requested by setting the required dimension to -1.
- **text** -- The text to display in the label.
- **manager** -- The UIManager that manages this label. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **container** -- The container that this element is within. If not provided or set to None will be the root window's container.
- **parent_element** -- The element this element 'belongs to' in the theming hierarchy.
- **object_id** -- A custom defined ID for fine-tuning of theming.
- **anchors** -- A dictionary describing what this element's relative_rect is relative to.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.
- **text_kwargs** -- a dictionary of variable arguments to pass to the translated string useful when you have multiple translations that need variables inserted in the middle.

clear_all_active_effects(*sub_chunk: TextLineChunkFTFont | None = None*)

Clears any active effects and redraws the text. A full reset, usually called before firing off a new effect if one is already in progress.

Parameters

sub_chunk -- An optional chunk so we only clear the effect from this chunk.

clear_text_surface(*sub_chunk: TextLineChunkFTFont | None = None*)

Clear the text surface

Parameters

sub_chunk -- An optional chunk so we only clear the surface for this chunk.

disable()

Disables the label so that its text changes to the disabled colour.

enable()

Re-enables the label so that its text changes to the normal colour

get_object_id() → str

The UI object ID of this text owner for use in effect events.

Returns
the ID string

get_text_letter_count(*sub_chunk:* TextLineChunkFTFont | *None* = *None*) → int

The amount of letters in the text

Parameters
sub_chunk -- An optional chunk to restrict the count to only this chunk.

Returns
number of letters as an int

on_locale_changed()

Called for each element when the locale is changed on their UIManager

rebuild()

Re-render the text to the label's underlying sprite image. This allows us to change what the displayed text is or remake it with different theming (if the theming has changed).

rebuild_from_changed_theme_data()

Checks if any theming parameters have changed, and if so triggers a full rebuild of the element.

set_active_effect(*effect_type:* UITextEffectType | *None*, *params:* *Dict[str, Any]* | *None* = *None*, *effect_tag:* *str* | *None* = *None*)

Set an animation effect to run on the text box. The effect will start running immediately after this call.

These effects are currently supported:

- TEXT_EFFECT_TYPING_APPEAR - Will look as if the text is being typed in.
- TEXT_EFFECT_FADE_IN - The text will fade in from the background colour.
- TEXT_EFFECT_FADE_OUT - The text will fade out to the background colour.

Parameters

- **effect_tag** -- if not None, only apply the effect to chunks with this tag.
- **params** -- Any parameters for the effect you are setting, if none are set defaults will be used.
- **effect_type** -- The type of the effect to set. If set to None instead it will cancel any active effect.

set_text(*text:* *str*, *, *text_kwargs:* *Dict[str, str]* | *None* = *None*)

Changes the string displayed by the label element. Labels do not support HTML styling.

Parameters

- **text** -- the text to set the label to.
- **text_kwargs** -- a dictionary of variable arguments to pass to the translated string useful when you have multiple translations that need variables inserted in the middle.

set_text_alpha(*alpha:* *int*, *sub_chunk:* TextLineChunkFTFont | *None* = *None*)

Set the global alpha value for the text

Parameters

- **alpha** -- the alpha to set.
- **sub_chunk** -- An optional chunk so we only set the alpha for this chunk.

set_text_offset_pos(*offset: Tuple[int, int]*, *sub_chunk:* TextLineChunkFTFont | *None* = *None*)

Move the text around by this offset.

Parameters

- **offset** -- the offset to set
- **sub_chunk** -- An optional chunk so we only set the offset for this chunk.

Returns

set_text_rotation(*rotation: int*, *sub_chunk:* TextLineChunkFTFont | *None* = *None*)

rotate the text by this int in degrees

Parameters

- **rotation** -- the rotation to set
- **sub_chunk** -- An optional chunk so we only set the rotation for this chunk.

Returns

set_text_scale(*scale: int*, *sub_chunk:* TextLineChunkFTFont | *None* = *None*)

Scale the text by this float

Parameters

- **scale** -- the scale to set
- **sub_chunk** -- An optional chunk so we only set the rotation for this chunk.

Returns

stop_finished_effect(*sub_chunk:* TextLineChunkFTFont | *None* = *None*)

Stops a finished effect. Will leave effected text in the state it was in when effect ended. Used when an effect reaches a natural end where we might want to keep it in the end of effect state (e.g. a fade out)

Parameters
sub_chunk -- An optional chunk so we only clear the effect from this chunk.

update(*time_delta: float*)

Called once every update loop of the UI Manager.

Parameters
time_delta -- The time in seconds between calls to update. Useful for timing things.

update_text_effect(*time_delta: float*)

Update any active text effect on the text owner

Parameters
time_delta -- the time delta in seconds

update_text_end_position(*end_pos: int*, *sub_chunk:* TextLineChunkFTFont | *None* = *None*)

The position in the text to render up to.

Parameters

- **end_pos** -- The current end position as an int
- **sub_chunk** -- An optional chunk to restrict the end_position to only this chunk.

pygame_gui.elements.ui_panel module

class pygame_gui.elements.ui_panel.**UIPanel**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *starting_height: int = 1*, *manager: IUIManagerInterface | None = None*, ***, *element_id: str = 'panel'*, *margins: Dict[str, int] | None = None*, *container: IContainerLikeInterface | None = None*, *parent_element: UIElement | None = None*, *object_id: ObjectID | str | None = None*, *anchors: Dict[str, str | UIElement] | None = None*, *visible: int = 1*)

Bases: *UIElement*, *IContainerLikeInterface*

A rectangular panel that holds a UI container and is designed to overlap other elements. It acts a little like a window that is not shuffled about in a stack - instead remaining at the same layer distance from the container it was initially placed in.

It's primary purpose is for things like involved HUDs in games that want to always sit on top of UI elements that may be present 'inside' the game world (e.g. player health bars). By creating a UI Panel at a height above the highest layer used by the game world's UI elements we can ensure that all elements added to the panel are always above the fray.

Parameters

- **relative_rect** -- The positioning and sizing rectangle for the panel. See the layout guide for details.
- **starting_height** -- How many layers above its container to place this panel on.
- **manager** -- The GUI manager that handles drawing and updating the UI and interactions between elements. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **margins** -- Controls the distance between the edge of the panel and where it's container should begin.
- **container** -- The container this panel is inside - distinct from this panel's own container.
- **parent_element** -- A hierarchical 'parent' used for signifying belonging and used in theming and events.
- **object_id** -- An identifier that can be used to help distinguish this particular panel from others.
- **anchors** -- Used to layout elements and dictate what the relative_rect is relative to. Defaults to the top left.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.

are_contents_hovered() → bool

Are any of the elements in the container hovered? Used for handling mousewheel events.

Returns

True if one of the elements is hovered, False otherwise.

disable()

Disables all elements in the panel, so they are no longer interactive.

enable()

Enables all elements in the panel, so they are interactive again.

`get_container`() → *IUIContainerInterface*

Returns the container that should contain all the UI elements in this panel.

Return UIContainer
The panel's container.

`hide`()

In addition to the base UIElement.hide() - call hide() of owned container - panel_container.

`kill`()

Overrides the basic kill() method of a pygame sprite so that we also kill all the UI elements in this panel.

`process_event`(*event: Event*) → bool

Can be overridden, also handle resizing windows. Gives UI Windows access to pygame events. Currently just blocks mouse click down events from passing through the panel.

Parameters
`event` -- The event to process.

Returns
Should return True if this element consumes this event.

`rebuild`()

A complete rebuild of the drawable shape used by this button.

`rebuild_from_changed_theme_data`()

Checks if any theming parameters have changed, and if so triggers a full rebuild of the button's drawable shape.

`set_dimensions`(*dimensions: Vector2 | Tuple[float, float]*, *clamp_to_container: bool = False*)

Set the size of this panel and then re-sizes and shifts the contents of the panel container to fit the new size.

Parameters

- `dimensions` -- The new dimensions to set.
- `clamp_to_container` -- Whether we should clamp the dimensions to the dimensions of the container or not.

`set_position`(*position: Vector2 | Tuple[float, float]*)

Method to directly set the absolute screen rect position of an element.

Parameters
`position` -- The new position to set.

`set_relative_position`(*position: Vector2 | Tuple[float, float]*)

Method to directly set the relative rect position of an element.

Parameters
`position` -- The new position to set.

`show`()

In addition to the base UIElement.show() - call show() of owned container - panel_container.

`update`(*time_delta: float*)

A method called every update cycle of our application. Designed to be overridden by derived classes but also has a little functionality to make sure the panel's layer 'thickness' is accurate and to handle window resizing.

Parameters
`time_delta` -- time passed in seconds between one call to this method and the next.

pygame_gui.elements.ui_progress_bar module

class pygame_gui.elements.ui_progress_bar.**UIProgressBar**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *manager:* IUIManagerInterface | *None* = *None*, *container:* IContainerLikeInterface | *None* = *None*, *parent_element:* UIElement | *None* = *None*, *object_id:* ObjectID | *str* | *None* = *None*, *anchors: Dict[str, str* | UIElement*]* | *None* = *None*, *visible: int = 1*)

Bases: *UIStatusBar*

A UI that will display a progress bar from 0 to 100%

Parameters

- **relative_rect** -- The rectangle that defines the size and position of the progress bar.
- **manager** -- The UIManager that manages this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **container** -- The container that this element is within. If not provided or set to None will be the root window's container.
- **parent_element** -- The element this element 'belongs to' in the theming hierarchy.
- **object_id** -- A custom defined ID for fine-tuning of theming.
- **anchors** -- A dictionary describing what this element's relative_rect is relative to.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.

status_text()

Subclass and override this method to change what text is displayed, or to suppress the text.

pygame_gui.elements.ui_screen_space_health_bar module

class pygame_gui.elements.ui_screen_space_health_bar.**UIScreenSpaceHealthBar**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *manager: IUIManagerInterface | None = None*, *sprite_to_monitor: SpriteWithHealth | None = None*, *container: IContainerLikeInterface | None = None*, *parent_element: UIElement | None = None*, *object_id: ObjectID | str | None = None*, *anchors: Dict[str, str | UIElement] | None = None*, *visible: int = 1*)

Bases: *UIStatusBar*

A UI that will display health capacity and current health for a sprite in 'screen space'. That means it won't move with the camera. This is a good choice for a user/player sprite.

Parameters

- **relative_rect** -- The rectangle that defines the size and position of the health bar.
- **manager** -- The UIManager that manages this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **sprite_to_monitor** -- The sprite we are displaying the health of.
- **container** -- The container that this element is within. If set to None will be the root window's container.
- **parent_element** -- The element this element 'belongs to' in the theming hierarchy.
- **object_id** -- A custom defined ID for fine-tuning of theming.
- **anchors** -- A dictionary describing what this element's relative_rect is relative to.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.

status_text()

Subclass and override this method to change what text is displayed, or to suppress the text.

pygame_gui.elements.ui_scrolling_container module

class pygame_gui.elements.ui_scrolling_container.**UIScrollingContainer**(*relative_rect: Rect*, *manager:* IUIManagerInterface | *None* = *None*, ***, *starting_height: int* = *1*, *container:* IContainerLikeInterface | *None* = *None*, *parent_element:* UIElement | *None* = *None*, *object_id:* ObjectID | *str* | *None* = *None*, *element_id: List[str]* | *None* = *None*, *anchors: Dict[str, str* | UIElement*]* | *None* = *None*, *visible: int* = *1*, *should_grow_automatically: bool* = *True*, *allow_scroll_x: bool* = *True*, *allow_scroll_y: bool* = *True*)

Bases: *UIElement*, *IContainerLikeInterface*

A container like UI element that lets users scroll around a larger container of content with scroll bars.

Parameters

- **relative_rect** -- The size and relative position of the container. This will also be the starting size of the scrolling area.
- **manager** -- The UI manager for this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **starting_height** -- The starting layer height of this container above its container. Defaults to 1.
- **container** -- The container this container is within. Defaults to None (which is the root container for the UI)
- **parent_element** -- A parent element for this container. Defaults to None, or the container if you've set that.
- **object_id** -- An object ID for this element.
- **anchors** -- Layout anchors in a dictionary.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.
- **allow_scroll_x** -- Whether a scrollbar should be added to scroll horizontally (when needed). Defaults to True.
- **allow_scroll_y** -- Whether a scrollbar should be added to scroll vertically (when needed). Defaults to True.

are_contents_hovered() → bool

Are any of the elements in the container hovered? Used for handling mousewheel events.

Returns
True if one of the elements is hovered, False otherwise.

`disable()`

Disables all elements in the container, so they are no longer interactive.

`enable()`

Enables all elements in the container, so they are interactive again.

`get_container()` → *IUIContainerInterface*

Gets the scrollable container area (the one that moves around with the scrollbars) from this container-like UI element.

Returns
the scrolling container.

`hide()`

In addition to the base UIElement.hide() - call hide() of owned container - _root_container. All other sub-elements (view_container, scrollbars) are children of _root_container, so it's visibility will propagate to them - there is no need to call their hide() methods separately.

`kill()`

Overrides the basic kill() method of a pygame sprite so that we also kill all the UI elements in this panel.

`set_dimensions`(*dimensions: Vector2 | Tuple[float, float]*, *clamp_to_container: bool = False*)

Method to directly set the dimensions of an element.

NOTE: Using this on elements inside containers with non-default anchoring arrangements may make a mess of them.

Parameters

- **dimensions** -- The new dimensions to set.
- **clamp_to_container** -- Whether we should clamp the dimensions to the dimensions of the container or not.

`set_position`(*position: Vector2 | Tuple[float, float]*)

Method to directly set the absolute screen rect position of an element.

Parameters
position -- The new position to set.

`set_relative_position`(*position: Vector2 | Tuple[float, float]*)

Method to directly set the relative rect position of an element.

Parameters
position -- The new position to set.

`set_scrollable_area_dimensions`(*dimensions: Vector2 | Tuple[float, float]*)

Set the size of the scrollable area container. It starts the same size as the view container, but often you want to expand it, or why have a scrollable container?

Parameters
dimensions -- The new dimensions.

`show()`

In addition to the base UIElement.show() - call show() of owned container - _root_container. All other sub-elements (view_container, scrollbars) are children of _root_container, so it's visibility will propagate to them - there is no need to call their show() methods separately.

update(*time_delta: float*)

Updates the scrolling container's position based upon the scroll bars and updates the scrollbar's visible percentage as well if that has changed.

Parameters

time_delta -- The time passed between frames, measured in seconds.

pygame_gui.elements.ui_selection_list module

class pygame_gui.elements.ui_selection_list.**UISelectionList**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *item_list: List[str] | List[Tuple[str, str]]*, *manager: IUIManagerInterface | None = None*, ***, *allow_multi_select: bool = False*, *allow_double_clicks: bool = True*, *container: IContainerLikeInterface | None = None*, *starting_height: int = 1*, *parent_element: UIElement | None = None*, *object_id: ObjectID | str | None = None*, *anchors: Dict[str, str | UIElement] | None = None*, *visible: int = 1*, *default_selection: str | Tuple[str, str] | List[str] | List[Tuple[str, str]] | None = None*)

Bases: *UIElement*

A rectangular element that holds any number of selectable text items displayed as a list.

Parameters

- **relative_rect** -- The positioning and sizing rectangle for the panel. See the layout guide for details.
- **item_list** -- A list of items as strings (item name only), or tuples of two strings (name, theme_object_id).
- **default_selection** -- Default item(s) that should be selected: a string or a (str, str) tuple for single-selection lists or a list of strings or list of (str, str) tuples for multi-selection lists.
- **manager** -- The GUI manager that handles drawing and updating the UI and interactions between elements. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **allow_multi_select** -- True if we are allowed to pick multiple things from the selection list.
- **allow_double_clicks** -- True if we can double-click on items in the selection list.
- **container** -- The container this element is inside (by default the root container) distinct from this panel's container.
- **starting_height** -- The starting height up from its container where this list is placed into a layer.
- **parent_element** -- A hierarchical 'parent' used for signifying belonging and used in theming and events.

- **object_id** -- An identifier that can be used to help distinguish this particular element from others with the same hierarchy.
- **anchors** -- Used to layout elements and dictate what the relative_rect is relative to. Defaults to the top left.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.

add_items(*new_items: List[str] | List[Tuple[str, str]]*) → None

Add any number of new items to the selection list. Uses the same format as when the list is first created.

Parameters
new_items -- the list of new items to add

disable()

Disables all elements in the selection list, so they are no longer interactive.

enable()

Enables all elements in the selection list, so they are interactive again.

get_multi_selection(*include_object_id: bool = False*) → List[str] | List[Tuple[str, str]]

Get all the selected items in our selection list. Only works if this is a multi-selection list.

Parameters
include_object_id -- if True adds the object id of this list item to the returned list of Tuples. If False we just get a list of the visible text strings only.

Returns
A list of the selected items in our selection list. May be empty if nothing selected.

get_single_selection(*include_object_id: bool = False*) → Tuple[str, str] | str | None

Get the selected item in a list, if any. Only works if this is a single-selection list.

Parameters
include_object_id -- if True adds the object id of this list item to the returned list of Tuples. If False we just get a list of the visible text strings only.

Returns
A single item name as a string or None.

get_single_selection_start_percentage()

The percentage through the height of the list where the top of the first selected option is.

hide()

In addition to the base UIElement.hide() - call hide() of owned container - list_and_scroll_bar_container. All other sub-elements (item_list_container, scrollbar) are children of list_and_scroll_bar_container, so it's visibility will propagate to them - there is no need to call their hide() methods separately.

kill()

Overrides the basic kill() method of a pygame sprite so that we also kill all the UI elements in this panel.

process_event(*event: Event*) → bool

Can be overridden, also handle resizing windows. Gives UI Windows access to pygame events. Currently just blocks mouse click down events from passing through the panel.

Parameters
event -- The event to process.

Returns
Should return True if this element makes use of this event.

rebuild()

A complete rebuild of the drawable shape used by this element.

rebuild_from_changed_theme_data()

Checks if any theming parameters have changed, and if so triggers a full rebuild of the button's drawable shape

remove_items(*items_to_remove: List[str] | List[Tuple[str, str]]*) → None

Will remove all instances of the items provided. The full tuple is required for items with a display name and an object ID.

Parameters

items_to_remove -- The list of new options to remove.

set_dimensions(*dimensions: Vector2 | Tuple[float, float]*, *clamp_to_container: bool = False*)

Set the size of this panel and then resizes and shifts the contents of the panel container to fit the new size.

Parameters

- **dimensions** -- The new dimensions to set.
- **clamp_to_container** -- clamp these dimensions to the size of the element's container.

set_item_list(*new_item_list: List[str] | List[Tuple[str, str]]*)

Set a new string list (or tuple of strings & ids list) as the item list for this selection list. This will change what is displayed in the list.

Tuples should be arranged like so:

(list_text, object_ID)

- list_text: displayed in the UI
- object_ID: used for theming and events

Parameters

new_item_list -- The new list to switch to. Can be a list of strings or tuples.

set_position(*position: Vector2 | Tuple[float, float]*)

Sets the absolute screen position of this slider, updating all subordinate button elements at the same time.

Parameters

position -- The absolute screen position to set.

set_relative_position(*position: Vector2 | Tuple[float, float]*)

Method to directly set the relative rect position of an element.

Parameters

position -- The new position to set.

show()

In addition to the base UIElement.show() - call show() of owned container - list_and_scroll_bar_container. All other sub-elements (item_list_container, scrollbar) are children of list_and_scroll_bar_container, so it's visibility will propagate to them - there is no need to call their show() methods separately.

update(*time_delta: float*)

A method called every update cycle of our application. Designed to be overridden by derived classes but also has a little functionality to make sure the panel's layer 'thickness' is accurate and to handle window resizing.

Parameters
time_delta -- time passed in seconds between one call to this method and the next.

pygame_gui.elements.ui_status_bar module

class pygame_gui.elements.ui_status_bar.**UIStatusBar**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *manager: IUIManagerInterface | None = None*, *sprite: SpriteWithHealth | None = None*, *follow_sprite: bool = True*, *percent_method: Callable[[], float] | None = None*, *container: IContainerLikeInterface | None = None*, *parent_element: UIElement | None = None*, *object_id: ObjectID | str | None = None*, *anchors: Dict[str, str | UIElement] | None = None*, *visible: int = 1*)

Bases: *UIElement*

Displays a status/progress bar.

This is a flexible class that can be used to display status for a sprite (health/mana/fatigue, etc.), or to provide a status bar on the screen not attached to any particular object. You can use multiple status bars for a sprite to show different status items if desired.

You can use the percent_full attribute to manually set the status, or you can provide a pointer to a method that will provide the percentage information.

This is a kitchen sink class with several ways to use it; you may want to look at the subclasses built on top of it that are designed to be simpler to use, such as UIProgressBar, UIWorldSpaceHealthBar, and UIScreenSpaceHealthBar.

Parameters

- **relative_rect** -- The rectangle that defines the size of the health bar.
- **sprite** -- Optional sprite to monitor for status info, and for drawing the bar with the sprite.
- **follow_sprite** -- If there's a sprite, this indicates whether the bar should be drawn at the sprite's location.
- **percent_method** -- Optional method signature to call to get the percent complete. (To provide a method signature, simply reference the method without parenthesis, such as self.health_percent.)
- **manager** -- The UIManager that manages this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **container** -- The container that this element is within. If set to None will be the root window's container.
- **parent_element** -- The element this element 'belongs to' in the theming hierarchy.
- **object_id** -- A custom defined ID for fine-tuning of theming.
- **anchors** -- A dictionary describing what this element's relative_rect is relative to.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.

property **percent_full**

Use this property to directly change the status bar.

rebuild()

Rebuild the status bar entirely because the theming data has changed.

rebuild_from_changed_theme_data()

Called by the UIManager to check the theming data and rebuild whatever needs rebuilding for this element when the theme data has changed.

redraw()

Redraw the status bar when something, other than it's position has changed.

status_text()

To display text in the bar, subclass UIStatusBar and override this method.

update(*time_delta: float*)

Updates the status bar sprite's image and rectangle with the latest status and position data from the sprite we are monitoring

Parameters
time_delta -- time passed in seconds between one call to this method and the next.

pygame_gui.elements.ui_text_box module

class pygame_gui.elements.ui_text_box.**UITextBox**(*html_text: str*, *relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *manager: IUIManagerInterface | None = None*, *wrap_to_height: bool = False*, *starting_height: int = 1*, *container: IContainerLikeInterface | None = None*, *parent_element: UIElement | None = None*, *object_id: ObjectID | str | None = None*, *anchors: Dict[str, str | UIElement] | None = None*, *visible: int = 1*, ***, *pre_parsing_enabled: bool = True*, *text_kwargs: Dict[str, str] | None = None*, *allow_split_dashes: bool = True*, *plain_text_display_only: bool = False*, *should_html_unescape_input_text: bool = False*, *placeholder_text: str | None = None*)

Bases: *UIElement*, *IUITextOwnerInterface*

A Text Box element lets us display word-wrapped, formatted text. If the text to display is longer than the height of the box given then the element will automatically create a vertical scroll bar so that all the text can be seen.

Formatting the text is done via a subset of HTML tags. Currently supported tags are:

- or - to encase bold styled text.
- , or <var></var> - to encase italic styled text.
- <u></u> - to encase underlined text.
- ** - to encase 'link' text that can be clicked on to generate events with the** id given in href.
- <body bgcolor='#FFFFFF'></body> - to change the background colour of encased text.
-
 or - to start a new line.
- ** - To set the font, colour and size of** encased text.
- <hr> a horizontal rule.

- **<effect id='custom_id'></effect> - to encase text to that will have an effect applied. The custom id**
  supplied is used when setting an effect on this text box.
- **<shadow size=1 offset=1,1 color=#808080></shadow> - puts a shadow behind the text encased. Can also be used to**
  make a glow effect.
- **<img src="data/images/test_images/london.jpg" float=right> Inserts an image into the text with options on how to**
  float the text around the image. Float options are left, right or none.

More may be added in the future if needed or frequently requested.

NOTE: if dimensions of the initial containing rect are set to -1 the text box will match the final dimension to whatever the text rendering produces. This lets us make dynamically sized text boxes depending on their contents.

Parameters

- `html_text` -- The HTML formatted text to display in this text box.
- `relative_rect` -- The 'visible area' rectangle, positioned relative to its container.
- `manager` -- The UIManager that manages this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- `wrap_to_height` -- False by default, if set to True the box will increase in height to match the text within.
- `starting_height` -- Sets the height, above its container, to start placing the text box at.
- `container` -- The container that this element is within. If not provided or set to None will be the root window's container.
- `parent_element` -- The element this element 'belongs to' in the theming hierarchy.
- `object_id` -- A custom defined ID for fine-tuning of theming.
- `anchors` -- A dictionary describing what this element's relative_rect is relative to.
- `visible` -- Whether the element is visible by default. Warning - container visibility may override this.
- `pre_parsing_enabled` -- When enabled will replace all newline characters with html
 tags.
- `text_kwargs` -- A dictionary of variable arguments to pass to the translated text useful when you have multiple translations that need variables inserted in the middle.
- `allow_split_dashes` -- Sets whether long words that don't fit on a single line will be split with a dash or just split without a dash (more compact).
- `plain_text_display_only` -- No markup based styling & formatting will be done on the input text.
- `should_html_unescape_input_text` -- When enabled turns plain text encoded html back into html for this text box. e.g. < will become '<'
- `placeholder_text` -- If the text line is empty, and not focused, this placeholder text will be shown instead.

`append_html_text`(*new_html_str: str*)

Adds a string, that is parsed for any HTML tags that pygame_gui supports, onto the bottom of the text box's current contents.

This is useful for making things like logs.

Parameters
new_html_str -- The, potentially HTML tag, containing string of text to append.

clear_all_active_effects(*sub_chunk:* TextLineChunkFTFont | *None* = *None*)

Clears any active effects and redraws the text. A full reset, usually called before firing off a new effect if one is already in progress.

Parameters
sub_chunk -- An optional chunk so we only clear the effect from this chunk.

clear_text_surface(*sub_chunk:* TextLineChunkFTFont | *None* = *None*)

Clear the text surface

Parameters
sub_chunk -- An optional chunk so we only clear the surface for this chunk.

disable()

Disable the text box. Basically just disables the scroll bar if one exists.

enable()

Enable the text box. Re-enables the scroll bar if one exists.

focus()

Called when we 'select focus' on this element.

full_redraw()

Trigger a full redraw of the entire text box. Useful if we have messed with the text chunks in a more fundamental fashion and need to reposition them (say, if some of them have gotten wider after being made bold).

NOTE: This doesn't reparse the text of our box. If you need to do that, just create a new text box.

get_object_id() → str

The UI object ID of this text owner for use in effect events.

Returns
the ID string

get_text_letter_count(*sub_chunk:* TextLineChunkFTFont | *None* = *None*) → int

The amount of letters in the text

Parameters
sub_chunk -- An optional chunk to restrict the count to only this chunk.

Returns
number of letters as an int

hide()

In addition to the base UIElement.hide() - call hide() of scroll_bar if it exists.

kill()

Overrides the standard sprite kill method to also kill any scroll bars belonging to this text box.

on_fresh_drawable_shape_ready()

Called by an element's drawable shape when it has a new image surface ready for use, normally after a rebuilding/redrawing of some kind.

on_locale_changed()

Called for each element when the locale is changed on their UIManager

parse_html_into_style_data()

Parses HTML styled string text into a format more useful for styling rendered text.

process_event(*event: Event*) → bool

A stub to override. Gives UI Elements access to pygame events.

Parameters
event -- The event to process.

Returns
Should return True if this element makes use of this event.

rebuild()

Rebuild whatever needs building.

rebuild_from_changed_theme_data()

Called by the UIManager to check the theming data and rebuild whatever needs rebuilding for this element when the theme data has changed.

redraw_from_chunks()

Redraws from slightly earlier in the process than 'redraw_from_text_block'. Useful if we have redrawn individual chunks already (say, to change their style slightly after being hovered) and now want to update the text block with those changes without doing a full redraw.

This won't work very well if redrawing a chunk changed its dimensions.

redraw_from_text_block()

Redraws the final parts of the text box element that don't include redrawing the actual text. Useful if we've just moved the position of the text (say, with a scroll bar) without actually changing the text itself.

property select_range

The selected range for this text. A tuple containing the start and end indexes of the current selection.

Made into a property to keep it synchronised with the underlying drawable shape's representation.

set_active_effect(*effect_type: UITextEffectType | None = None*, *params: Dict[str, Any] | None = None*, *effect_tag: str | None = None*)

Set an animation effect to run on the text box. The effect will start running immediately after this call.

These effects are currently supported:

- TEXT_EFFECT_TYPING_APPEAR - Will look as if the text is being typed in.
- TEXT_EFFECT_FADE_IN - The text will fade in from the background colour.
- TEXT_EFFECT_FADE_OUT - The text will fade out to the background colour.

Parameters

- **effect_tag** -- if not None, only apply the effect to chunks with this tag.
- **params** -- Any parameters for the effect you are setting, if none are set defaults will be used.
- **effect_type** -- The type of the effect to set. If set to None instead it will cancel any active effect.

set_dimensions(*dimensions: Vector2 | Tuple[float, float]*, *clamp_to_container: bool = False*)

Method to directly set the dimensions of a text box.

Parameters

- **dimensions** -- The new dimensions to set.
- **clamp_to_container** -- Whether we should clamp the dimensions to the dimensions of the container or not.

set_position(*position: Vector2 | Tuple[float, float]*)

Sets the absolute screen position of this text box, updating its subordinate scroll bar at the same time.

Parameters

position -- The absolute screen position to set.

set_relative_position(*position: Vector2 | Tuple[float, float]*)

Sets the relative screen position of this text box, updating its subordinate scroll bar at the same time.

Parameters

position -- The relative screen position to set.

set_text_alpha(*alpha: int*, *sub_chunk:* TextLineChunkFTFont | *None* = *None*)

Set the global alpha value for the text

Parameters

- **alpha** -- the alpha to set.
- **sub_chunk** -- An optional chunk so we only set the alpha for this chunk.

set_text_offset_pos(*offset: Tuple[int, int]*, *sub_chunk:* TextLineChunkFTFont | *None* = *None*)

Move the text around by this offset.

Parameters

- **offset** -- the offset to set
- **sub_chunk** -- An optional chunk so we only set the offset for this chunk.

Returns

set_text_rotation(*rotation: int*, *sub_chunk:* TextLineChunkFTFont | *None* = *None*)

rotate the text by this int in degrees

Parameters

- **rotation** -- the rotation to set
- **sub_chunk** -- An optional chunk so we only set the rotation for this chunk.

Returns

set_text_scale(*scale: float*, *sub_chunk:* TextLineChunkFTFont | *None* = *None*)

Scale the text by this float

Parameters

- **scale** -- the scale to set
- **sub_chunk** -- An optional chunk so we only set the rotation for this chunk.

Returns

show()

In addition to the base UIElement.show() - call show() of scroll_bar if it exists.

stop_finished_effect(*sub_chunk:* TextLineChunkFTFont | *None* = *None*)

Stops a finished effect. Will leave effected text in the state it was in when effect ended. Used when an effect reaches a natural end where we might want to keep it in the end of effect state (e.g. a fade out)

Parameters
sub_chunk -- An optional chunk so we only clear the effect from this chunk.

unfocus()

Called when this element is no longer the current focus.

update(*time_delta: float*)

Called once every update loop of the UI Manager. Used to react to scroll bar movement (if there is one), update the text effect (if there is one) and check if we are hovering over any text links (if there are any).

Parameters
time_delta -- The time in seconds between calls to update. Useful for timing things.

update_text_effect(*time_delta: float*)

Update any active text effect on the text owner

Parameters
time_delta -- the time delta in seconds

update_text_end_position(*end_pos: int*, *sub_chunk:* TextLineChunkFTFont | *None* = *None*)

The position in the text to render up to.

Parameters

- **end_pos** -- The current end position as an int
- **sub_chunk** -- An optional chunk to restrict the end_position to only this chunk.

pygame_gui.elements.ui_text_entry_box module

class pygame_gui.elements.ui_text_entry_box.**UITextEntryBox**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *initial_text: str = ''*, *manager:* IUIManagerInterface | *None* = *None*, *container:* IContainerLikeInterface | *None* = *None*, *parent_element:* UIElement | *None* = *None*, *object_id:* ObjectID | *str* | *None* = *None*, *anchors: Dict[str, str |* UIElement*]* | *None* = *None*, *visible: int = 1*, ***, *placeholder_text: str* | *None* = *None*)

Bases: *UITextBox*

Inherits from UITextBox but allows you to enter text with the keyboard, much like UITextEntryLine.

Parameters

- **relative_rect** -- The 'visible area' rectangle, positioned relative to its container.
- **initial_text** -- The text that starts in the text box.
- **manager** -- The UIManager that manages this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **container** -- The container that this element is within. If not provided or set to None will be the root window's container.

- **parent_element** -- The element this element 'belongs to' in the theming hierarchy.
- **object_id** -- A custom defined ID for fine-tuning of theming.
- **anchors** -- A dictionary describing what this element's relative_rect is relative to.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.

Param
placeholder_text: If the text line is empty, and not focused, this placeholder text will be shown instead.

focus()

Called when we 'select focus' on this element. In this case it sets up the keyboard to repeat held key presses, useful for natural feeling keyboard input.

get_text() → str

Gets the text in the entry box element.

Returns
A string.

process_event(*event: Event*) → bool

Allows the text entry box to react to input events, which is its primary function. The entry element reacts to various types of mouse clicks (double click selecting words, drag select), keyboard combos (CTRL+C, CTRL+V, CTRL+X, CTRL+A), individual editing keys (Backspace, Delete, Left & Right arrows) and other keys for inputting letters, symbols and numbers.

Parameters
event -- The current event to consider reacting to.

Returns
Returns True if we've done something with the input event.

redraw_from_text_block()

Redraws the final parts of the text box element that don't include redrawing the actual text. Useful if we've just moved the position of the text (say, with a scroll bar) without actually changing the text itself.

unfocus()

Called when this element is no longer the current focus.

update(*time_delta: float*)

Called every update loop of our UI Manager. Largely handles text drag selection and making sure our edit cursor blinks on and off.

Parameters
time_delta -- The time in seconds between this update method call and the previous one.

pygame_gui.elements.ui_text_entry_line module

class pygame_gui.elements.ui_text_entry_line.**UITextEntryLine**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *manager:* IUIManagerInterface | *None* = *None*, *container:* IContainerLikeInterface | *None* = *None*, *parent_element:* UIElement | *None* = *None*, *object_id:* ObjectID | *str* | *None* = *None*, *anchors: Dict[str, str* | UIElement*]* | *None* = *None*, *visible: int = 1*, *, *initial_text: str* | *None* = *None*, *placeholder_text: str* | *None* = *None*)

Bases: *UIElement*

A GUI element for text entry from a keyboard, on a single line. The element supports the standard copy and paste keyboard shortcuts CTRL+V, CTRL+C & CTRL+X as well as CTRL+A.

There are methods that allow the entry element to restrict the characters that can be input into the text box

The height of the text entry line element will be determined by the font used rather than the standard method for UIElements of just using the height of the input rectangle.

Parameters

- **relative_rect** -- A rectangle describing the position and width of the text entry element.
- **manager** -- The UIManager that manages this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **container** -- The container that this element is within. If not provided or set to None will be the root window's container.
- **parent_element** -- The element this element 'belongs to' in the theming hierarchy.
- **object_id** -- A custom defined ID for fine-tuning of theming.
- **anchors** -- A dictionary describing what this element's relative_rect is relative to.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.

Param
initial_text: A string that will start in this box. This text will remain when box is focused for editing.

Param
placeholder_text: If the text line is empty, and not focused, this placeholder text will be shown instead.

disable()

Disables the element so that it is no longer interactive.

enable()

Re-enables the element, so we can once again interact with it.

focus()

Called when we 'select focus' on this element. In this case it sets up the keyboard to repeat held key presses, useful for natural feeling keyboard input.

get_text() → str

Gets the text in the entry line element.

Returns
A string.

`on_locale_changed`()

Called for each element when the locale is changed on their UIManager

`process_event`(*event: Event*) → bool

Allows the text entry box to react to input events, which is its primary function. The entry element reacts to various types of mouse clicks (double click selecting words, drag select), keyboard combos (CTRL+C, CTRL+V, CTRL+X, CTRL+A), individual editing keys (Backspace, Delete, Left & Right arrows) and other keys for inputting letters, symbols and numbers.

Parameters
`event` -- The current event to consider reacting to.

Returns
Returns True if we've done something with the input event.

`rebuild`()

Rebuild whatever needs building.

`rebuild_from_changed_theme_data`()

Called by the UIManager to check the theming data and rebuild whatever needs rebuilding for this element when the theme data has changed.

`redraw`()

Redraws the entire text entry element onto the underlying sprite image. Usually called when the displayed text has been edited or changed in some fashion.

`property select_range`

The selected range for this text. A tuple containing the start and end indexes of the current selection.

Made into a property to keep it synchronised with the underlying drawable shape's representation.

`set_allowed_characters`(*allowed_characters: str | List[str]*)

Sets a whitelist of characters that will be the only ones allowed in our text entry element. We can either set the list directly, or request one of the already existing lists by a string identifier. The currently supported lists for allowed characters are:

- 'numbers'
- 'letters'
- 'alpha_numeric'

Parameters
`allowed_characters` -- The characters to allow, either in a list form or one of the supported string ids.

`set_forbidden_characters`(*forbidden_characters: str | List[str]*)

Sets a blacklist of characters that will be banned from our text entry element. We can either set the list directly, or request one of the already existing lists by a string identifier. The currently supported lists for forbidden characters are:

- 'numbers'
- 'forbidden_file_path'

Parameters
forbidden_characters -- The characters to forbid, either in a list form or one of the supported string ids.

`set_text`(*text: str*)

Allows the text displayed in the text entry element to be set via code. Useful for setting an initial or existing value that can be edited.

The string to set must be valid for the text entry element for this to work.

Parameters
text -- The text string to set.

`set_text_hidden`(*is_hidden=True*)

Passing in True will hide text typed into the text line, replacing it with characters and also disallow copying the text into the clipboard. It is designed for basic 'password box' usage.

Parameters
is_hidden -- Can be set to True or False. Defaults to True because if you are calling this you likely want a password box with no fuss. Set it back to False if you want to un-hide the text (e.g. for one of those 'Show my password' buttons).

`set_text_length_limit`(*limit: int*)

Allows a character limit to be set on the text entry element. By default, there is no limit on the number of characters that can be entered.

Parameters
limit -- The character limit as an integer.

`unfocus`()

Called when this element is no longer the current focus.

`update`(*time_delta: float*)

Called every update loop of our UI Manager. Largely handles text drag selection and making sure our edit cursor blinks on and off.

Parameters
time_delta -- The time in seconds between this update method call and the previous one.

`validate_text_string`(*text_to_validate: str*) → bool

Checks a string of text to see if any of its characters don't meet the requirements of the allowed and forbidden character sets.

Parameters
text_to_validate -- The text string to check.

pygame_gui.elements.ui_tool_tip module

class pygame_gui.elements.ui_tool_tip.**UITooltip**(*html_text: str*, *hover_distance: Tuple[int, int]*, *manager:* IUIManagerInterface | *None* = *None*, *parent_element:* IUIElementInterface | *None* = *None*, *object_id:* ObjectID | *str* | *None* = *None*, *anchors: Dict[str, str* | UIElement*]* | *None* = *None*, ***, *wrap_width: int* | *None* = *None*, *text_kwargs: Dict[str, str]* | *None* = *None*)

Bases: *UIElement*, *IUITooltipInterface*

A tool tip is a floating block of text that gives additional information after a user hovers over an interactive part of a GUI for a short time. In Pygame GUI the tooltip's text is style-able with HTML.

At the moment the tooltips are only available as an option on UIButton elements.

Tooltips also don't allow a container as they are designed to overlap normal UI boundaries and be contained only within the 'root' window/container, which is synonymous with the pygame display surface.

Parameters

- **html_text** -- Text styled with HTML, to be displayed on the tooltip.
- **hover_distance** -- Distance in pixels between the tooltip and the thing being hovered.
- **manager** -- The UIManager that manages this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **parent_element** -- The element this element 'belongs to' in the theming hierarchy.
- **object_id** -- A custom defined ID for fine-tuning of theming.
- **anchors** -- A dictionary describing what this element's relative_rect is relative to.
- **text_kwargs** -- a dictionary of variable arguments to pass to the translated text useful when you have multiple translations that need variables inserted in the middle.

find_valid_position(*position: Vector2 | Tuple[float, float]*) → bool

Finds a valid position for the tool tip inside the root container of the UI.

The algorithm starts from the position of the target we are providing a tool tip for then it tries to fit the rectangle for the tool tip onto the screen by moving it above, below, to the left and to the right, until we find a position that fits the whole tooltip rectangle on the screen at once.

If we fail to manage this then the method will return False. Otherwise, it returns True and set the position of the tool tip to our valid position.

Parameters
position -- A 2D vector representing the position of the target this tool tip is for.

Returns
returns True if we find a valid (visible) position and False if we do not.

hide()

This is a base method hide() of a UIElement, but since it's not intended to be used on a UIToolTip - display a warning.

kill()

Overrides the UIElement's default kill method to also kill the text block element that helps make up the complete tool tip.

rebuild()

Rebuild anything that might need rebuilding.

rebuild_from_changed_theme_data()

Called by the UIManager to check the theming data and rebuild whatever needs rebuilding for this element when the theme data has changed.

set_dimensions(*dimensions: Vector2 | Tuple[float, float]*, *clamp_to_container: bool = False*)

Directly sets the dimensions of this tool tip. This will overwrite the normal theming.

Parameters

- **dimensions** -- The new dimensions to set.

- **clamp_to_container** -- Whether we should clamp the dimensions to the dimensions of the container or not.

set_position(*position: Vector2 | Tuple[float, float]*)

Sets the absolute screen position of this tool tip, updating its subordinate text box at the same time.

Parameters
position -- The absolute screen position to set.

set_relative_position(*position: Vector2 | Tuple[float, float]*)

Sets the relative screen position of this tool tip, updating its subordinate text box at the same time.

Parameters
position -- The relative screen position to set.

show()

This is a base method show() of a UIElement, but since it's not intended to be used on a UIToolTip - display a warning.

pygame_gui.elements.ui_vertical_scroll_bar module

class pygame_gui.elements.ui_vertical_scroll_bar.**UIVerticalScrollBar**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *visible_percentage: float*, *manager: IUIManagerInterface | None = None*, *container: IContainerLikeInterface | None = None*, *parent_element: UIElement | None = None*, *object_id: ObjectID | str | None = None*, *anchors: Dict[str, str | UIElement] | None = None*, *visible: int = 1*)

Bases: *UIElement*

A vertical scroll bar allows users to position a smaller visible area within a vertically larger area.

Parameters

- **relative_rect** -- The size and position of the scroll bar.
- **visible_percentage** -- The vertical percentage of the larger area that is visible, between 0.0 and 1.0.
- **manager** -- The UIManager that manages this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **container** -- The container that this element is within. If not provided or set to None will be the root window's container.
- **parent_element** -- The element this element 'belongs to' in the theming hierarchy.
- **object_id** -- A custom defined ID for fine tuning of theming.
- **anchors** -- A dictionary describing what this element's relative_rect is relative to.

- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.

check_has_moved_recently() → bool

Returns True if the scroll bar was moved in the last call to the update function.

Returns
True if we've recently moved the scroll bar, False otherwise.

disable()

Disables the scroll bar so it is no longer interactive.

enable()

Enables the scroll bar so it is interactive once again.

hide()

In addition to the base UIElement.hide() - hide the self.button_container which will propagate and hide all the buttons.

kill()

Overrides the kill() method of the UI element class to kill all the buttons in the scroll bar and clear any of the parts of the scroll bar that are currently recorded as the 'last focused vertical scroll bar element' on the ui manager.

NOTE: the 'last focused' state on the UI manager is used so that the mouse wheel will move whichever scrollbar we last fiddled with even if we've been doing other stuff. This seems to be consistent with the most common mousewheel/scrollbar interactions used elsewhere.

process_event(*event: Event*) → bool

Checks an event from pygame's event queue to see if the scroll bar needs to react to it. In this case it is just mousewheel events, mainly because the buttons that make up the scroll bar will handle the required mouse click events.

Parameters
event -- The event to process.

Returns
Returns True if we've done something with the input event.

rebuild()

Rebuild anything that might need rebuilding.

rebuild_from_changed_theme_data()

Called by the UIManager to check the theming data and rebuild whatever needs rebuilding for this element when the theme data has changed.

redraw_scrollbar()

Redraws the 'scrollbar' portion of the whole UI element. Called when we change the visible percentage.

reset_scroll_position()

Reset the current scroll position back to the top.

set_dimensions(*dimensions: Vector2 | Tuple[float, float]*, *clamp_to_container: bool = False*)

Method to directly set the dimensions of an element.

Parameters

- **dimensions** -- The new dimensions to set.
- **clamp_to_container** -- Whether we should clamp the dimensions to the dimensions of the container or not.

set_position(*position: Vector2 | Tuple[float, float]*)

Sets the absolute screen position of this scroll bar, updating all subordinate button elements at the same time.

Parameters
position -- The absolute screen position to set.

set_relative_position(*position: Vector2 | Tuple[float, float]*)

Sets the relative screen position of this scroll bar, updating all subordinate button elements at the same time.

Parameters
position -- The relative screen position to set.

set_scroll_from_start_percentage(*new_start_percentage: float*)

Set the scroll bar's scrolling position from a percentage between 0.0 and 1.0.

Parameters
new_start_percentage -- the percentage to set.

set_visible_percentage(*percentage: float*)

Sets the percentage of the total 'scrollable area' that is currently visible. This will affect the size of the scrollbar and should be called if the vertical size of the 'scrollable area' or the vertical size of the visible area change.

Parameters
percentage -- A float between 0.0 and 1.0 representing the percentage that is visible.

show()

In addition to the base UIElement.show() - show the self.button_container which will propagate and show all the buttons.

property start_percentage

turning start_percentage into a property, so we can round it to mitigate floating point errors

update(*time_delta: float*)

Called once per update loop of our UI manager. Deals largely with moving the scroll bar and updating the resulting 'start_percentage' variable that is then used by other 'scrollable' UI elements to control the point they start drawing.

Reacts to presses of the up and down arrow buttons, movement of the mouse wheel and dragging of the scroll bar itself.

Parameters
time_delta -- A float, roughly representing the time in seconds between calls to this method.

pygame_gui.elements.ui_window module

class pygame_gui.elements.ui_window.**UIWindow**(*rect: Rect | FRect | Tuple[float, float, float, float]*, *manager: IUIManagerInterface | None = None*, *window_display_title: str = ''*, *element_id: List[str] | str | None = None*, *object_id: ObjectID | str | None = None*, *resizable: bool = False*, *visible: int = 1*, *draggable: bool = True*, ***, *ignore_shadow_for_initial_size_and_pos: bool = True*, *always_on_top: bool = False*)

Bases: *UIElement*, *IContainerLikeInterface*, *IWindowInterface*

A base class for window GUI elements, any windows should inherit from this class.

Parameters

- **rect** -- A rectangle, representing size and position of the window (including title bar, shadow and borders).
- **manager** -- The UIManager that manages this UIWindow. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **window_display_title** -- A string that will appear in the windows title bar if it has one.
- **element_id** -- An element ID for this window, if one is not supplied it defaults to 'window'.
- **object_id** -- An optional object ID for this window, useful for distinguishing different windows.
- **resizable** -- Whether this window is resizable or not, defaults to False.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.
- **draggable** -- Whether this window is draggable or not, defaults to True.

property always_on_top: bool

Whether the window is always above normal windows or not. :return:

are_contents_hovered() → bool

Are any of the elements in the container hovered? Used for handling mousewheel events.

Returns
True if one of the elements is hovered, False otherwise.

can_hover() → bool

Called to test if this window can be hovered.

change_layer(*new_layer: int*)

Move this window, and it's contents, to a new layer in the UI.

Parameters
new_layer -- The layer to move to.

check_clicked_inside_or_blocking(*event: Event*) → bool

A quick event check outside the normal event processing so that this window is brought to the front of the window stack if we click on any of the elements contained within it.

Parameters
event -- The event to check.

Returns
returns True if the event represents a click inside this window or the window is blocking.

check_hover(*time_delta: float*, *hovered_higher_element: bool*) → bool

For the window the only hovering we care about is the edges if this is a resizable window.

Parameters

- **time_delta** -- time passed in seconds between one call to this method and the next.
- **hovered_higher_element** -- Have we already hovered an element/window above this one?

disable()

Disables the window and it's contents so it is no longer interactive.

enable()

Enables the window and it's contents so it is interactive again.

get_container() → *IUIContainerInterface*

Returns the container that should contain all the UI elements in this window.

Return UIContainer
The window's container.

get_hovering_edge_id() → str

Gets the ID of the combination of edges we are hovering for use by the cursor system.

Returns
a string containing the edge combination ID (e.g. xy,yx,xl,xr,yt,yb)

get_layer_thickness() → int

The layer 'thickness' of this window/ :return: an integer

get_relative_mouse_pos()

Returns the current mouse position relative to the inside of this window.

If the cursor is outside the window contents area it returns None

Returns
tuple of relative mouse co-ords or None

get_top_layer() → int

Returns the 'highest' layer used by this window so that we can correctly place other windows on top of it.

Returns
The top layer for this window as a number (greater numbers are higher layers).

hide()

In addition to the base UIElement.hide() - hide the _window_root_container which will propagate and hide all the children.

kill()

Overrides the basic kill() method of a pygame sprite so that we also kill all the UI elements in this window, and remove if from the window stack.

on_close_window_button_pressed()

Override this method to call 'hide()' instead if you want to hide a window when the close button is pressed.

on_moved_to_front()

Called when a window is moved to the front of its stack.

process_event(*event: Event*) → bool

Handles resizing & closing windows. Gives UI Windows access to pygame events. Derived windows should super() call this class if they implement their own process_event method.

Parameters
event -- The event to process.

Return bool
Return True if this element should consume this event and not pass it to the rest of the UI.

rebuild()

Rebuilds the window when the theme has changed.

rebuild_from_changed_theme_data()

Called by the UIManager to check the theming data and rebuild whatever needs rebuilding for this element when the theme data has changed.

set_blocking(*state: bool*)

Sets whether this window being open should block clicks to the rest of the UI or not. Defaults to False.

Parameters
state -- True if this window should block mouse clicks.

set_dimensions(*dimensions: Vector2 | Tuple[float, float]*, *clamp_to_container: bool = False*)

Set the size of this window and then re-sizes and shifts the contents of the windows container to fit the new size.

Parameters

- **dimensions** -- The new dimensions to set.
- **clamp_to_container** -- Whether we should clamp the dimensions to the dimensions of the container or not.

set_display_title(*new_title: str*)

Set the title of the window.

Parameters
new_title -- The title to set.

set_position(*position: Vector2 | Tuple[float, float]*)

Method to directly set the absolute screen rect position of an element.

Parameters
position -- The new position to set.

set_relative_position(*position: Vector2 | Tuple[float, float]*)

Method to directly set the relative rect position of an element.

Parameters
position -- The new position to set.

should_use_window_edge_resize_cursor() → bool

Returns true if this window is in a state where we should display one of the resizing cursors

Returns
True if a resizing cursor is needed.

show()

In addition to the base UIElement.show() - show the _window_root_container which will propagate and show all the children.

update(*time_delta: float*)

A method called every update cycle of our application. Designed to be overridden by derived classes but also has a little functionality to make sure the window's layer 'thickness' is accurate and to handle window resizing.

Parameters
time_delta -- time passed in seconds between one call to this method and the next.

pygame_gui.elements.ui_world_space_health_bar module

class pygame_gui.elements.ui_world_space_health_bar.**UIWorldSpaceHealthBar**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *sprite_to_monitor: Sprite | ExampleHealthSprite*, *manager: IUIManagerInterface | None = None*, *container: IContainerLikeInterface | None = None*, *parent_element: UIElement | None = None*, *object_id: ObjectID | str | None = None*, *anchors: Dict[str, str | UIElement] | None = None*, *visible: int = 1*)

Bases: *UIStatusBar*

A UI that will display a sprite's 'health_capacity' and their 'current_health' in 'world space' above the sprite. This means that the health bar will move with the camera and the sprite itself.

A sprite passed to this class must have the attributes 'health_capacity' and 'current_health'.

Parameters

- **relative_rect** -- The rectangle that defines the size of the health bar.
- **sprite_to_monitor** -- The sprite we are displaying the health of.
- **manager** -- The UIManager that manages this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **container** -- The container that this element is within. If not provided or set to None will be the root window's container.
- **parent_element** -- The element this element 'belongs to' in the theming hierarchy.
- **object_id** -- A custom defined ID for fine-tuning of theming.
- **anchors** -- A dictionary describing what this element's relative_rect is relative to.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.

class **ExampleHealthSprite**(**groups*)

Bases: Sprite

An example sprite with health instance attributes.

Parameters

groups -- Sprite groups to put the sprite in.

Module contents

class pygame_gui.elements.**UI2DSlider**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *start_value_x: float | int*, *value_range_x: Tuple[float, float] | Tuple[int, int]*, *start_value_y: float | int*, *value_range_y: Tuple[float, float] | Tuple[int, int]*, *invert_y: bool = False*, *manager: IUIManagerInterface | None = None*, *container: IContainerLikeInterface | None = None*, *starting_height: int = 1*, *parent_element: UIElement | None = None*, *object_id: ObjectID | str | None = None*, *anchors: Dict[str, str | UIElement] | None = None*, *visible: int = 1*)

Bases: *UIElement*

A 2d slider is intended to help users adjust values within a range, for example a volume control.

Parameters

- **relative_rect** -- A rectangle describing the position and dimensions of the element.
- **start_value_x** -- The x value to start the slider at.
- **value_range_x** -- The full range of x values.
- **start_value_y** -- The y value to start the slider at.
- **value_range_y** -- The full range of y values.
- **invert_y** -- Should the y increase from bottom to top instead of the other way round?
- **manager** -- The UIManager that manages this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **container** -- The container that this element is within. If set to None will be the root window's container.
- **parent_element** -- The element this element "belongs to" in the theming hierarchy.
- **object_id** -- A custom defined ID for fine-tuning of theming.
- **anchors** -- A dictionary describing what this element's relative_rect is relative to.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.

disable()

Disable the slider. It should not be interactive and will use the disabled theme colours.

enable()

Enable the slider. It should become interactive and will use the normal theme colours.

get_current_value() → Tuple[float, int]

Gets the current value the slider is set to.

Returns

The current value recorded by the slider.

hide()

In addition to the base UIElement.hide() - hide the sliding button and hide the button_container which will propagate and hide the left and right buttons.

kill()

Overrides the normal sprite kill() method to also kill the button elements that help make up the slider.

rebuild()

Rebuild anything that might need rebuilding.

rebuild_from_changed_theme_data() → None

Called by the UIManager to check the theming data and rebuild whatever needs rebuilding for this element when the theme data has changed.

Returns
None

set_current_value(*value_x: float | int*, *value_y: float | int*, *warn: bool = True*) → None

Sets the value of the slider, which will move the position of the slider to match. Will issue a warning if the value set is not in the value range.

Parameters

- **value_x** -- The x value to set.
- **value_y** -- The y value to set.
- **warn** -- set to false in order to suppress the default warning, instead the value will be clamped.

Param
invert_y: Should the y value be inverted? If not passed then will use self.invert_value for this info.

Returns
None

set_dimensions(*dimensions: Vector2 | Tuple[float, float]*, *clamp_to_container: bool = False*)

Method to directly set the dimensions of an element.

Parameters

- **dimensions** -- The new dimensions to set.
- **clamp_to_container** -- Whether we should clamp the dimensions to the dimensions of the container or not.

set_position(*position: Vector2 | Tuple[float, float]*) → None

Sets the absolute screen position of this slider, updating all subordinate button elements at the same time.

Parameters
position -- The absolute screen position to set.

Returns
None

set_relative_position(*position: Vector2 | Tuple[float, float]*) → None

Sets the relative screen position of this slider, updating all subordinate button elements at the same time.

Parameters
position -- The relative screen position to set.

Returns
None

show()

In addition to the base UIElement.show() - show the sliding button and show the button_container which will propagate and show the left and right buttons.

update(*time_delta: float*)

Takes care of actually moving the slider based on interactions reported by the buttons or based on movement of the mouse if we are gripping the slider itself.

Parameters

time_delta -- the time in seconds between calls to update.

class pygame_gui.elements.**UIAutoResizingContainer**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *min_edges_rect: Rect | None = None*, *max_edges_rect: Rect | None = None*, *resize_left: bool = True*, *resize_right: bool = True*, *resize_top: bool = True*, *resize_bottom: bool = True*, *manager: IUIManagerInterface | None = None*, ***, *starting_height: int = 1*, *container: IContainerLikeInterface | None = None*, *parent_element: UIElement | None = None*, *object_id: ObjectID | str | None = None*, *anchors: Dict[str, str | UIElement] | None = None*, *visible: int = 1*)

Bases: *UIContainer*

A container like UI element that updates its size as elements within it change size, or new elements are added

Parameters

- **relative_rect** -- The starting size and relative position of the container.
- **min_edges_rect** -- The Rect which defines the maximum values for the left and top, and the minimum values for the right and bottom edges of the container. Defaults to the None (current rect)
- **max_edges_rect** -- The Rect which defines the minimum values for the left and top, and the maximum values for the right and bottom edges of the container. Defaults to the None (unbounded)
- **resize_left** -- Should the left side be resized?
- **resize_right** -- Should the right side be resized?
- **resize_top** -- Should the top side be resized?
- **resize_bottom** -- Should the bottom side be resized?
- **manager** -- The UI manager for this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **starting_height** -- The starting layer height of this container above its container. Defaults to 1.
- **container** -- The container this container is within. Defaults to None (which is the root container for the UI)
- **parent_element** -- A parent element for this container. Defaults to None, or the container if you've set that.
- **object_id** -- An object ID for this element.
- **anchors** -- Layout anchors in a dictionary.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.

add_element(*element:* IUIElementInterface) → None

Add a UIElement to the container. The UIElement's relative_rect parameter will be relative to this container. Overridden to also update dimensions.

Parameters
element -- A UIElement to add to this container.

Returns
None

on_contained_elements_changed(*target:* UIElement) → None

Update the positioning of the contained elements of this container. To be called when one of the contained elements may have moved, been resized or changed its anchors.

Parameters
target -- the UI element that has been benn moved resized or changed its anchors.

Returns
None

recalculate_abs_edges_rect() → None

Used to recalculate the absolute rects from the min and max edges rect which control the minimum and maximum sizes of the container. Usually called when the container of this container has moved, or the minimum or maximum rects have changed.

Returns
None

remove_element(*element:* IUIElementInterface) → None

Remove a UIElement from this container.

Parameters
element -- A UIElement to remove from this container.

Returns
None

update(*time_delta: float*) → None

Updates the container's size based upon the elements inside.

Call this function if you have added or removed an element from this container and want to update the size in the same frame, otherwise it will update in the next frame.

Parameters
time_delta -- The time passed between frames, measured in seconds.

Returns
None

update_containing_rect_position() → None

Overridden to also recalculate the absolute rects which control the minimum and maximum sizes of the container

Returns
None

update_max_edges_rect(*new_rect: Rect*) → None

Updates the container's minimum values for the left and top, and the maximum value for the right and bottom edges based upon the edges of the new rect.

Call the update function if you want to update the elements contained in the same frame, otherwise the elements contained within will update in the next frame.

Parameters
new_rect -- Rect to update the max_edges_rect with

Returns
None

update_min_edges_rect(*new_rect: Rect*) → None

Updates the container's maximum values for the left and top, and the minimum value for the right and bottom edges based upon the edges of the new rect.

Call the update function if you want to update the elements contained in the same frame, otherwise the elements contained within will update in the next frame.

Parameters
new_rect -- Rect to update the min_edges_rect with

Returns
None

class pygame_gui.elements.**UIButton**(*relative_rect: Rect | FRect | Tuple[float, float, float, float] | Vector2 | Tuple[float, float]*, *text: str*, *manager: IUIManagerInterface | None = None*, *container: IContainerLikeInterface | None = None*, *tool_tip_text: str | None = None*, *starting_height: int = 1*, *parent_element: UIElement | None = None*, *object_id: ObjectID | str | None = None*, *anchors: Dict[str, str | UIElement] | None = None*, *allow_double_clicks: bool = False*, *generate_click_events_from: Iterable[int] = frozenset({1})*, *visible: int = 1*, ***, *command: Callable | Dict[int, Callable] | None = None*, *tool_tip_object_id: ObjectID | None = None*, *text_kwargs: Dict[str, str] | None = None*, *tool_tip_text_kwargs: Dict[str, str] | None = None*, *max_dynamic_width: int | None = None*)

Bases: *UIElement*

A push button, a lot of the appearance of the button, including images to be displayed, is set up via the theme file. This button is designed to be pressed, do something, and then reset - rather than to be toggled on or off.

The button element is reused throughout the UI as part of other elements as it happens to be a very flexible interactive element.

Parameters

- **relative_rect** -- Normally a rectangle describing the position (relative to its container) and dimensions. Also accepts a position Coordinate where the dimensions will be dynamic depending on the text contents. Dynamic dimensions can be requested by setting the required dimension to -1.
- **text** -- Text for the button.
- **manager** -- The UIManager that manages this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **container** -- The container that this element is within. If not provided or set to None will be the root window's container.
- **tool_tip_text** -- Optional tool tip text, can be formatted with HTML. If supplied will appear on hover.
- **starting_height** -- The height in layers above its container that this element will be placed.
- **parent_element** -- The element this element 'belongs to' in the theming hierarchy.
- **object_id** -- A custom defined ID for fine-tuning of theming.

- **anchors** -- A dictionary describing what this element's relative_rect is relative to.
- **allow_double_clicks** -- Enables double-clicking on buttons which will generate a unique event.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.
- **command** -- Functions to be called when an event is triggered by this element.
- **text_kwargs** -- a dictionary of variable arguments to pass to the translated string useful when you have multiple translations that need variables inserted in the middle.

bind(*event: int*, *function: Callable | None = None*)

Bind a function to an element event.

Parameters

- **event** -- The event to bind.
- **function** -- The function to bind. None to unbind.

can_hover() → bool

Tests whether we can trigger the hover state for this button, other states take priority over it.

Returns
True if we are able to hover this button.

check_pressed() → bool

A direct way to check if this button has been pressed in the last update cycle.

Returns
True if the button has been pressed.

disable()

Disables the button so that it is no longer interactive.

enable()

Re-enables the button, so we can once again interact with it.

hide()

In addition to the base UIElement.hide() - Change the hovered state to a normal state.

hover_point(*hover_x: int*, *hover_y: int*) → bool

Tests if a position should be considered 'hovering' the button. Normally this just means our mouse pointer is inside the buttons rectangle, however if we are holding onto the button for a purpose(e.g. dragging a window around by its menu bar) the hover radius can be made to grow, so we don't keep losing touch with whatever we are moving.

Parameters

- **hover_x** -- horizontal pixel coordinate to test.
- **hover_y** -- vertical pixel coordinate to test

Returns
Returns True if we are hovering.

in_hold_range(*position: Vector2 | Tuple[int, int] | Tuple[float, float]*) → bool

Imagines a potentially larger rectangle around our button in which range we still grip hold of our button when moving the mouse. Makes it easier to use scrollbars.

Parameters
position -- The position we are testing.

Return bool
Returns True if our position is inside the hold range.

kill()

Overrides the standard sprite kill method to also kill any tooltips belonging to this button.

on_hovered()

Called when we enter the hover state, it sets the colours and image of the button to the appropriate values and redraws it.

on_locale_changed()

Called for each element when the locale is changed on their UIManager

on_self_event(*event: int*, *data: Dict[str, Any] | None = None*)

Called when an event is triggered by this element. Handles these events either by posting the event back to the event queue, or by running a function supplied by the user.

Parameters

- **event** -- The event triggered.
- **data** -- event data

on_unhovered()

Called when we leave the hover state. Resets the colours and images to normal and kills any tooltip that was created while we were hovering the button.

process_event(*event: Event*) → bool

Handles various interactions with the button.

Parameters
event -- The event to process.

Returns
Return True if we want to consume this event, so it is not passed on to the rest of the UI.

rebuild()

A complete rebuild of the drawable shape used by this button.

rebuild_from_changed_theme_data()

Checks if any theming parameters have changed, and if so triggers a full rebuild of the button's drawable shape

select()

Called when we select focus this element. Changes the colours and image to the appropriate ones for the new state then redraws the button.

set_hold_range(*xy_range: Tuple[int, int]*)

Set x and y values, in pixels, around our button to use as the hold range for time when we want to drag a button about but don't want it to slip out of our grasp too easily.

Imagine it as a large rectangle around our button, larger in all directions by whatever values we specify here.

Parameters
xy_range -- The x and y values used to create our larger 'holding' rectangle.

set_text(*text: str*, *, *text_kwargs: Dict[str, str] | None = None*)

Sets the text on the button. The button will rebuild.

Parameters

- **text** -- The new text to set.
- **text_kwargs** -- a dictionary of variable arguments to pass to the translated string useful when you have multiple translations that need variables inserted in the middle.

unselect()

Called when we are no longer select focusing this element. Restores the colours and image to the default state then redraws the button.

update(*time_delta: float*)

Sets the pressed state for an update cycle if we've pressed this button recently.

Parameters

time_delta -- the time in seconds between one call to update and the next.

class pygame_gui.elements.**UIDropDownMenu**(*options_list: List[str] | List[Tuple[str, str]]*, *starting_option: str | Tuple[str, str]*, *relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *manager:* IUIManagerInterface | *None = None*, *container:* IContainerLikeInterface | *None = None*, *parent_element:* UIElement | *None = None*, *object_id:* ObjectID | *str* | *None = None*, *expansion_height_limit: int* | *None = None*, *anchors: Dict[str, str* | UIElement*]* | *None = None*, *visible: int = 1*, *, *expand_on_option_click: bool = True*)

Bases: *UIContainer*

A drop-down menu lets us choose one text option from a list. That list of options can be expanded and hidden at the press of a button. While the element is called a drop-down, it can also be made to 'climb up' by changing the 'expand_direction' styling option to 'up' in the theme file.

The drop-down is implemented through two states, one representing the 'closed' menu state and one for when it has been 'expanded'.

Parameters

- **options_list** -- The list of options to choose from. They must be strings.
- **starting_option** -- The starting option, selected when the menu is first created.
- **relative_rect** -- The size and position of the element when not expanded.
- **manager** -- The UIManager that manages this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **container** -- The container that this element is within. If set to None will be the root window's container.
- **parent_element** -- The element this element 'belongs to' in the theming hierarchy.
- **object_id** -- A custom defined ID for fine-tuning of theming.
- **expansion_height_limit** -- Limit on the height that this will expand to, defaults to the container bounds.
- **anchors** -- A dictionary describing what this element's relative_rect is relative to.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.

- **expand_on_option_click** -- If this is set to False the drop-down will only expand when the open/close button is pressed and not when the selected option is pressed.

add_options(*new_options: List[str] | List[Tuple[str, str]]*) → None

Add new options to the drop-down. Will close the drop-down if it is currently open.

In many cases it may be easier just to recreate the drop-down with whatever the new options list is.

Parameters

new_options -- The list of new options to add.

disable()

Disables the button so that it is no longer interactive.

enable()

Re-enables the button so, we can once again interact with it.

hide()

In addition to the base UIElement.hide() - if the current state is 'expanded' call its hide() method, which begins a transition of the UIDropDownMenu to the 'closed' state, and call the hide() method of the 'closed' state which hides all it's children widgets.

kill()

Overrides the standard sprite kill to also properly kill/finish the current state of the drop-down. Depending on whether it is expanded or closed the drop-down menu will have different elements to clean up.

on_fresh_drawable_shape_ready()

Called by an element's drawable shape when it has a new image surface ready for use, normally after a rebuilding/redrawing of some kind.

process_event(*event: Event*) → bool

Handles various interactions with the drop-down menu by passing them along to the active state.

Parameters

event -- The event to process.

Returns

Return True if we want to consume this event, so it is not passed on to the rest of the UI.

rebuild()

A complete rebuild of the drawable parts of this element.

rebuild_from_changed_theme_data()

Triggers the element to rebuild if any of its theming data has changed, which involves a lot of checking and validating its theming data.

remove_options(*options_to_remove: List[str] | List[Tuple[str, str]]*) → None

Will remove all instances of the options provided.

Parameters

options_to_remove -- The list of new options to remove.

set_dimensions(*dimensions: Vector2 | Tuple[float, float]*, *clamp_to_container: bool = False*)

Sets the dimensions of this drop down, updating all subordinate button elements at the same time.

Parameters

- **dimensions** -- The new dimensions to set.
- **clamp_to_container** -- Whether we should clamp the dimensions to the dimensions of the container or not.

set_position(*position: Vector2 | Tuple[float, float]*)

Sets the absolute screen position of this drop down, updating all subordinate button elements at the same time.

Parameters
position -- The absolute screen position to set.

set_relative_position(*position: Vector2 | Tuple[float, float]*)

Sets the relative screen position of this drop down, updating all subordinate button elements at the same time.

Parameters
position -- The relative screen position to set.

show()

In addition to the base UIElement.show() - call show() on the closed state - showing its buttons.

unfocus()

A stub to override. Called when we stop focusing this UI element.

update(*time_delta: float*)

The update here deals with transitioning between the two states of the drop-down menu and then passes the rest of the work onto whichever state is active.

Parameters
time_delta -- The time in second between calls to update.

class pygame_gui.elements.**UIHorizontalScrollBar**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *visible_percentage: float*, *manager: IUIManagerInterface | None = None*, *container: IContainerLikeInterface | None = None*, *parent_element: UIElement | None = None*, *object_id: ObjectID | str | None = None*, *anchors: Dict[str, str | UIElement] | None = None*, *visible: int = 1*)

Bases: *UIElement*

A horizontal scroll bar allows users to position a smaller visible area within a horizontally larger area.

Parameters

- **relative_rect** -- The size and position of the scroll bar.
- **visible_percentage** -- The horizontal percentage of the larger area that is visible, between 0.0 and 1.0.
- **manager** -- The UIManager that manages this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **container** -- The container that this element is within. If set to None will be the root window's container.
- **parent_element** -- The element this element 'belongs to' in the theming hierarchy.
- **object_id** -- A custom defined ID for fine-tuning of theming.
- **anchors** -- A dictionary describing what this element's relative_rect is relative to.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.

check_has_moved_recently() → bool

Returns True if the scroll bar was moved in the last call to the update function.

Returns
True if we've recently moved the scroll bar, False otherwise.

disable()

Disables the scroll bar, so it is no longer interactive.

enable()

Enables the scroll bar, so it is interactive once again.

hide()

In addition to the base UIElement.hide() - hide the self.button_container which will propagate and hide all the buttons.

kill()

Overrides the kill() method of the UI element class to kill all the buttons in the scroll bar and clear any of the parts of the scroll bar that are currently recorded as the 'last focused horizontal scroll bar element' on the ui manager.

NOTE: the 'last focused' state on the UI manager is used so that the mouse wheel will move whichever scrollbar we last fiddled with even if we've been doing other stuff. This seems to be consistent with the most common mousewheel/scrollbar interactions used elsewhere.

process_event(*event: Event*) → bool

Checks an event from pygame's event queue to see if the scroll bar needs to react to it. In this case it is just mousewheel events, mainly because the buttons that make up the scroll bar will handle the required mouse click events.

Parameters
event -- The event to process.

Returns
Returns True if we've done something with the input event.

rebuild()

Rebuild anything that might need rebuilding.

rebuild_from_changed_theme_data()

Called by the UIManager to check the theming data and rebuild whatever needs rebuilding for this element when the theme data has changed.

redraw_scrollbar()

Redraws the 'scrollbar' portion of the whole UI element. Called when we change the visible percentage.

reset_scroll_position()

Reset the current scroll position back to the top.

set_dimensions(*dimensions: Vector2 | Tuple[float, float]*, *clamp_to_container: bool = False*)

Method to directly set the dimensions of an element.

Parameters

- **dimensions** -- The new dimensions to set.
- **clamp_to_container** -- Whether we should clamp the dimensions to the dimensions of the container or not.

set_position(*position: Vector2 | Tuple[float, float]*)

Sets the absolute screen position of this scroll bar, updating all subordinate button elements at the same time.

Parameters
position -- The absolute screen position to set.

set_relative_position(*position: Vector2 | Tuple[float, float]*)

Sets the relative screen position of this scroll bar, updating all subordinate button elements at the same time.

Parameters
position -- The relative screen position to set.

set_scroll_from_start_percentage(*new_start_percentage: float*)

Set the scroll bar's scrolling position from a percentage between 0.0 and 1.0.

Parameters
new_start_percentage -- the percentage to set.

set_visible_percentage(*percentage: float*)

Sets the percentage of the total 'scrollable area' that is currently visible. This will affect the size of the scrollbar and should be called if the horizontal size of the 'scrollable area' or the horizontal size of the visible area change.

Parameters
percentage -- A float between 0.0 and 1.0 representing the percentage that is visible.

show()

In addition to the base UIElement.show() - show the self.button_container which will propagate and show all the buttons.

property start_percentage

turning start_percentage into a property, so we can round it to mitigate floating point errors

update(*time_delta: float*)

Called once per update loop of our UI manager. Deals largely with moving the scroll bar and updating the resulting 'start_percentage' variable that is then used by other 'scrollable' UI elements to control the point they start drawing.

Reacts to presses of the up and down arrow buttons, movement of the mouse wheel and dragging of the scroll bar itself.

Parameters
time_delta -- A float, roughly representing the time in seconds between calls to this method.

class pygame_gui.elements.**UIHorizontalSlider**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *start_value: float | int*, *value_range: Tuple[float | int, float | int]*, *manager: IUIManagerInterface | None = None*, *container: IContainerLikeInterface | None = None*, *parent_element: UIElement | None = None*, *object_id: ObjectID | str | None = None*, *anchors: Dict[str, str | UIElement] | None = None*, *visible: int = 1*, *click_increment: float | int = 1*)

Bases: *UIElement*

A horizontal slider is intended to help users adjust values within a range, for example a volume control.

Parameters

- **relative_rect** -- A rectangle describing the position and dimensions of the element.

- **start_value** -- The value to start the slider at.
- **value_range** -- The full range of values.
- **manager** -- The UIManager that manages this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **container** -- The container that this element is within. If set to None will be the root window's container.
- **parent_element** -- The element this element 'belongs to' in the theming hierarchy.
- **object_id** -- A custom defined ID for fine-tuning of theming.
- **anchors** -- A dictionary describing what this element's relative_rect is relative to.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.
- **click_increment** -- the amount to increment by when clicking one of the arrow buttons.

disable()

Disable the slider. It should not be interactive and will use the disabled theme colours.

enable()

Enable the slider. It should become interactive and will use the normal theme colours.

get_current_value() → float | int

Gets the current value the slider is set to.

Returns
The current value recorded by the slider.

hide()

In addition to the base UIElement.hide() - hide the sliding button and hide the button_container which will propagate and hide the left and right buttons.

kill()

Overrides the normal sprite kill() method to also kill the button elements that help make up the slider.

process_event(*event: Event*) → bool

A stub to override. Gives UI Elements access to pygame events.

Parameters
event -- The event to process.

Returns
Should return True if this element makes use of this event.

rebuild()

Rebuild anything that might need rebuilding.

rebuild_from_changed_theme_data()

Called by the UIManager to check the theming data and rebuild whatever needs rebuilding for this element when the theme data has changed.

set_current_value(*value: float | int*, *warn: bool = True*)

Sets the value of the slider, which will move the position of the slider to match. Will issue a warning if the value set is not in the value range.

Parameters

- **value** -- The value to set.

- **warn** -- set to 'False' to suppress the default warning, instead the value will be clamped.

set_dimensions(*dimensions: Vector2 | Tuple[float, float]*, *clamp_to_container: bool = False*)

Method to directly set the dimensions of an element.

Parameters

- **dimensions** -- The new dimensions to set.
- **clamp_to_container** -- Whether we should clamp the dimensions to the dimensions of the container or not.

set_position(*position: Vector2 | Tuple[float, float]*)

Sets the absolute screen position of this slider, updating all subordinate button elements at the same time.

Parameters

position -- The absolute screen position to set.

set_relative_position(*position: Vector2 | Tuple[float, float]*)

Sets the relative screen position of this slider, updating all subordinate button elements at the same time.

Parameters

position -- The relative screen position to set.

show()

In addition to the base UIElement.show() - show the sliding button and show the button_container which will propagate and show the left and right buttons.

update(*time_delta: float*)

Takes care of actually moving the slider based on interactions reported by the buttons or based on movement of the mouse if we are gripping the slider itself.

Parameters

time_delta -- the time in seconds between calls to update.

class pygame_gui.elements.**UIImage**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *image_surface: Surface*, *manager: IUIManagerInterface | None = None*, *image_is_alpha_premultiplied: bool = False*, *container: IContainerLikeInterface | None = None*, *parent_element: UIElement | None = None*, *object_id: ObjectID | str | None = None*, *anchors: Dict[str, str | UIElement] | None = None*, *visible: int = 1*, ***, *starting_height: int = 1*)

Bases: *UIElement*

Displays a pygame surface as a UI element, intended for an image, but it can serve other purposes.

Parameters

- **relative_rect** -- The rectangle that contains, positions and scales the image relative to its container.
- **image_surface** -- A pygame surface to display.
- **manager** -- The UIManager that manages this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **container** -- The container that this element is within. If not provided or set to None will be the root window's container.
- **parent_element** -- The element this element 'belongs to' in the theming hierarchy.
- **object_id** -- A custom defined ID for fine-tuning of theming.

- **anchors** -- A dictionary describing what this element's relative_rect is relative to.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.

rebuild_from_changed_theme_data()

A stub to override when we want to rebuild from theme data.

set_dimensions(*dimensions: Vector2 | Tuple[float, float]*, *clamp_to_container: bool = False*)

Set the dimensions of this image, scaling the image surface to match.

Parameters

- **dimensions** -- The new dimensions of the image.
- **clamp_to_container** -- Whether we should clamp the dimensions to the dimensions of the container or not.

set_image(*new_image: Surface | None*, *image_is_alpha_premultiplied: bool = False*) → None

Allows users to change the image displayed on a UIImage element during run time, without recreating the element.

GUI images are converted to the correct format for the GUI if the supplied image is not the dimensions of the UIImage element it will be scaled to fit. In this situation, an original size image is retained as well in case of future resizing events.

Parameters

- **new_image** -- the new image surface to use in the UIImage element.
- **image_is_alpha_premultiplied** -- set to True if the image is already in alpha multiplied colour format.

class pygame_gui.elements.**UILabel**(*relative_rect: Rect | FRect | Tuple[float, float, float, float] | Vector2 | Tuple[float, float]*, *text: str*, *manager: IUIManagerInterface | None = None*, *container: IContainerLikeInterface | None = None*, *parent_element: UIElement | None = None*, *object_id: ObjectID | str | None = None*, *anchors: Dict[str, str | UIElement] | None = None*, *visible: int = 1*, ***, *text_kwargs: Dict[str, str] | None = None*)

Bases: *UIElement*, *IUITextOwnerInterface*

A label lets us display a single line of text with a single font style. It's a quick to rebuild and simple alternative to the text box element.

Parameters

- **relative_rect** -- Normally a rectangle describing the position (relative to its container) and dimensions. Also accepts a position Coordinate where the dimensions will be dynamic depending on the text contents. Dynamic dimensions can be requested by setting the required dimension to -1.
- **text** -- The text to display in the label.
- **manager** -- The UIManager that manages this label. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **container** -- The container that this element is within. If not provided or set to None will be the root window's container.
- **parent_element** -- The element this element 'belongs to' in the theming hierarchy.
- **object_id** -- A custom defined ID for fine-tuning of theming.

- **anchors** -- A dictionary describing what this element's relative_rect is relative to.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.
- **text_kwargs** -- a dictionary of variable arguments to pass to the translated string useful when you have multiple translations that need variables inserted in the middle.

clear_all_active_effects(*sub_chunk:* TextLineChunkFTFont | *None* = *None*)

Clears any active effects and redraws the text. A full reset, usually called before firing off a new effect if one is already in progress.

Parameters
sub_chunk -- An optional chunk so we only clear the effect from this chunk.

clear_text_surface(*sub_chunk:* TextLineChunkFTFont | *None* = *None*)

Clear the text surface

Parameters
sub_chunk -- An optional chunk so we only clear the surface for this chunk.

disable()

Disables the label so that its text changes to the disabled colour.

enable()

Re-enables the label so that its text changes to the normal colour

get_object_id() → str

The UI object ID of this text owner for use in effect events.

Returns
the ID string

get_text_letter_count(*sub_chunk:* TextLineChunkFTFont | *None* = *None*) → int

The amount of letters in the text

Parameters
sub_chunk -- An optional chunk to restrict the count to only this chunk.

Returns
number of letters as an int

on_locale_changed()

Called for each element when the locale is changed on their UIManager

rebuild()

Re-render the text to the label's underlying sprite image. This allows us to change what the displayed text is or remake it with different theming (if the theming has changed).

rebuild_from_changed_theme_data()

Checks if any theming parameters have changed, and if so triggers a full rebuild of the element.

set_active_effect(*effect_type:* UITextEffectType | *None*, *params:* *Dict[str, Any]* | *None* = *None*, *effect_tag:* *str* | *None* = *None*)

Set an animation effect to run on the text box. The effect will start running immediately after this call.

These effects are currently supported:

- TEXT_EFFECT_TYPING_APPEAR - Will look as if the text is being typed in.
- TEXT_EFFECT_FADE_IN - The text will fade in from the background colour.

- TEXT_EFFECT_FADE_OUT - The text will fade out to the background colour.

Parameters

- **effect_tag** -- if not None, only apply the effect to chunks with this tag.
- **params** -- Any parameters for the effect you are setting, if none are set defaults will be used.
- **effect_type** -- The type of the effect to set. If set to None instead it will cancel any active effect.

set_text(*text: str*, *, *text_kwargs: Dict[str, str] | None = None*)

Changes the string displayed by the label element. Labels do not support HTML styling.

Parameters

- **text** -- the text to set the label to.
- **text_kwargs** -- a dictionary of variable arguments to pass to the translated string useful when you have multiple translations that need variables inserted in the middle.

set_text_alpha(*alpha: int*, *sub_chunk:* TextLineChunkFTFont | *None = None*)

Set the global alpha value for the text

Parameters

- **alpha** -- the alpha to set.
- **sub_chunk** -- An optional chunk so we only set the alpha for this chunk.

set_text_offset_pos(*offset: Tuple[int, int]*, *sub_chunk:* TextLineChunkFTFont | *None = None*)

Move the text around by this offset.

Parameters

- **offset** -- the offset to set
- **sub_chunk** -- An optional chunk so we only set the offset for this chunk.

Returns

set_text_rotation(*rotation: int*, *sub_chunk:* TextLineChunkFTFont | *None = None*)

rotate the text by this int in degrees

Parameters

- **rotation** -- the rotation to set
- **sub_chunk** -- An optional chunk so we only set the rotation for this chunk.

Returns

set_text_scale(*scale: int*, *sub_chunk:* TextLineChunkFTFont | *None = None*)

Scale the text by this float

Parameters

- **scale** -- the scale to set
- **sub_chunk** -- An optional chunk so we only set the rotation for this chunk.

Returns

stop_finished_effect(*sub_chunk:* TextLineChunkFTFont | *None* = *None*)

Stops a finished effect. Will leave effected text in the state it was in when effect ended. Used when an effect reaches a natural end where we might want to keep it in the end of effect state (e.g. a fade out)

Parameters
sub_chunk -- An optional chunk so we only clear the effect from this chunk.

update(*time_delta: float*)

Called once every update loop of the UI Manager.

Parameters
time_delta -- The time in seconds between calls to update. Useful for timing things.

update_text_effect(*time_delta: float*)

Update any active text effect on the text owner

Parameters
time_delta -- the time delta in seconds

update_text_end_position(*end_pos: int*, *sub_chunk:* TextLineChunkFTFont | *None* = *None*)

The position in the text to render up to.

Parameters

- **end_pos** -- The current end position as an int
- **sub_chunk** -- An optional chunk to restrict the end_position to only this chunk.

class pygame_gui.elements.**UIPanel**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *starting_height: int = 1*, *manager:* IUIManagerInterface | *None* = *None*, **, element_id: str = 'panel'*, *margins: Dict[str, int]* | *None* = *None*, *container:* IContainerLikeInterface | *None* = *None*, *parent_element:* UIElement | *None* = *None*, *object_id:* ObjectID | *str* | *None* = *None*, *anchors: Dict[str, str* | UIElement*]* | *None* = *None*, *visible: int = 1*)

Bases: *UIElement*, *IContainerLikeInterface*

A rectangular panel that holds a UI container and is designed to overlap other elements. It acts a little like a window that is not shuffled about in a stack - instead remaining at the same layer distance from the container it was initially placed in.

It's primary purpose is for things like involved HUDs in games that want to always sit on top of UI elements that may be present 'inside' the game world (e.g. player health bars). By creating a UI Panel at a height above the highest layer used by the game world's UI elements we can ensure that all elements added to the panel are always above the fray.

Parameters

- **relative_rect** -- The positioning and sizing rectangle for the panel. See the layout guide for details.
- **starting_height** -- How many layers above its container to place this panel on.
- **manager** -- The GUI manager that handles drawing and updating the UI and interactions between elements. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **margins** -- Controls the distance between the edge of the panel and where it's container should begin.
- **container** -- The container this panel is inside - distinct from this panel's own container.

- **parent_element** -- A hierarchical 'parent' used for signifying belonging and used in theming and events.
- **object_id** -- An identifier that can be used to help distinguish this particular panel from others.
- **anchors** -- Used to layout elements and dictate what the relative_rect is relative to. Defaults to the top left.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.

are_contents_hovered() → bool

Are any of the elements in the container hovered? Used for handling mousewheel events.

Returns
True if one of the elements is hovered, False otherwise.

disable()

Disables all elements in the panel, so they are no longer interactive.

enable()

Enables all elements in the panel, so they are interactive again.

get_container() → *IUIContainerInterface*

Returns the container that should contain all the UI elements in this panel.

Return UIContainer
The panel's container.

hide()

In addition to the base UIElement.hide() - call hide() of owned container - panel_container.

kill()

Overrides the basic kill() method of a pygame sprite so that we also kill all the UI elements in this panel.

process_event(*event: Event*) → bool

Can be overridden, also handle resizing windows. Gives UI Windows access to pygame events. Currently just blocks mouse click down events from passing through the panel.

Parameters
event -- The event to process.

Returns
Should return True if this element consumes this event.

rebuild()

A complete rebuild of the drawable shape used by this button.

rebuild_from_changed_theme_data()

Checks if any theming parameters have changed, and if so triggers a full rebuild of the button's drawable shape.

set_dimensions(*dimensions: Vector2 | Tuple[float, float]*, *clamp_to_container: bool = False*)

Set the size of this panel and then re-sizes and shifts the contents of the panel container to fit the new size.

Parameters

- **dimensions** -- The new dimensions to set.
- **clamp_to_container** -- Whether we should clamp the dimensions to the dimensions of the container or not.

set_position(*position: Vector2 | Tuple[float, float]*)

Method to directly set the absolute screen rect position of an element.

Parameters
position -- The new position to set.

set_relative_position(*position: Vector2 | Tuple[float, float]*)

Method to directly set the relative rect position of an element.

Parameters
position -- The new position to set.

show()

In addition to the base UIElement.show() - call show() of owned container - panel_container.

update(*time_delta: float*)

A method called every update cycle of our application. Designed to be overridden by derived classes but also has a little functionality to make sure the panel's layer 'thickness' is accurate and to handle window resizing.

Parameters
time_delta -- time passed in seconds between one call to this method and the next.

class pygame_gui.elements.**UIProgressBar**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *manager: IUIManagerInterface | None = None*, *container: IContainerLikeInterface | None = None*, *parent_element: UIElement | None = None*, *object_id: ObjectID | str | None = None*, *anchors: Dict[str, str | UIElement] | None = None*, *visible: int = 1*)

Bases: *UIStatusBar*

A UI that will display a progress bar from 0 to 100%

Parameters

- **relative_rect** -- The rectangle that defines the size and position of the progress bar.
- **manager** -- The UIManager that manages this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **container** -- The container that this element is within. If not provided or set to None will be the root window's container.
- **parent_element** -- The element this element 'belongs to' in the theming hierarchy.
- **object_id** -- A custom defined ID for fine-tuning of theming.
- **anchors** -- A dictionary describing what this element's relative_rect is relative to.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.

status_text()

Subclass and override this method to change what text is displayed, or to suppress the text.

class pygame_gui.elements.**UIScreenSpaceHealthBar**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *manager: IUIManagerInterface | None = None*, *sprite_to_monitor: SpriteWithHealth | None = None*, *container: IContainerLikeInterface | None = None*, *parent_element: UIElement | None = None*, *object_id: ObjectID | str | None = None*, *anchors: Dict[str, str | UIElement] | None = None*, *visible: int = 1*)

Bases: *UIStatusBar*

A UI that will display health capacity and current health for a sprite in 'screen space'. That means it won't move with the camera. This is a good choice for a user/player sprite.

Parameters

- **relative_rect** -- The rectangle that defines the size and position of the health bar.
- **manager** -- The UIManager that manages this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **sprite_to_monitor** -- The sprite we are displaying the health of.
- **container** -- The container that this element is within. If set to None will be the root window's container.
- **parent_element** -- The element this element 'belongs to' in the theming hierarchy.
- **object_id** -- A custom defined ID for fine-tuning of theming.
- **anchors** -- A dictionary describing what this element's relative_rect is relative to.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.

status_text()

Subclass and override this method to change what text is displayed, or to suppress the text.

class pygame_gui.elements.**UIScrollingContainer**(*relative_rect: Rect*, *manager: IUIManagerInterface | None = None*, ***, *starting_height: int = 1*, *container: IContainerLikeInterface | None = None*, *parent_element: UIElement | None = None*, *object_id: ObjectID | str | None = None*, *element_id: List[str] | None = None*, *anchors: Dict[str, str | UIElement] | None = None*, *visible: int = 1*, *should_grow_automatically: bool = True*, *allow_scroll_x: bool = True*, *allow_scroll_y: bool = True*)

Bases: *UIElement*, *IContainerLikeInterface*

A container like UI element that lets users scroll around a larger container of content with scroll bars.

Parameters

- **relative_rect** -- The size and relative position of the container. This will also be the starting size of the scrolling area.
- **manager** -- The UI manager for this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **starting_height** -- The starting layer height of this container above its container. Defaults to 1.

- **container** -- The container this container is within. Defaults to None (which is the root container for the UI)
- **parent_element** -- A parent element for this container. Defaults to None, or the container if you've set that.
- **object_id** -- An object ID for this element.
- **anchors** -- Layout anchors in a dictionary.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.
- **allow_scroll_x** -- Whether a scrollbar should be added to scroll horizontally (when needed). Defaults to True.
- **allow_scroll_y** -- Whether a scrollbar should be added to scroll vertically (when needed). Defaults to True.

are_contents_hovered() → bool

Are any of the elements in the container hovered? Used for handling mousewheel events.

Returns
True if one of the elements is hovered, False otherwise.

disable()

Disables all elements in the container, so they are no longer interactive.

enable()

Enables all elements in the container, so they are interactive again.

get_container() → *IUIContainerInterface*

Gets the scrollable container area (the one that moves around with the scrollbars) from this container-like UI element.

Returns
the scrolling container.

hide()

In addition to the base UIElement.hide() - call hide() of owned container - _root_container. All other sub-elements (view_container, scrollbars) are children of _root_container, so it's visibility will propagate to them - there is no need to call their hide() methods separately.

kill()

Overrides the basic kill() method of a pygame sprite so that we also kill all the UI elements in this panel.

set_dimensions(*dimensions: Vector2 | Tuple[float, float]*, *clamp_to_container: bool = False*)

Method to directly set the dimensions of an element.

NOTE: Using this on elements inside containers with non-default anchoring arrangements may make a mess of them.

Parameters

- **dimensions** -- The new dimensions to set.
- **clamp_to_container** -- Whether we should clamp the dimensions to the dimensions of the container or not.

set_position(*position: Vector2 | Tuple[float, float]*)

Method to directly set the absolute screen rect position of an element.

Parameters
position -- The new position to set.

set_relative_position(*position: Vector2 | Tuple[float, float]*)

Method to directly set the relative rect position of an element.

Parameters
position -- The new position to set.

set_scrollable_area_dimensions(*dimensions: Vector2 | Tuple[float, float]*)

Set the size of the scrollable area container. It starts the same size as the view container, but often you want to expand it, or why have a scrollable container?

Parameters
dimensions -- The new dimensions.

show()

In addition to the base UIElement.show() - call show() of owned container - _root_container. All other sub-elements (view_container, scrollbars) are children of _root_container, so it's visibility will propagate to them - there is no need to call their show() methods separately.

update(*time_delta: float*)

Updates the scrolling container's position based upon the scroll bars and updates the scrollbar's visible percentage as well if that has changed.

Parameters
time_delta -- The time passed between frames, measured in seconds.

class pygame_gui.elements.**UISelectionList**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *item_list: List[str] | List[Tuple[str, str]]*, *manager: IUIManagerInterface | None = None*, ***, *allow_multi_select: bool = False*, *allow_double_clicks: bool = True*, *container: IContainerLikeInterface | None = None*, *starting_height: int = 1*, *parent_element: UIElement | None = None*, *object_id: ObjectID | str | None = None*, *anchors: Dict[str, str | UIElement] | None = None*, *visible: int = 1*, *default_selection: str | Tuple[str, str] | List[str] | List[Tuple[str, str]] | None = None*)

Bases: *UIElement*

A rectangular element that holds any number of selectable text items displayed as a list.

Parameters

- **relative_rect** -- The positioning and sizing rectangle for the panel. See the layout guide for details.
- **item_list** -- A list of items as strings (item name only), or tuples of two strings (name, theme_object_id).
- **default_selection** -- Default item(s) that should be selected: a string or a (str, str) tuple for single-selection lists or a list of strings or list of (str, str) tuples for multi-selection lists.
- **manager** -- The GUI manager that handles drawing and updating the UI and interactions between elements. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **allow_multi_select** -- True if we are allowed to pick multiple things from the selection list.
- **allow_double_clicks** -- True if we can double-click on items in the selection list.

- **container** -- The container this element is inside (by default the root container) distinct from this panel's container.
- **starting_height** -- The starting height up from its container where this list is placed into a layer.
- **parent_element** -- A hierarchical 'parent' used for signifying belonging and used in theming and events.
- **object_id** -- An identifier that can be used to help distinguish this particular element from others with the same hierarchy.
- **anchors** -- Used to layout elements and dictate what the relative_rect is relative to. Defaults to the top left.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.

add_items(*new_items: List[str] | List[Tuple[str, str]]*) → None

Add any number of new items to the selection list. Uses the same format as when the list is first created.

Parameters
new_items -- the list of new items to add

disable()

Disables all elements in the selection list, so they are no longer interactive.

enable()

Enables all elements in the selection list, so they are interactive again.

get_multi_selection(*include_object_id: bool = False*) → List[str] | List[Tuple[str, str]]

Get all the selected items in our selection list. Only works if this is a multi-selection list.

Parameters
include_object_id -- if True adds the object id of this list item to the returned list of Tuples. If False we just get a list of the visible text strings only.

Returns
A list of the selected items in our selection list. May be empty if nothing selected.

get_single_selection(*include_object_id: bool = False*) → Tuple[str, str] | str | None

Get the selected item in a list, if any. Only works if this is a single-selection list.

Parameters
include_object_id -- if True adds the object id of this list item to the returned list of Tuples. If False we just get a list of the visible text strings only.

Returns
A single item name as a string or None.

get_single_selection_start_percentage()

The percentage through the height of the list where the top of the first selected option is.

hide()

In addition to the base UIElement.hide() - call hide() of owned container - list_and_scroll_bar_container. All other sub-elements (item_list_container, scrollbar) are children of list_and_scroll_bar_container, so it's visibility will propagate to them - there is no need to call their hide() methods separately.

kill()

Overrides the basic kill() method of a pygame sprite so that we also kill all the UI elements in this panel.

process_event(*event: Event*) → bool

Can be overridden, also handle resizing windows. Gives UI Windows access to pygame events. Currently just blocks mouse click down events from passing through the panel.

Parameters
event -- The event to process.

Returns
Should return True if this element makes use of this event.

rebuild()

A complete rebuild of the drawable shape used by this element.

rebuild_from_changed_theme_data()

Checks if any theming parameters have changed, and if so triggers a full rebuild of the button's drawable shape

remove_items(*items_to_remove: List[str] | List[Tuple[str, str]]*) → None

Will remove all instances of the items provided. The full tuple is required for items with a display name and an object ID.

Parameters
items_to_remove -- The list of new options to remove.

set_dimensions(*dimensions: Vector2 | Tuple[float, float]*, *clamp_to_container: bool = False*)

Set the size of this panel and then resizes and shifts the contents of the panel container to fit the new size.

Parameters

- **dimensions** -- The new dimensions to set.
- **clamp_to_container** -- clamp these dimensions to the size of the element's container.

set_item_list(*new_item_list: List[str] | List[Tuple[str, str]]*)

Set a new string list (or tuple of strings & ids list) as the item list for this selection list. This will change what is displayed in the list.

Tuples should be arranged like so:

(list_text, object_ID)

- list_text: displayed in the UI
- object_ID: used for theming and events

Parameters
new_item_list -- The new list to switch to. Can be a list of strings or tuples.

set_position(*position: Vector2 | Tuple[float, float]*)

Sets the absolute screen position of this slider, updating all subordinate button elements at the same time.

Parameters
position -- The absolute screen position to set.

set_relative_position(*position: Vector2 | Tuple[float, float]*)

Method to directly set the relative rect position of an element.

Parameters
position -- The new position to set.

show()

In addition to the base UIElement.show() - call show() of owned container - list_and_scroll_bar_container. All other sub-elements (item_list_container, scrollbar) are children of list_and_scroll_bar_container, so it's visibility will propagate to them - there is no need to call their show() methods separately.

update(*time_delta: float*)

A method called every update cycle of our application. Designed to be overridden by derived classes but also has a little functionality to make sure the panel's layer 'thickness' is accurate and to handle window resizing.

Parameters

time_delta -- time passed in seconds between one call to this method and the next.

class pygame_gui.elements.**UIStatusBar**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *manager: IUIManagerInterface | None = None*, *sprite: SpriteWithHealth | None = None*, *follow_sprite: bool = True*, *percent_method: Callable[[], float] | None = None*, *container: IContainerLikeInterface | None = None*, *parent_element: UIElement | None = None*, *object_id: ObjectID | str | None = None*, *anchors: Dict[str, str | UIElement] | None = None*, *visible: int = 1*)

Bases: *UIElement*

Displays a status/progress bar.

This is a flexible class that can be used to display status for a sprite (health/mana/fatigue, etc.), or to provide a status bar on the screen not attached to any particular object. You can use multiple status bars for a sprite to show different status items if desired.

You can use the percent_full attribute to manually set the status, or you can provide a pointer to a method that will provide the percentage information.

This is a kitchen sink class with several ways to use it; you may want to look at the subclasses built on top of it that are designed to be simpler to use, such as UIProgressBar, UIWorldSpaceHealthBar, and UIScreenSpaceHealthBar.

Parameters

- **relative_rect** -- The rectangle that defines the size of the health bar.
- **sprite** -- Optional sprite to monitor for status info, and for drawing the bar with the sprite.
- **follow_sprite** -- If there's a sprite, this indicates whether the bar should be drawn at the sprite's location.
- **percent_method** -- Optional method signature to call to get the percent complete. (To provide a method signature, simply reference the method without parenthesis, such as self.health_percent.)
- **manager** -- The UIManager that manages this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **container** -- The container that this element is within. If set to None will be the root window's container.
- **parent_element** -- The element this element 'belongs to' in the theming hierarchy.
- **object_id** -- A custom defined ID for fine-tuning of theming.
- **anchors** -- A dictionary describing what this element's relative_rect is relative to.

- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.

property percent_full

Use this property to directly change the status bar.

rebuild()

Rebuild the status bar entirely because the theming data has changed.

rebuild_from_changed_theme_data()

Called by the UIManager to check the theming data and rebuild whatever needs rebuilding for this element when the theme data has changed.

redraw()

Redraw the status bar when something, other than it's position has changed.

status_text()

To display text in the bar, subclass UIStatusBar and override this method.

update(*time_delta: float*)

Updates the status bar sprite's image and rectangle with the latest status and position data from the sprite we are monitoring

Parameters

time_delta -- time passed in seconds between one call to this method and the next.

class pygame_gui.elements.**UITabContainer**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *manager:* IUIManagerInterface | *None* = *None*, *container:* IContainerLikeInterface | *None* = *None*, ***, *starting_height: int = 1*, *parent_element:* IUIElementInterface | *None* = *None*, *object_id:* ObjectID | *str* | *None* = *None*, *anchors: Dict[str, str | IUIElementInterface]* | *None* = *None*, *visible: int = 1*)

Bases: *UIElement*

** EXPERIMENTAL ** A tab container. The displayed panel can be switched by clicking on the tab.

Parameters

- **relative_rect** -- Normally a rectangle describing the position (relative to its container) and dimensions. Also accepts a position Tuple, or Vector2 where the dimensions will be dynamic depending on the button's contents. Dynamic dimensions can be requested by setting the required dimension to -1.
- **manager** -- The UIManager that manages this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **container** -- The container that this element is within. If not provided or set to None will be the root window's container.
- **starting_height** -- The height in layers above its container that this element will be placed.
- **parent_element** -- The element this element 'belongs to' in the theming hierarchy.
- **object_id** -- A custom defined ID for fine-tuning of theming.
- **anchors** -- A dictionary describing what this element's relative_rect is relative to.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.

add_tab(*title_text: str*, *title_object_id: str*) → int

Create a new tab.

:return : the integer id of the newly created tab

disable()

Disables the window and it's contents so it is no longer interactive.

enable()

Enables the window and it's contents so it is interactive again.

hide()

In addition to the base UIElement.hide() - hide the _window_root_container which will propagate and hide all the children.

kill()

Overrides the basic kill() method of a pygame sprite so that we also kill all the UI elements in this panel.

process_event(*event: Event*)

Handles various interactions with the tab container.

Parameters

event -- The event to process.

Returns

Return True if we want to consume this event, so it is not passed on to the rest of the UI.

rebuild(*count: int | None = None*)

Rebuilds the tab container.

set_dimensions(*dimensions: Vector2 | Tuple[float, float]*, *clamp_to_container: bool = False*)

Set the size of this tab container.

Parameters

- **dimensions** -- The new dimensions to set.
- **clamp_to_container** -- Whether we should clamp the dimensions to the dimensions of the container or not.

show()

In addition to the base UIElement.show() - show the _window_root_container which will propagate and show all the children.

class pygame_gui.elements.**UITextBox**(*html_text: str*, *relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *manager: IUIManagerInterface | None = None*, *wrap_to_height: bool = False*, *starting_height: int = 1*, *container: IContainerLikeInterface | None = None*, *parent_element: UIElement | None = None*, *object_id: ObjectID | str | None = None*, *anchors: Dict[str, str | UIElement] | None = None*, *visible: int = 1*, ***, *pre_parsing_enabled: bool = True*, *text_kwargs: Dict[str, str] | None = None*, *allow_split_dashes: bool = True*, *plain_text_display_only: bool = False*, *should_html_unescape_input_text: bool = False*, *placeholder_text: str | None = None*)

Bases: *UIElement*, *IUITextOwnerInterface*

A Text Box element lets us display word-wrapped, formatted text. If the text to display is longer than the height of the box given then the element will automatically create a vertical scroll bar so that all the text can be seen.

Formatting the text is done via a subset of HTML tags. Currently supported tags are:

- or - to encase bold styled text.
- , or <var></var> - to encase italic styled text.
- <u></u> - to encase underlined text.
- ** - to encase 'link' text that can be clicked on to generate events with the** id given in href.
- <body bgcolor='#FFFFFF'></body> - to change the background colour of encased text.
-
 or - to start a new line.
- ** - To set the font, colour and size of** encased text.
- <hr> a horizontal rule.
- **<effect id='custom_id'></effect> - to encase text to that will have an effect applied. The custom id** supplied is used when setting an effect on this text box.
- **<shadow size=1 offset=1,1 color=#808080></shadow> - puts a shadow behind the text encased. Can also be used to** make a glow effect.
- **<img src="data/images/test_images/london.jpg" float=right> Inserts an image into the text with options on how to** float the text around the image. Float options are left, right or none.

More may be added in the future if needed or frequently requested.

NOTE: if dimensions of the initial containing rect are set to -1 the text box will match the final dimension to whatever the text rendering produces. This lets us make dynamically sized text boxes depending on their contents.

Parameters

- `html_text` -- The HTML formatted text to display in this text box.
- `relative_rect` -- The 'visible area' rectangle, positioned relative to its container.
- `manager` -- The UIManager that manages this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- `wrap_to_height` -- False by default, if set to True the box will increase in height to match the text within.
- `starting_height` -- Sets the height, above its container, to start placing the text box at.
- `container` -- The container that this element is within. If not provided or set to None will be the root window's container.
- `parent_element` -- The element this element 'belongs to' in the theming hierarchy.
- `object_id` -- A custom defined ID for fine-tuning of theming.
- `anchors` -- A dictionary describing what this element's relative_rect is relative to.
- `visible` -- Whether the element is visible by default. Warning - container visibility may override this.
- `pre_parsing_enabled` -- When enabled will replace all newline characters with html
 tags.
- `text_kwargs` -- A dictionary of variable arguments to pass to the translated text useful when you have multiple translations that need variables inserted in the middle.

- **allow_split_dashes** -- Sets whether long words that don't fit on a single line will be split with a dash or just split without a dash (more compact).
- **plain_text_display_only** -- No markup based styling & formatting will be done on the input text.
- **should_html_unescape_input_text** -- When enabled turns plain text encoded html back into html for this text box. e.g. < will become '<'
- **placeholder_text** -- If the text line is empty, and not focused, this placeholder text will be shown instead.

append_html_text(*new_html_str: str*)

Adds a string, that is parsed for any HTML tags that pygame_gui supports, onto the bottom of the text box's current contents.

This is useful for making things like logs.

Parameters
new_html_str -- The, potentially HTML tag, containing string of text to append.

clear_all_active_effects(*sub_chunk:* TextLineChunkFTFont | *None* = *None*)

Clears any active effects and redraws the text. A full reset, usually called before firing off a new effect if one is already in progress.

Parameters
sub_chunk -- An optional chunk so we only clear the effect from this chunk.

clear_text_surface(*sub_chunk:* TextLineChunkFTFont | *None* = *None*)

Clear the text surface

Parameters
sub_chunk -- An optional chunk so we only clear the surface for this chunk.

disable()

Disable the text box. Basically just disables the scroll bar if one exists.

enable()

Enable the text box. Re-enables the scroll bar if one exists.

focus()

Called when we 'select focus' on this element.

full_redraw()

Trigger a full redraw of the entire text box. Useful if we have messed with the text chunks in a more fundamental fashion and need to reposition them (say, if some of them have gotten wider after being made bold).

NOTE: This doesn't reparse the text of our box. If you need to do that, just create a new text box.

get_object_id() → str

The UI object ID of this text owner for use in effect events.

Returns
the ID string

get_text_letter_count(*sub_chunk:* TextLineChunkFTFont | *None* = *None*) → int

The amount of letters in the text

Parameters
sub_chunk -- An optional chunk to restrict the count to only this chunk.

Returns
number of letters as an int

`hide()`

In addition to the base UIElement.hide() - call hide() of scroll_bar if it exists.

`kill()`

Overrides the standard sprite kill method to also kill any scroll bars belonging to this text box.

`on_fresh_drawable_shape_ready()`

Called by an element's drawable shape when it has a new image surface ready for use, normally after a rebuilding/redrawing of some kind.

`on_locale_changed()`

Called for each element when the locale is changed on their UIManager

`parse_html_into_style_data()`

Parses HTML styled string text into a format more useful for styling rendered text.

`process_event`(*event: Event*) → bool

A stub to override. Gives UI Elements access to pygame events.

Parameters
event -- The event to process.

Returns
Should return True if this element makes use of this event.

`rebuild()`

Rebuild whatever needs building.

`rebuild_from_changed_theme_data()`

Called by the UIManager to check the theming data and rebuild whatever needs rebuilding for this element when the theme data has changed.

`redraw_from_chunks()`

Redraws from slightly earlier in the process than 'redraw_from_text_block'. Useful if we have redrawn individual chunks already (say, to change their style slightly after being hovered) and now want to update the text block with those changes without doing a full redraw.

This won't work very well if redrawing a chunk changed its dimensions.

`redraw_from_text_block()`

Redraws the final parts of the text box element that don't include redrawing the actual text. Useful if we've just moved the position of the text (say, with a scroll bar) without actually changing the text itself.

`property select_range`

The selected range for this text. A tuple containing the start and end indexes of the current selection.

Made into a property to keep it synchronised with the underlying drawable shape's representation.

`set_active_effect`(*effect_type:* UITextEffectType | *None* = *None*, *params: Dict[str, Any]* | *None* = *None*, *effect_tag: str* | *None* = *None*)

Set an animation effect to run on the text box. The effect will start running immediately after this call.

These effects are currently supported:

- TEXT_EFFECT_TYPING_APPEAR - Will look as if the text is being typed in.
- TEXT_EFFECT_FADE_IN - The text will fade in from the background colour.

- TEXT_EFFECT_FADE_OUT - The text will fade out to the background colour.

Parameters

- **effect_tag** -- if not None, only apply the effect to chunks with this tag.
- **params** -- Any parameters for the effect you are setting, if none are set defaults will be used.
- **effect_type** -- The type of the effect to set. If set to None instead it will cancel any active effect.

set_dimensions(*dimensions: Vector2 | Tuple[float, float]*, *clamp_to_container: bool = False*)

Method to directly set the dimensions of a text box.

Parameters

- **dimensions** -- The new dimensions to set.
- **clamp_to_container** -- Whether we should clamp the dimensions to the dimensions of the container or not.

set_position(*position: Vector2 | Tuple[float, float]*)

Sets the absolute screen position of this text box, updating its subordinate scroll bar at the same time.

Parameters
position -- The absolute screen position to set.

set_relative_position(*position: Vector2 | Tuple[float, float]*)

Sets the relative screen position of this text box, updating its subordinate scroll bar at the same time.

Parameters
position -- The relative screen position to set.

set_text_alpha(*alpha: int*, *sub_chunk:* TextLineChunkFTFont | *None = None*)

Set the global alpha value for the text

Parameters

- **alpha** -- the alpha to set.
- **sub_chunk** -- An optional chunk so we only set the alpha for this chunk.

set_text_offset_pos(*offset: Tuple[int, int]*, *sub_chunk:* TextLineChunkFTFont | *None = None*)

Move the text around by this offset.

Parameters

- **offset** -- the offset to set
- **sub_chunk** -- An optional chunk so we only set the offset for this chunk.

Returns

set_text_rotation(*rotation: int*, *sub_chunk:* TextLineChunkFTFont | *None = None*)

rotate the text by this int in degrees

Parameters

- **rotation** -- the rotation to set
- **sub_chunk** -- An optional chunk so we only set the rotation for this chunk.

Returns

set_text_scale(*scale: float*, *sub_chunk:* TextLineChunkFTFont | *None* = *None*)

Scale the text by this float

Parameters

- **scale** -- the scale to set
- **sub_chunk** -- An optional chunk so we only set the rotation for this chunk.

Returns

show()

In addition to the base UIElement.show() - call show() of scroll_bar if it exists.

stop_finished_effect(*sub_chunk:* TextLineChunkFTFont | *None* = *None*)

Stops a finished effect. Will leave effected text in the state it was in when effect ended. Used when an effect reaches a natural end where we might want to keep it in the end of effect state (e.g. a fade out)

Parameters

sub_chunk -- An optional chunk so we only clear the effect from this chunk.

unfocus()

Called when this element is no longer the current focus.

update(*time_delta: float*)

Called once every update loop of the UI Manager. Used to react to scroll bar movement (if there is one), update the text effect (if there is one) and check if we are hovering over any text links (if there are any).

Parameters

time_delta -- The time in seconds between calls to update. Useful for timing things.

update_text_effect(*time_delta: float*)

Update any active text effect on the text owner

Parameters

time_delta -- the time delta in seconds

update_text_end_position(*end_pos: int*, *sub_chunk:* TextLineChunkFTFont | *None* = *None*)

The position in the text to render up to.

Parameters

- **end_pos** -- The current end position as an int
- **sub_chunk** -- An optional chunk to restrict the end_position to only this chunk.

class pygame_gui.elements.**UITextEntryBox**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *initial_text: str = ''*, *manager:* IUIManagerInterface | *None* = *None*, *container:* IContainerLikeInterface | *None* = *None*, *parent_element:* UIElement | *None* = *None*, *object_id:* ObjectID | *str* | *None* = *None*, *anchors: Dict[str, str |* UIElement*]* | *None* = *None*, *visible: int = 1*, ***, *placeholder_text: str* | *None* = *None*)

Bases: *UITextBox*

Inherits from UITextBox but allows you to enter text with the keyboard, much like UITextEntryLine.

Parameters

- **relative_rect** -- The 'visible area' rectangle, positioned relative to its container.
- **initial_text** -- The text that starts in the text box.

- **manager** -- The UIManager that manages this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **container** -- The container that this element is within. If not provided or set to None will be the root window's container.
- **parent_element** -- The element this element 'belongs to' in the theming hierarchy.
- **object_id** -- A custom defined ID for fine-tuning of theming.
- **anchors** -- A dictionary describing what this element's relative_rect is relative to.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.

Param
placeholder_text: If the text line is empty, and not focused, this placeholder text will be shown instead.

focus()

Called when we 'select focus' on this element. In this case it sets up the keyboard to repeat held key presses, useful for natural feeling keyboard input.

get_text() → str

Gets the text in the entry box element.

Returns
A string.

process_event(*event: Event*) → bool

Allows the text entry box to react to input events, which is its primary function. The entry element reacts to various types of mouse clicks (double click selecting words, drag select), keyboard combos (CTRL+C, CTRL+V, CTRL+X, CTRL+A), individual editing keys (Backspace, Delete, Left & Right arrows) and other keys for inputting letters, symbols and numbers.

Parameters
event -- The current event to consider reacting to.

Returns
Returns True if we've done something with the input event.

redraw_from_text_block()

Redraws the final parts of the text box element that don't include redrawing the actual text. Useful if we've just moved the position of the text (say, with a scroll bar) without actually changing the text itself.

unfocus()

Called when this element is no longer the current focus.

update(*time_delta: float*)

Called every update loop of our UI Manager. Largely handles text drag selection and making sure our edit cursor blinks on and off.

Parameters
time_delta -- The time in seconds between this update method call and the previous one.

class pygame_gui.elements.**UITextEntryLine**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *manager: IUIManagerInterface | None = None*, *container: IContainerLikeInterface | None = None*, *parent_element: UIElement | None = None*, *object_id: ObjectID | str | None = None*, *anchors: Dict[str, str | UIElement] | None = None*, *visible: int = 1*, ***, *initial_text: str | None = None*, *placeholder_text: str | None = None*)

Bases: *UIElement*

A GUI element for text entry from a keyboard, on a single line. The element supports the standard copy and paste keyboard shortcuts CTRL+V, CTRL+C & CTRL+X as well as CTRL+A.

There are methods that allow the entry element to restrict the characters that can be input into the text box

The height of the text entry line element will be determined by the font used rather than the standard method for UIElements of just using the height of the input rectangle.

Parameters

- **relative_rect** -- A rectangle describing the position and width of the text entry element.
- **manager** -- The UIManager that manages this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **container** -- The container that this element is within. If not provided or set to None will be the root window's container.
- **parent_element** -- The element this element 'belongs to' in the theming hierarchy.
- **object_id** -- A custom defined ID for fine-tuning of theming.
- **anchors** -- A dictionary describing what this element's relative_rect is relative to.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.

Param
initial_text: A string that will start in this box. This text will remain when box is focused for editing.

Param
placeholder_text: If the text line is empty, and not focused, this placeholder text will be shown instead.

`disable()`

Disables the element so that it is no longer interactive.

`enable()`

Re-enables the element, so we can once again interact with it.

`focus()`

Called when we 'select focus' on this element. In this case it sets up the keyboard to repeat held key presses, useful for natural feeling keyboard input.

`get_text()` → str

Gets the text in the entry line element.

Returns
A string.

`on_locale_changed()`

Called for each element when the locale is changed on their UIManager

`process_event(event: Event)` → bool

Allows the text entry box to react to input events, which is its primary function. The entry element reacts to various types of mouse clicks (double click selecting words, drag select), keyboard combos (CTRL+C, CTRL+V, CTRL+X, CTRL+A), individual editing keys (Backspace, Delete, Left & Right arrows) and other keys for inputting letters, symbols and numbers.

Parameters
event -- The current event to consider reacting to.

Returns
Returns True if we've done something with the input event.

rebuild()

Rebuild whatever needs building.

rebuild_from_changed_theme_data()

Called by the UIManager to check the theming data and rebuild whatever needs rebuilding for this element when the theme data has changed.

redraw()

Redraws the entire text entry element onto the underlying sprite image. Usually called when the displayed text has been edited or changed in some fashion.

property select_range

The selected range for this text. A tuple containing the start and end indexes of the current selection.

Made into a property to keep it synchronised with the underlying drawable shape's representation.

set_allowed_characters(*allowed_characters: str | List[str]*)

Sets a whitelist of characters that will be the only ones allowed in our text entry element. We can either set the list directly, or request one of the already existing lists by a string identifier. The currently supported lists for allowed characters are:

- 'numbers'
- 'letters'
- 'alpha_numeric'

Parameters
allowed_characters -- The characters to allow, either in a list form or one of the supported string ids.

set_forbidden_characters(*forbidden_characters: str | List[str]*)

Sets a blacklist of characters that will be banned from our text entry element. We can either set the list directly, or request one of the already existing lists by a string identifier. The currently supported lists for forbidden characters are:

- 'numbers'
- 'forbidden_file_path'

Parameters
forbidden_characters -- The characters to forbid, either in a list form or one of the supported string ids.

set_text(*text: str*)

Allows the text displayed in the text entry element to be set via code. Useful for setting an initial or existing value that can be edited.

The string to set must be valid for the text entry element for this to work.

Parameters
text -- The text string to set.

set_text_hidden(*is_hidden=True*)

Passing in True will hide text typed into the text line, replacing it with characters and also disallow copying the text into the clipboard. It is designed for basic 'password box' usage.

Parameters

is_hidden -- Can be set to True or False. Defaults to True because if you are calling this you likely want a password box with no fuss. Set it back to False if you want to un-hide the text (e.g. for one of those 'Show my password' buttons).

set_text_length_limit(*limit: int*)

Allows a character limit to be set on the text entry element. By default, there is no limit on the number of characters that can be entered.

Parameters

limit -- The character limit as an integer.

unfocus()

Called when this element is no longer the current focus.

update(*time_delta: float*)

Called every update loop of our UI Manager. Largely handles text drag selection and making sure our edit cursor blinks on and off.

Parameters

time_delta -- The time in seconds between this update method call and the previous one.

validate_text_string(*text_to_validate: str*) → bool

Checks a string of text to see if any of its characters don't meet the requirements of the allowed and forbidden character sets.

Parameters

text_to_validate -- The text string to check.

class pygame_gui.elements.**UITooltip**(*html_text: str*, *hover_distance: Tuple[int, int]*, *manager:* IUIManagerInterface | *None* = *None*, *parent_element:* IUIElementInterface | *None* = *None*, *object_id:* ObjectID | *str* | *None* = *None*, *anchors: Dict[str, str* | UIElement*]* | *None* = *None*, ***, *wrap_width: int* | *None* = *None*, *text_kwargs: Dict[str, str]* | *None* = *None*)

Bases: *UIElement*, *IUITooltipInterface*

A tool tip is a floating block of text that gives additional information after a user hovers over an interactive part of a GUI for a short time. In Pygame GUI the tooltip's text is style-able with HTML.

At the moment the tooltips are only available as an option on UIButton elements.

Tooltips also don't allow a container as they are designed to overlap normal UI boundaries and be contained only within the 'root' window/container, which is synonymous with the pygame display surface.

Parameters

- **html_text** -- Text styled with HTML, to be displayed on the tooltip.
- **hover_distance** -- Distance in pixels between the tooltip and the thing being hovered.
- **manager** -- The UIManager that manages this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **parent_element** -- The element this element 'belongs to' in the theming hierarchy.
- **object_id** -- A custom defined ID for fine-tuning of theming.

- **anchors** -- A dictionary describing what this element's relative_rect is relative to.
- **text_kwargs** -- a dictionary of variable arguments to pass to the translated text useful when you have multiple translations that need variables inserted in the middle.

find_valid_position(*position: Vector2 | Tuple[float, float]*) → bool

Finds a valid position for the tool tip inside the root container of the UI.

The algorithm starts from the position of the target we are providing a tool tip for then it tries to fit the rectangle for the tool tip onto the screen by moving it above, below, to the left and to the right, until we find a position that fits the whole tooltip rectangle on the screen at once.

If we fail to manage this then the method will return False. Otherwise, it returns True and set the position of the tool tip to our valid position.

Parameters
position -- A 2D vector representing the position of the target this tool tip is for.

Returns
returns True if we find a valid (visible) position and False if we do not.

hide()

This is a base method hide() of a UIElement, but since it's not intended to be used on a UIToolTip - display a warning.

kill()

Overrides the UIElement's default kill method to also kill the text block element that helps make up the complete tool tip.

rebuild()

Rebuild anything that might need rebuilding.

rebuild_from_changed_theme_data()

Called by the UIManager to check the theming data and rebuild whatever needs rebuilding for this element when the theme data has changed.

set_dimensions(*dimensions: Vector2 | Tuple[float, float]*, *clamp_to_container: bool = False*)

Directly sets the dimensions of this tool tip. This will overwrite the normal theming.

Parameters

- **dimensions** -- The new dimensions to set.
- **clamp_to_container** -- Whether we should clamp the dimensions to the dimensions of the container or not.

set_position(*position: Vector2 | Tuple[float, float]*)

Sets the absolute screen position of this tool tip, updating its subordinate text box at the same time.

Parameters
position -- The absolute screen position to set.

set_relative_position(*position: Vector2 | Tuple[float, float]*)

Sets the relative screen position of this tool tip, updating its subordinate text box at the same time.

Parameters
position -- The relative screen position to set.

show()

This is a base method show() of a UIElement, but since it's not intended to be used on a UIToolTip - display a warning.

class pygame_gui.elements.**UIVerticalScrollBar**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *visible_percentage: float*, *manager: IUIManagerInterface | None = None*, *container: IContainerLikeInterface | None = None*, *parent_element: UIElement | None = None*, *object_id: ObjectID | str | None = None*, *anchors: Dict[str, str | UIElement] | None = None*, *visible: int = 1*)

Bases: *UIElement*

A vertical scroll bar allows users to position a smaller visible area within a vertically larger area.

Parameters

- **relative_rect** -- The size and position of the scroll bar.
- **visible_percentage** -- The vertical percentage of the larger area that is visible, between 0.0 and 1.0.
- **manager** -- The UIManager that manages this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **container** -- The container that this element is within. If not provided or set to None will be the root window's container.
- **parent_element** -- The element this element 'belongs to' in the theming hierarchy.
- **object_id** -- A custom defined ID for fine tuning of theming.
- **anchors** -- A dictionary describing what this element's relative_rect is relative to.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.

check_has_moved_recently() → bool

Returns True if the scroll bar was moved in the last call to the update function.

Returns

True if we've recently moved the scroll bar, False otherwise.

disable()

Disables the scroll bar so it is no longer interactive.

enable()

Enables the scroll bar so it is interactive once again.

hide()

In addition to the base UIElement.hide() - hide the self.button_container which will propagate and hide all the buttons.

kill()

Overrides the kill() method of the UI element class to kill all the buttons in the scroll bar and clear any of the parts of the scroll bar that are currently recorded as the 'last focused vertical scroll bar element' on the ui manager.

NOTE: the 'last focused' state on the UI manager is used so that the mouse wheel will move whichever scrollbar we last fiddled with even if we've been doing other stuff. This seems to be consistent with the most common mousewheel/scrollbar interactions used elsewhere.

process_event(*event: Event*) → bool

Checks an event from pygame's event queue to see if the scroll bar needs to react to it. In this case it is just mousewheel events, mainly because the buttons that make up the scroll bar will handle the required mouse click events.

Parameters
event -- The event to process.

Returns
Returns True if we've done something with the input event.

`rebuild()`

Rebuild anything that might need rebuilding.

`rebuild_from_changed_theme_data()`

Called by the UIManager to check the theming data and rebuild whatever needs rebuilding for this element when the theme data has changed.

`redraw_scrollbar()`

Redraws the 'scrollbar' portion of the whole UI element. Called when we change the visible percentage.

`reset_scroll_position()`

Reset the current scroll position back to the top.

`set_dimensions(dimensions: Vector2 | Tuple[float, float], clamp_to_container: bool = False)`

Method to directly set the dimensions of an element.

Parameters

- **dimensions** -- The new dimensions to set.
- **clamp_to_container** -- Whether we should clamp the dimensions to the dimensions of the container or not.

`set_position(position: Vector2 | Tuple[float, float])`

Sets the absolute screen position of this scroll bar, updating all subordinate button elements at the same time.

Parameters
position -- The absolute screen position to set.

`set_relative_position(position: Vector2 | Tuple[float, float])`

Sets the relative screen position of this scroll bar, updating all subordinate button elements at the same time.

Parameters
position -- The relative screen position to set.

`set_scroll_from_start_percentage(new_start_percentage: float)`

Set the scroll bar's scrolling position from a percentage between 0.0 and 1.0.

Parameters
new_start_percentage -- the percentage to set.

`set_visible_percentage(percentage: float)`

Sets the percentage of the total 'scrollable area' that is currently visible. This will affect the size of the scrollbar and should be called if the vertical size of the 'scrollable area' or the vertical size of the visible area change.

Parameters
percentage -- A float between 0.0 and 1.0 representing the percentage that is visible.

`show()`

In addition to the base UIElement.show() - show the self.button_container which will propagate and show all the buttons.

property start_percentage

turning start_percentage into a property, so we can round it to mitigate floating point errors

update(*time_delta: float*)

Called once per update loop of our UI manager. Deals largely with moving the scroll bar and updating the resulting 'start_percentage' variable that is then used by other 'scrollable' UI elements to control the point they start drawing.

Reacts to presses of the up and down arrow buttons, movement of the mouse wheel and dragging of the scroll bar itself.

Parameters

time_delta -- A float, roughly representing the time in seconds between calls to this method.

class pygame_gui.elements.**UIWindow**(*rect: Rect | FRect | Tuple[float, float, float, float]*, *manager: IUIManagerInterface | None = None*, *window_display_title: str = ''*, *element_id: List[str] | str | None = None*, *object_id: ObjectID | str | None = None*, *resizable: bool = False*, *visible: int = 1*, *draggable: bool = True*, ***, *ignore_shadow_for_initial_size_and_pos: bool = True*, *always_on_top: bool = False*)

Bases: *UIElement*, *IContainerLikeInterface*, *IWindowInterface*

A base class for window GUI elements, any windows should inherit from this class.

Parameters

- **rect** -- A rectangle, representing size and position of the window (including title bar, shadow and borders).
- **manager** -- The UIManager that manages this UIWindow. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **window_display_title** -- A string that will appear in the windows title bar if it has one.
- **element_id** -- An element ID for this window, if one is not supplied it defaults to 'window'.
- **object_id** -- An optional object ID for this window, useful for distinguishing different windows.
- **resizable** -- Whether this window is resizable or not, defaults to False.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.
- **draggable** -- Whether this window is draggable or not, defaults to True.

property always_on_top: bool

Whether the window is always above normal windows or not. :return:

are_contents_hovered() → bool

Are any of the elements in the container hovered? Used for handling mousewheel events.

Returns

True if one of the elements is hovered, False otherwise.

can_hover() → bool

Called to test if this window can be hovered.

change_layer(*new_layer: int*)

Move this window, and it's contents, to a new layer in the UI.

Parameters
new_layer -- The layer to move to.

check_clicked_inside_or_blocking(*event: Event*) → bool

A quick event check outside the normal event processing so that this window is brought to the front of the window stack if we click on any of the elements contained within it.

Parameters
event -- The event to check.

Returns
returns True if the event represents a click inside this window or the window is blocking.

check_hover(*time_delta: float*, *hovered_higher_element: bool*) → bool

For the window the only hovering we care about is the edges if this is a resizable window.

Parameters

- **time_delta** -- time passed in seconds between one call to this method and the next.
- **hovered_higher_element** -- Have we already hovered an element/window above this one?

disable()

Disables the window and it's contents so it is no longer interactive.

enable()

Enables the window and it's contents so it is interactive again.

get_container() → *IUIContainerInterface*

Returns the container that should contain all the UI elements in this window.

Return UIContainer
The window's container.

get_hovering_edge_id() → str

Gets the ID of the combination of edges we are hovering for use by the cursor system.

Returns
a string containing the edge combination ID (e.g. xy,yx,xl,xr,yt,yb)

get_layer_thickness() → int

The layer 'thickness' of this window/ :return: an integer

get_relative_mouse_pos()

Returns the current mouse position relative to the inside of this window.

If the cursor is outside the window contents area it returns None

Returns
tuple of relative mouse co-ords or None

get_top_layer() → int

Returns the 'highest' layer used by this window so that we can correctly place other windows on top of it.

Returns
The top layer for this window as a number (greater numbers are higher layers).

hide()

In addition to the base UIElement.hide() - hide the _window_root_container which will propagate and hide all the children.

kill()

Overrides the basic kill() method of a pygame sprite so that we also kill all the UI elements in this window, and remove if from the window stack.

on_close_window_button_pressed()

Override this method to call 'hide()' instead if you want to hide a window when the close button is pressed.

on_moved_to_front()

Called when a window is moved to the front of its stack.

process_event(*event: Event*) → bool

Handles resizing & closing windows. Gives UI Windows access to pygame events. Derived windows should super() call this class if they implement their own process_event method.

Parameters
event -- The event to process.

Return bool
Return True if this element should consume this event and not pass it to the rest of the UI.

rebuild()

Rebuilds the window when the theme has changed.

rebuild_from_changed_theme_data()

Called by the UIManager to check the theming data and rebuild whatever needs rebuilding for this element when the theme data has changed.

set_blocking(*state: bool*)

Sets whether this window being open should block clicks to the rest of the UI or not. Defaults to False.

Parameters
state -- True if this window should block mouse clicks.

set_dimensions(*dimensions: Vector2 | Tuple[float, float]*, *clamp_to_container: bool = False*)

Set the size of this window and then re-sizes and shifts the contents of the windows container to fit the new size.

Parameters

- **dimensions** -- The new dimensions to set.
- **clamp_to_container** -- Whether we should clamp the dimensions to the dimensions of the container or not.

set_display_title(*new_title: str*)

Set the title of the window.

Parameters
new_title -- The title to set.

set_position(*position: Vector2 | Tuple[float, float]*)

Method to directly set the absolute screen rect position of an element.

Parameters
position -- The new position to set.

set_relative_position(*position: Vector2 | Tuple[float, float]*)

Method to directly set the relative rect position of an element.

Parameters
position -- The new position to set.

should_use_window_edge_resize_cursor() → bool

Returns true if this window is in a state where we should display one of the resizing cursors

Returns
True if a resizing cursor is needed.

show()

In addition to the base UIElement.show() - show the _window_root_container which will propagate and show all the children.

update(*time_delta: float*)

A method called every update cycle of our application. Designed to be overridden by derived classes but also has a little functionality to make sure the window's layer 'thickness' is accurate and to handle window resizing.

Parameters
time_delta -- time passed in seconds between one call to this method and the next.

class pygame_gui.elements.**UIWorldSpaceHealthBar**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *sprite_to_monitor: Sprite | ExampleHealthSprite*, *manager: IUIManagerInterface | None = None*, *container: IContainerLikeInterface | None = None*, *parent_element: UIElement | None = None*, *object_id: ObjectID | str | None = None*, *anchors: Dict[str, str | UIElement] | None = None*, *visible: int = 1*)

Bases: *UIStatusBar*

A UI that will display a sprite's 'health_capacity' and their 'current_health' in 'world space' above the sprite. This means that the health bar will move with the camera and the sprite itself.

A sprite passed to this class must have the attributes 'health_capacity' and 'current_health'.

Parameters

- **relative_rect** -- The rectangle that defines the size of the health bar.
- **sprite_to_monitor** -- The sprite we are displaying the health of.
- **manager** -- The UIManager that manages this element. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **container** -- The container that this element is within. If not provided or set to None will be the root window's container.
- **parent_element** -- The element this element 'belongs to' in the theming hierarchy.
- **object_id** -- A custom defined ID for fine-tuning of theming.
- **anchors** -- A dictionary describing what this element's relative_rect is relative to.
- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.

class ExampleHealthSprite(**groups*)

Bases: Sprite

An example sprite with health instance attributes.

Parameters
groups -- Sprite groups to put the sprite in.

pygame_gui.windows package

Submodules

pygame_gui.windows.ui_colour_picker_dialog module

class pygame_gui.windows.ui_colour_picker_dialog.**UIColourChannelEditor**(*relative_rect: Rect | FRect | Tuple[float, float, float, float]*, *name: str*, *channel_index: int*, *value_range: Tuple[int, int]*, *initial_value: int*, *manager: IUIManagerInterface | None = None*, *container: IContainerLikeInterface | None = None*, *parent_element: UIElement | None = None*, *object_id: ObjectID | str | None = None*, *anchors: Dict[str, str | UIElement] | None = None*, *visible: int = 1*)

Bases: *UIElement*

This colour picker specific element lets us edit a single colour channel (Red, Green, Blue, Hue etc.). It's bundled along with the colour picker class because I don't see much use for it outside a colour picker, but it still seemed sensible to make a class for a pattern in the colour picker that is repeated six times.

Parameters

- **relative_rect** -- The relative rectangle for sizing and positioning the element, relative to the anchors.
- **name** -- Name for this colour channel, (e.g 'R:' or 'B:'). Used for the label.
- **channel_index** -- Index for the colour channel (e.g. red is 0, blue is 1, hue is also 0, saturation is 1)
- **value_range** -- Range of values for this channel (0 to 255 for R,G,B - 0 to 360 for hue, 0 to 100 for the rest)
- **initial_value** -- Starting value for this colour channel.
- **manager** -- The UI manager for the UI system. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **container** -- UI container for this element.
- **parent_element** -- An element to parent this element, used for theming hierarchies and events.
- **object_id** -- A specific theming/event ID for this element.
- **anchors** -- A dictionary of anchors used for setting up what this element's relative_rect is relative to.

- **visible** -- Whether the element is visible by default. Warning - container visibility may override this.

hide()

In addition to the base UIElement.hide() - call hide() of the element_container - which will propagate to the sub-elements - label, entry and slider.

process_event(*event: Event*) → bool

Handles events that this UI element is interested in. In this case we are responding to the slider being moved and the user finishing entering text in the text entry element.

Parameters
event -- The pygame Event to process.

Returns
True if event is consumed by this element and should not be passed on to other elements.

set_dimensions(*dimensions: Vector2 | Tuple[float, float]*, *clamp_to_container: bool = False*)

Method to directly set the dimensions of an element.

Parameters

- **dimensions** -- The new dimensions to set.
- **clamp_to_container** -- Whether we should clamp the dimensions to the dimensions of the container or not.

set_position(*position: Vector2 | Tuple[float, float]*)

Sets the absolute screen position of this channel, updating all subordinate elements at the same time.

Parameters
position -- The absolute screen position to set.

set_relative_position(*position: Vector2 | Tuple[float, float]*)

Sets the relative screen position of this channel, updating all subordinate elements at the same time.

Parameters
position -- The relative screen position to set.

set_value(*new_value: int*)

For when we need to set the value of the colour channel from outside, usually from adjusting the colour elsewhere in the colour picker. Makes sure the new value is within the allowed range.

Parameters
new_value -- Value to set.

show()

In addition to the base UIElement.show() - call show() of the element_container - which will propagate to the sub-elements - label, entry and slider.

class pygame_gui.windows.ui_colour_picker_dialog.**UIColourPickerDialog**(*rect: Rect | FRect | Tuple[float, float, float, float]*, *manager: IUIManagerInterface | None = None*, ***, *initial_colour: Color = Color(0, 0, 0, 255)*, *window_title: str = 'pygame-gui.colour_picker_title_bar'*, *object_id: ObjectID | str = ObjectID(object_id='#colour_picker_dialog', class_id=None)*, *visible: int = 1*, *always_on_top: bool = False*)

Bases: *UIWindow*

A colour picker window that gives us a small range of UI tools to pick a final colour.

Parameters

- **rect** -- The size and position of the colour picker window. Includes the size of shadow, border and title bar.
- **manager** -- The manager for the whole of the UI. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **initial_colour** -- The starting colour for the colour picker, defaults to black.
- **window_title** -- The title for the window, defaults to 'Colour Picker'
- **object_id** -- The object ID for the window, used for theming - defaults to '#colour_picker_dialog'
- **visible** -- Whether the element is visible by default.

changed_hsv_update_rgb()

Updates the RGB channels when we've altered the HSV ones.

changed_rgb_update_hsv()

Updates the HSV channels when we've altered the RGB ones.

get_colour() → Color

Get the current colour :return:

process_event(*event: Event*) → bool

Handles events that this UI element is interested in. In this case we are responding to the colour channel elements being changed, the OK or Cancel buttons being pressed or the user clicking the mouse inside the Saturation & Value picking square.

Parameters
event -- The pygame Event to process.

Returns
True if event is consumed by this element and should not be passed on to other elements.

set_colour(*colour: Color*) → None

Set the current colour and update all the necessary elements :param colour: The colour to set :return: None

update_colour_2d_slider()

This is used to update the 2D slider from the sliders :return: None

update_current_colour_image()

Updates the 'current colour' image when the current colour has been changed.

update_saturation_value_square()

Updates the appearance of the big square that lets us visually pick the Saturation and Value of our current Hue. This is done by drawing a very small 4x4 pixel square with a pattern like so:

[black] [hue at max saturation & value] [black] [white]

And then using the smoothscale transform to enlarge it so that the colours blend smoothly from one to the other.

pygame_gui.windows.ui_confirmation_dialog module

class pygame_gui.windows.ui_confirmation_dialog.**UIConfirmationDialog**(*rect: Rect | FRect | Tuple[float, float, float, float]*, *action_long_desc: str*, *manager: IUIManagerInterface | None = None*, ***, *window_title: str = 'pygame-gui.Confirm'*, *action_short_name: str = 'pygame-gui.OK'*, *blocking: bool = True*, *object_id: ObjectID | str = ObjectID(object_id='#confirmation_dialog', class_id=None)*, *visible: int = 1*, *action_long_desc_text_kwargs: Dict[str, str] | None = None*, *always_on_top: bool = False*)

Bases: *UIWindow*

A confirmation dialog that lets a user choose between continuing on a path they've chosen or cancelling. It's good practice to give a very brief description of the action they are confirming on the button they click to confirm it i.e. 'Delete' for a file deletion operation or, 'Rename' for a file rename operation.

Parameters

- **rect** -- The size and position of the window, includes the menu bar across the top.
- **action_long_desc** -- Long-ish description of action. Can make use of HTML to style the text.
- **manager** -- The UIManager that manages this UIElement. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **window_title** -- The title of the window.
- **action_short_name** -- Short, one or two-word description of action for button.

- **blocking** -- Whether this window should block all other mouse interactions with the GUI until it is closed.
- **object_id** -- A custom defined ID for fine-tuning of theming. Defaults to '#confirmation_dialog'.
- **visible** -- Whether the element is visible by default.
- **action_long_desc_text_kwargs** -- a dictionary of variable arguments to pass to the translated string useful when you have multiple translations that need variables inserted in the middle.

process_event(*event: Event*) → bool

Process any events relevant to the confirmation dialog.

We close the window when the cancel button is pressed, and post a confirmation event (UI_CONFIRMATION_DIALOG_CONFIRMED) when the OK button is pressed, and also close the window.

Parameters
event -- a pygame.Event.

Returns
Return True if we 'consumed' this event and don't want to pass it on to the rest of the UI.

pygame_gui.windows.ui_console_window module

class pygame_gui.windows.ui_console_window.**UIConsoleWindow**(*rect: Rect | FRect | Tuple[float, float, float, float]*, *manager: IUIManagerInterface | None = None*, *window_title: str = 'pygame-gui.console_title_bar'*, *object_id: ObjectID | str = ObjectID(object_id='#console_window', class_id=None)*, *visible: int = 1*, *preload_bold_log_font: bool = True*, *always_on_top: bool = False*)

Bases: *UIWindow*

Create a basic console window. By default, it doesn't do anything except log commands entered into the console in the text box log. There is an event and a few methods however that allow you to hook this console window up to do whatever you would like with the entered text commands.

See the pygame GUI examples repository for an example using it to run the Python Interactive Shell.

Parameters

- **rect** -- A rect determining the size and position of the console window.
- **manager** -- The UI manager. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **window_title** -- The title displayed in the windows title bar.
- **object_id** -- The object ID for the window, used for theming - defaults to '#console_window'
- **visible** -- Whether the element is visible by default.

add_output_line_to_log(*text_to_add: str*, *is_bold: bool = True*, *remove_line_break: bool = False*, *escape_html: bool = True*) → None

Adds a single line of text to the log text box. This is intended as a hook to add output/responses to commands entered into the console.

Parameters

- **text_to_add** -- The single line of text to add to the log.
- **is_bold** -- Determines whether the output is shown in bold or not. Defaults to True.
- **remove_line_break** -- Set this to True to remove the automatic line break at the end of the line of text. Sometimes you might want to add some output all on a single line (e.g. a console style 'progress bar')
- **escape_html** -- Determines whether to escape any HTML in this line of text. Defaults to True, as most people won't be expecting every > or < to be processed as HTML.

clear_log()

Clear the console log, re-starting with a fresh, empty console with no old commands.

process_event(*event: Event*) → bool

See if we need to handle an event passed down by the UI manager. Returns True if the console window dealt with this event.

Parameters

event -- The event to handle

restore_default_prefix() → None

Restore the console log prefix to its default value (which is: '> ')

set_log_prefix(*prefix: str*) → None

Set the prefix to add before commands when they are displayed in the log. This defaults to '> '.

Parameters

prefix -- A string that is prepended to commands before they are added to the log text box.

set_logged_commands_escape_html(*should_escape: bool*) → None

Sets whether commands should have any HTML characters escaped before being added to the log. This is because the log uses an HTML Text box and most of the time we don't want to assume that every entered > or < is part of an HTML tag.

By default, HTML is escaped for commands added to the log.

Parameters

should_escape -- A bool to switch escaping HTML on commands on or off.

pygame_gui.windows.ui_file_dialog module

class pygame_gui.windows.ui_file_dialog.**UIFileDialog**(*rect: Rect | FRect | Tuple[float, float, float, float]*, *manager: IUIManagerInterface | None*, *window_title: str = 'pygame-gui.file_dialog_title_bar'*, *allowed_suffixes: Set[str] | None = None*, *initial_file_path: str | None = None*, *object_id: ObjectID | str = ObjectID(object_id='#file_dialog', class_id=None)*, *allow_existing_files_only: bool = False*, *allow_picking_directories: bool = False*, *visible: int = 1*, *always_on_top: bool = False*)

Bases: *UIWindow*

A dialog window for handling file selection operations. The dialog will let you pick a file from a file system but won't do anything with it once you have, the path will just be returned leaving it up to the rest of the application to decide what to do with it.

Parameters

- **rect** -- The size and position of the file dialog window. Includes the size of shadow, border and title bar.
- **manager** -- The manager for the whole of the UI. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **window_title** -- The title for the window, defaults to 'File Dialog'
- **initial_file_path** -- The initial path to open the file dialog at.
- **object_id** -- The object ID for the window, used for theming - defaults to '#file_dialog'
- **visible** -- Whether the element is visible by default.

process_event(*event: Event*) → bool

Handles events that this UI element is interested in. There are a lot of buttons in the file dialog.

Parameters

event -- The pygame Event to process.

Returns

True if event is consumed by this element and should not be passed on to other elements.

update_current_file_list()

Updates the currently displayed list of files and directories. Usually called when the directory path has changed.

pygame_gui.windows.ui_message_window module

class pygame_gui.windows.ui_message_window.**UIMessageWindow**(*rect: Rect | FRect | Tuple[float, float, float, float]*, *html_message: str*, *manager: IUIManagerInterface | None = None*, ***, *window_title: str = 'pygame-gui.message_window_title_bar'*, *object_id: ObjectID | str = ObjectID(object_id='#message_window', class_id=None)*, *visible: int = 1*, *html_message_text_kwargs: Dict[str, str] | None = None*, *always_on_top: bool = False*)

Bases: *UIWindow*

A simple popup window for delivering text-only messages to users.

Parameters

- **rect** -- The size and position of the window, includes the menu bar across the top.
- **html_message** -- The message itself. Can make use of HTML (a subset of) to style the text.
- **manager** -- The UIManager that manages this UIElement. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **window_title** -- The title of the window.
- **object_id** -- A custom defined ID for fine-tuning of theming. Defaults to '#message_window'.
- **visible** -- Whether the element is visible by default.

process_event(*event: Event*) → bool

Process any events relevant to the message window. In this case we just close the window when the dismiss button is pressed.

Parameters
event -- a pygame.Event.

Returns
Return True if we 'consumed' this event and don't want to pass it on to the rest of the UI.

Module contents

class pygame_gui.windows.**UIColourPickerDialog**(*rect: Rect | FRect | Tuple[float, float, float, float]*, *manager: IUIManagerInterface | None = None*, ***, *initial_colour: Color = Color(0, 0, 0, 255)*, *window_title: str = 'pygame-gui.colour_picker_title_bar'*, *object_id: ObjectID | str = ObjectID(object_id='#colour_picker_dialog', class_id=None)*, *visible: int = 1*, *always_on_top: bool = False*)

Bases: *UIWindow*

A colour picker window that gives us a small range of UI tools to pick a final colour.

Parameters

- **rect** -- The size and position of the colour picker window. Includes the size of shadow, border and title bar.
- **manager** -- The manager for the whole of the UI. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **initial_colour** -- The starting colour for the colour picker, defaults to black.
- **window_title** -- The title for the window, defaults to 'Colour Picker'
- **object_id** -- The object ID for the window, used for theming - defaults to '#colour_picker_dialog'
- **visible** -- Whether the element is visible by default.

changed_hsv_update_rgb()

Updates the RGB channels when we've altered the HSV ones.

changed_rgb_update_hsv()

Updates the HSV channels when we've altered the RGB ones.

get_colour() → Color

Get the current colour :return:

process_event(*event: Event*) → bool

Handles events that this UI element is interested in. In this case we are responding to the colour channel elements being changed, the OK or Cancel buttons being pressed or the user clicking the mouse inside the Saturation & Value picking square.

Parameters
event -- The pygame Event to process.

Returns
True if event is consumed by this element and should not be passed on to other elements.

set_colour(*colour: Color*) → None

Set the current colour and update all the necessary elements :param colour: The colour to set :return: None

update_colour_2d_slider()

This is used to update the 2D slider from the sliders :return: None

update_current_colour_image()

Updates the 'current colour' image when the current colour has been changed.

update_saturation_value_square()

Updates the appearance of the big square that lets us visually pick the Saturation and Value of our current Hue. This is done by drawing a very small 4x4 pixel square with a pattern like so:

[black] [hue at max saturation & value] [black] [white]

And then using the smoothscale transform to enlarge it so that the colours blend smoothly from one to the other.

class pygame_gui.windows.**UIConfirmationDialog**(*rect: Rect | FRect | Tuple[float, float, float, float]*, *action_long_desc: str*, *manager:* IUIManagerInterface | *None = None*, ***, *window_title: str = 'pygame-gui.Confirm'*, *action_short_name: str = 'pygame-gui.OK'*, *blocking: bool = True*, *object_id:* ObjectID | *str = ObjectID(object_id='#confirmation_dialog', class_id=None)*, *visible: int = 1*, *action_long_desc_text_kwargs: Dict[str, str] | None = None*, *always_on_top: bool = False*)

Bases: *UIWindow*

A confirmation dialog that lets a user choose between continuing on a path they've chosen or cancelling. It's good practice to give a very brief description of the action they are confirming on the button they click to confirm it i.e. 'Delete' for a file deletion operation or, 'Rename' for a file rename operation.

Parameters

- **rect** -- The size and position of the window, includes the menu bar across the top.
- **action_long_desc** -- Long-ish description of action. Can make use of HTML to style the text.
- **manager** -- The UIManager that manages this UIElement. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **window_title** -- The title of the window.
- **action_short_name** -- Short, one or two-word description of action for button.
- **blocking** -- Whether this window should block all other mouse interactions with the GUI until it is closed.
- **object_id** -- A custom defined ID for fine-tuning of theming. Defaults to '#confirmation_dialog'.
- **visible** -- Whether the element is visible by default.
- **action_long_desc_text_kwargs** -- a dictionary of variable arguments to pass to the translated string useful when you have multiple translations that need variables inserted in the middle.

process_event(*event: Event*) → bool

Process any events relevant to the confirmation dialog.

We close the window when the cancel button is pressed, and post a confirmation event (UI_CONFIRMATION_DIALOG_CONFIRMED) when the OK button is pressed, and also close the window.

Parameters

event -- a pygame.Event.

Returns

Return True if we 'consumed' this event and don't want to pass it on to the rest of the UI.

class pygame_gui.windows.**UIConsoleWindow**(*rect: Rect | FRect | Tuple[float, float, float, float]*, *manager:* IUIManagerInterface | *None = None*, *window_title: str = 'pygame-gui.console_title_bar'*, *object_id:* ObjectID | *str = ObjectID(object_id='#console_window', class_id=None)*, *visible: int = 1*, *preload_bold_log_font: bool = True*, *always_on_top: bool = False*)

Bases: *UIWindow*

Create a basic console window. By default, it doesn't do anything except log commands entered into the console in the text box log. There is an event and a few methods however that allow you to hook this console window up to do whatever you would like with the entered text commands.

See the pygame GUI examples repository for an example using it to run the Python Interactive Shell.

Parameters

- **rect** -- A rect determining the size and position of the console window.
- **manager** -- The UI manager. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **window_title** -- The title displayed in the windows title bar.
- **object_id** -- The object ID for the window, used for theming - defaults to '#console_window'
- **visible** -- Whether the element is visible by default.

add_output_line_to_log(*text_to_add: str*, *is_bold: bool = True*, *remove_line_break: bool = False*, *escape_html: bool = True*) → None

Adds a single line of text to the log text box. This is intended as a hook to add output/responses to commands entered into the console.

Parameters

- **text_to_add** -- The single line of text to add to the log.
- **is_bold** -- Determines whether the output is shown in bold or not. Defaults to True.
- **remove_line_break** -- Set this to True to remove the automatic line break at the end of the line of text. Sometimes you might want to add some output all on a single line (e.g. a console style 'progress bar')
- **escape_html** -- Determines whether to escape any HTML in this line of text. Defaults to True, as most people won't be expecting every > or < to be processed as HTML.

clear_log()

Clear the console log, re-starting with a fresh, empty console with no old commands.

process_event(*event: Event*) → bool

See if we need to handle an event passed down by the UI manager. Returns True if the console window dealt with this event.

Parameters

event -- The event to handle

restore_default_prefix() → None

Restore the console log prefix to its default value (which is: '> ')

set_log_prefix(*prefix: str*) → None

Set the prefix to add before commands when they are displayed in the log. This defaults to '> '.

Parameters

prefix -- A string that is prepended to commands before they are added to the log text box.

set_logged_commands_escape_html(*should_escape: bool*) → None

Sets whether commands should have any HTML characters escaped before being added to the log. This is because the log uses an HTML Text box and most of the time we don't want to assume that every entered > or < is part of an HTML tag.

By default, HTML is escaped for commands added to the log.

Parameters

should_escape -- A bool to switch escaping HTML on commands on or off.

class pygame_gui.windows.**UIFileDialog**(*rect: Rect | FRect | Tuple[float, float, float, float]*, *manager: IUIManagerInterface | None*, *window_title: str = 'pygame-gui.file_dialog_title_bar'*, *allowed_suffixes: Set[str] | None = None*, *initial_file_path: str | None = None*, *object_id: ObjectID | str = ObjectID(object_id='#file_dialog', class_id=None)*, *allow_existing_files_only: bool = False*, *allow_picking_directories: bool = False*, *visible: int = 1*, *always_on_top: bool = False*)

Bases: *UIWindow*

A dialog window for handling file selection operations. The dialog will let you pick a file from a file system but won't do anything with it once you have, the path will just be returned leaving it up to the rest of the application to decide what to do with it.

Parameters

- **rect** -- The size and position of the file dialog window. Includes the size of shadow, border and title bar.
- **manager** -- The manager for the whole of the UI. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **window_title** -- The title for the window, defaults to 'File Dialog'
- **initial_file_path** -- The initial path to open the file dialog at.
- **object_id** -- The object ID for the window, used for theming - defaults to '#file_dialog'
- **visible** -- Whether the element is visible by default.

process_event(*event: Event*) → bool

Handles events that this UI element is interested in. There are a lot of buttons in the file dialog.

Parameters

event -- The pygame Event to process.

Returns

True if event is consumed by this element and should not be passed on to other elements.

update_current_file_list()

Updates the currently displayed list of files and directories. Usually called when the directory path has changed.

class pygame_gui.windows.**UIMessageWindow**(*rect: Rect | FRect | Tuple[float, float, float, float]*, *html_message: str*, *manager: IUIManagerInterface | None = None*, ***, *window_title: str = 'pygame-gui.message_window_title_bar'*, *object_id: ObjectID | str = ObjectID(object_id='#message_window', class_id=None)*, *visible: int = 1*, *html_message_text_kwargs: Dict[str, str] | None = None*, *always_on_top: bool = False*)

Bases: *UIWindow*

A simple popup window for delivering text-only messages to users.

Parameters

- **rect** -- The size and position of the window, includes the menu bar across the top.
- **html_message** -- The message itself. Can make use of HTML (a subset of) to style the text.
- **manager** -- The UIManager that manages this UIElement. If not provided or set to None, it will try to use the first UIManager that was created by your application.
- **window_title** -- The title of the window.
- **object_id** -- A custom defined ID for fine-tuning of theming. Defaults to '#message_window'.
- **visible** -- Whether the element is visible by default.

process_event(*event: Event*) → bool

Process any events relevant to the message window. In this case we just close the window when the dismiss button is pressed.

Parameters
event -- a pygame.Event.

Returns
Return True if we 'consumed' this event and don't want to pass it on to the rest of the UI.

Submodules

pygame_gui.ui_manager module

class pygame_gui.ui_manager.**UIManager**(*window_resolution: Tuple[int, int]*, *theme_path: str | PathLike | StringIO | PackageResource | dict | None = None*, *enable_live_theme_updates: bool = True*, *resource_loader: IResourceLoader | None = None*, *starting_language: str = 'en'*, *translation_directory_paths: List[str] | None = None*)

Bases: *IUIManagerInterface*

The UI Manager class helps keep track of all the moving parts in the pygame_gui system.

Before doing anything else with pygame_gui create a UIManager and remember to update it every frame.

Parameters

- **window_resolution** -- window resolution.
- **theme_path** -- relative file path to theme or theme dictionary.
- **enable_live_theme_updates** -- Lets the theme update in-game after we edit the theme file

add_font_paths(*font_name: str*, *regular_path: str*, *bold_path: str | None = None*, *italic_path: str | None = None*, *bold_italic_path: str | None = None*)

Add file paths for custom fonts you want to use in the UI. For each font name you add you can specify font files for different styles. Fonts with designed styles tend to render a lot better than fonts that are forced to make use of pygame's bold and italic styling effects, so if you plan to use bold and italic text at small sizes - find fonts with these styles available as separate files.

The font name you specify here can be used to choose the font in the blocks of HTML-subset formatted text, available in some of the UI elements like the UITextBox.

It is recommended that you also preload any fonts you use at an appropriate moment in your project rather than letting the library dynamically load them when they are required. That is because dynamic loading of large font files can cause UI elements with a lot of font usage to appear rather slowly as they are waiting for the fonts they need to load.

Parameters

- **font_name** -- The name of the font that will be used to reference it elsewhere in the GUI.
- **regular_path** -- The path of the font file for this font with no styles applied.
- **bold_path** -- The path of the font file for this font with just bold style applied.
- **italic_path** -- The path of the font file for this font with just italic style applied.
- **bold_italic_path** -- The path of the font file for this font with bold & italic style applied.

calculate_scaled_mouse_position(*position: Tuple[int, int]*) → Tuple[int, int]

Scaling an input mouse position by a scale factor.

clear_and_reset()

Clear all existing windows and the root container, which should get rid of all created UI elements. We then recreate the UIWindowStack and the root container.

create_new_theme(*theme_path: str | PathLike | StringIO | PackageResource | dict | None = None*) → *UIAppearanceTheme*

Create a new theme using self information. :param theme_path: relative file path to theme or theme dictionary.

create_tool_tip(*text: str*, *position: Tuple[int, int]*, *hover_distance: Tuple[int, int]*, *parent_element:* IUIElementInterface, *object_id:* ObjectID, *, *wrap_width: int | None = None*, *text_kwargs: Dict[str, str] | None = None*) → *IUITooltipInterface*

Creates a tool tip ands returns it. Have hidden this away in the manager, so we can call it from other UI elements and create tool tips without creating cyclical import problems.

Parameters

- **text** -- The tool tips text, can utilise the HTML subset used in all UITextBoxes.
- **position** -- The screen position to create the tool tip for.
- **hover_distance** -- The distance we should hover away from our target position.
- **parent_element** -- The UIElement that spawned this tool tip.
- **object_id** -- the object_id of the tooltip.
- **wrap_width** -- an optional width for the tool tip, will overwrite any value from the theme file.
- **text_kwargs** -- a dictionary of variable arguments to pass to the translated string useful when you have multiple translations that need variables inserted in the middle.

Returns

A tool tip placed somewhere on the screen.

draw_ui(*window_surface: Surface*)

Draws all the UI elements on the screen. Generally you want this to be after the rest of your game sprites have been drawn.

If you want to do something particularly unusual with drawing you may have to write your own UI manager.

Parameters
window_surface -- The screen or window surface on which we are going to draw all of our UI Elements.

get_double_click_time() → float

Returns time between clicks that counts as a double click.

Returns
A float, time measured in seconds.

get_focus_set()

Gets the focused set.

Returns
The set of elements that currently have interactive focus. If None, nothing is currently focused.

get_hovering_any_element() → bool

True if any UI element (other than the root container) is hovered by the mouse.

Combined with 'get_focus_set()' and the return value from process_events(), it should make it easier to switch input events between the UI and other parts of an application.

get_locale()

Get the locale language code being used in the UIManager

Returns
A two-letter ISO 639-1 code for the current locale.

get_mouse_position() → Tuple[int, int]

Wrapping pygame mouse position, so we can mess with it.

get_root_container() → *IUIContainerInterface*

Returns the 'root' container. The one all UI elements are placed in by default if they are not placed anywhere else, fills the whole OS/pygame window.

Returns
A container.

get_shadow(*size: Tuple[int, int]*, *shadow_width: int = 2*, *shape: str = 'rectangle'*, *corner_radius: List[int] | None = None*) → Surface

Returns a 'shadow' surface scaled to the requested size.

Parameters

- **size** -- The size of the object we are shadowing + it's shadow.
- **shadow_width** -- The width of the shadowed edge.
- **shape** -- The shape of the requested shadow.
- **corner_radius** -- The radius of the shadow corners if this is a rectangular shadow.

Returns
A shadow as a pygame Surface.

get_sprite_group() → *LayeredGUIGroup*

Gets the sprite group used by the entire UI to keep it in the correct order for drawing and processing input.

Returns
The UI's sprite group.

get_theme() → *IUIAppearanceThemeInterface*

Gets the theme so the data in it can be accessed.

Returns
The theme data used by this UIManager

get_universal_empty_surface() → Surface

Sometimes we want to hide sprites or just have sprites with no visual component, when we do we can just use this empty surface to save having lots of empty surfaces all over memory.

Returns
An empty and therefore invisible pygame.surface.Surface

get_window_stack() → *IUIWindowStackInterface*

The UIWindowStack organises any windows in the UI Manager so that they are correctly sorted and move windows we interact with to the top of the stack.

Returns
The stack of windows.

preload_fonts(*font_list: List[Dict[str, str | int | float]]*)

It's a good idea to preload the exact fonts your program uses during the loading phase of your program. By default, the pygame_gui library will still work, but will spit out reminder warnings when you haven't done this. Loading fonts on the fly will slow down the apparent responsiveness when creating UI elements that use a lot of different fonts.

To preload custom fonts, or to use custom fonts at all (i.e. ones that aren't the default 'noto_sans' font) you must first add the paths to the files for those fonts, then load the specific fonts with a list of font descriptions in a dictionary form like so:

```
{'name': 'noto_sans', 'point_size': 12, 'style': 'bold_italic',
'antialiased': 1}
```

You can specify size either in pygame.Font point sizes with 'point_size', or in HTML style sizes with 'html_size'. Style options are:

- 'regular'
- 'italic'
- 'bold'
- 'bold_italic'

The name parameter here must match the one you used when you added the file paths.

Parameters
font_list -- A list of font descriptions in dictionary format as described above.

print_layer_debug()

Print some formatted information on the current state of the UI Layers.

Handy for debugging layer problems.

print_unused_fonts()

Helps you identify which preloaded fonts you are actually still using in your project after you've fiddled around with the text a lot by printing out a list of fonts that have not been used yet at the time this function is called.

Of course if you don't run the code path in which a particular font is used before calling this function then it won't be of much use, so take its results under advisement rather than as gospel.

process_events(*event: Event*)

This is the top level method through which all input to UI elements is processed and reacted to.

One of the key things it controls is the currently 'focused' element of which there can be only one at a time. It also manages 'consumed events' these events will not be passed on to elements below them in the GUI hierarchy and helps us stop buttons underneath windows from receiving input.

Parameters
event -- pygame.event.Event - the event to process.

Returns
A boolean indicating whether the event was consumed.

set_active_cursor(*cursor: Cursor*)

This is for users of the library to set the currently active cursor, it will be currently only be overridden by the resizing cursors.

The expected input is a pygame.cursors.Cursor:

```
manager.set_active_cursor(pygame.cursors.Cursor(pygame.SYSTEM_CURSOR_ARROW))
```

set_focus_set(*focus:* IUIElementInterface | *Set[*IUIElementInterface*]* | *None*)

Set a set of element as the focused set.

Parameters
focus -- The set of element to focus on.

set_locale(*locale: str*)

Set a locale language code to use in the UIManager

Parameters
locale -- A two letter ISO 639-1 code for a supported language.

TODO: Make this raise an exception for an unsupported language?

set_text_hovered(*hovering_text: bool*)

Set to true when hovering an area containing selectable text.

Currently, switches the cursor to the I-Beam cursor.

Parameters
hovering_text_input -- set to True to toggle the I-Beam cursor

set_ui_theme(*theme:* IUIAppearanceThemeInterface, *update_all_sprites: bool = False*)

Set ui theme.

Parameters

- **theme** -- The theme to set.
- **update_all_sprites** --

set_visual_debug_mode(*is_active: bool*)

Loops through all our UIElements to turn visual debug mode on or off. Also calls print_layer_debug()

Parameters
is_active -- True to activate visual debug and False to turn it off.

set_window_resolution(*window_resolution: Tuple[int, int]*)

Sets the window resolution.

Parameters
window_resolution -- the resolution to set.

update(*time_delta: float*)

From here all our UI elements are updated and which element is currently 'hovered' is checked; which means the mouse pointer is overlapping them. This is managed centrally, so we aren't ever overlapping two elements at once.

It also updates the shape cache to continue storing already created elements shapes in the long term cache, in case we need them later.

Finally, if live theme updates are enabled, it checks to see if the theme file has been modified and triggers all the UI elements to rebuild if it has.

Parameters
time_delta -- The time passed since the last call to update, in seconds.

Module contents

Pygame GUI module

Provides bits and bobs of UI to help make more complicated interactions with games built in pygame easier to accomplish.

class pygame_gui.**PackageResource**(*package: str*, *resource: str*)

Bases: object

A data class to handle input for importlib.resources as single parameter.

Parameters

- **package** -- The python package our resource is located in (e.g. 'pygame_gui.data')
- **resource** -- The name of the resource (e.g. 'default_theme.json')

to_path() → str

If we don't have any importlib module to use, we can try to turn the resource into a file path.

Returns
A string path.

class pygame_gui.**UIManager**(*window_resolution: Tuple[int, int]*, *theme_path: str | PathLike | StringIO | PackageResource | dict | None = None*, *enable_live_theme_updates: bool = True*, *resource_loader: IResourceLoader | None = None*, *starting_language: str = 'en'*, *translation_directory_paths: List[str] | None = None*)

Bases: *IUIManagerInterface*

The UI Manager class helps keep track of all the moving parts in the pygame_gui system.

Before doing anything else with pygame_gui create a UIManager and remember to update it every frame.

Parameters

- **window_resolution** -- window resolution.
- **theme_path** -- relative file path to theme or theme dictionary.
- **enable_live_theme_updates** -- Lets the theme update in-game after we edit the theme file

add_font_paths(*font_name: str*, *regular_path: str*, *bold_path: str | None = None*, *italic_path: str | None = None*, *bold_italic_path: str | None = None*)

Add file paths for custom fonts you want to use in the UI. For each font name you add you can specify font files for different styles. Fonts with designed styles tend to render a lot better than fonts that are forced to make use of pygame's bold and italic styling effects, so if you plan to use bold and italic text at small sizes - find fonts with these styles available as separate files.

The font name you specify here can be used to choose the font in the blocks of HTML-subset formatted text, available in some of the UI elements like the UITextBox.

It is recommended that you also preload any fonts you use at an appropriate moment in your project rather than letting the library dynamically load them when they are required. That is because dynamic loading of large font files can cause UI elements with a lot of font usage to appear rather slowly as they are waiting for the fonts they need to load.

Parameters

- **font_name** -- The name of the font that will be used to reference it elsewhere in the GUI.
- **regular_path** -- The path of the font file for this font with no styles applied.
- **bold_path** -- The path of the font file for this font with just bold style applied.
- **italic_path** -- The path of the font file for this font with just italic style applied.
- **bold_italic_path** -- The path of the font file for this font with bold & italic style applied.

calculate_scaled_mouse_position(*position: Tuple[int, int]*) → Tuple[int, int]

Scaling an input mouse position by a scale factor.

clear_and_reset()

Clear all existing windows and the root container, which should get rid of all created UI elements. We then recreate the UIWindowStack and the root container.

create_new_theme(*theme_path: str | PathLike | StringIO | PackageResource | dict | None = None*) → *UIAppearanceTheme*

Create a new theme using self information. :param theme_path: relative file path to theme or theme dictionary.

create_tool_tip(*text: str*, *position: Tuple[int, int]*, *hover_distance: Tuple[int, int]*, *parent_element: IUIElementInterface*, *object_id: ObjectID*, ***, *wrap_width: int | None = None*, *text_kwargs: Dict[str, str] | None = None*) → *IUITooltipInterface*

Creates a tool tip ands returns it. Have hidden this away in the manager, so we can call it from other UI elements and create tool tips without creating cyclical import problems.

Parameters

- **text** -- The tool tips text, can utilise the HTML subset used in all UITextBoxes.
- **position** -- The screen position to create the tool tip for.
- **hover_distance** -- The distance we should hover away from our target position.
- **parent_element** -- The UIElement that spawned this tool tip.
- **object_id** -- the object_id of the tooltip.
- **wrap_width** -- an optional width for the tool tip, will overwrite any value from the theme file.
- **text_kwargs** -- a dictionary of variable arguments to pass to the translated string useful when you have multiple translations that need variables inserted in the middle.

Returns
A tool tip placed somewhere on the screen.

`draw_ui`(*window_surface: Surface*)

Draws all the UI elements on the screen. Generally you want this to be after the rest of your game sprites have been drawn.

If you want to do something particularly unusual with drawing you may have to write your own UI manager.

Parameters
window_surface -- The screen or window surface on which we are going to draw all of our UI Elements.

`get_double_click_time`() → float

Returns time between clicks that counts as a double click.

Returns
A float, time measured in seconds.

`get_focus_set`()

Gets the focused set.

Returns
The set of elements that currently have interactive focus. If None, nothing is currently focused.

`get_hovering_any_element`() → bool

True if any UI element (other than the root container) is hovered by the mouse.

Combined with 'get_focus_set()' and the return value from process_events(), it should make it easier to switch input events between the UI and other parts of an application.

`get_locale`()

Get the locale language code being used in the UIManager

Returns
A two-letter ISO 639-1 code for the current locale.

`get_mouse_position`() → Tuple[int, int]

Wrapping pygame mouse position, so we can mess with it.

`get_root_container`() → *IUIContainerInterface*

Returns the 'root' container. The one all UI elements are placed in by default if they are not placed anywhere else, fills the whole OS/pygame window.

Returns
A container.

`get_shadow`(*size: Tuple[int, int]*, *shadow_width: int = 2*, *shape: str = 'rectangle'*, *corner_radius: List[int] | None = None*) → Surface

Returns a 'shadow' surface scaled to the requested size.

Parameters

- **size** -- The size of the object we are shadowing + it's shadow.
- **shadow_width** -- The width of the shadowed edge.
- **shape** -- The shape of the requested shadow.
- **corner_radius** -- The radius of the shadow corners if this is a rectangular shadow.

Returns
A shadow as a pygame Surface.

`get_sprite_group`() → *LayeredGUIGroup*

Gets the sprite group used by the entire UI to keep it in the correct order for drawing and processing input.

Returns
The UI's sprite group.

`get_theme`() → *IUIAppearanceThemeInterface*

Gets the theme so the data in it can be accessed.

Returns
The theme data used by this UIManager

`get_universal_empty_surface`() → Surface

Sometimes we want to hide sprites or just have sprites with no visual component, when we do we can just use this empty surface to save having lots of empty surfaces all over memory.

Returns
An empty and therefore invisible pygame.surface.Surface

`get_window_stack`() → *IUIWindowStackInterface*

The UIWindowStack organises any windows in the UI Manager so that they are correctly sorted and move windows we interact with to the top of the stack.

Returns
The stack of windows.

`preload_fonts`(*font_list: List[Dict[str, str | int | float]]*)

It's a good idea to preload the exact fonts your program uses during the loading phase of your program. By default, the pygame_gui library will still work, but will spit out reminder warnings when you haven't done this. Loading fonts on the fly will slow down the apparent responsiveness when creating UI elements that use a lot of different fonts.

To preload custom fonts, or to use custom fonts at all (i.e. ones that aren't the default 'noto_sans' font) you must first add the paths to the files for those fonts, then load the specific fonts with a list of font descriptions in a dictionary form like so:

```
{'name': 'noto_sans', 'point_size': 12, 'style': 'bold_italic',
'antialiased': 1}
```

You can specify size either in pygame.Font point sizes with 'point_size', or in HTML style sizes with 'html_size'. Style options are:

- `'regular'`
- `'italic'`
- `'bold'`
- `'bold_italic'`

The name parameter here must match the one you used when you added the file paths.

Parameters
font_list -- A list of font descriptions in dictionary format as described above.

`print_layer_debug`()

Print some formatted information on the current state of the UI Layers.

Handy for debugging layer problems.

print_unused_fonts()

Helps you identify which preloaded fonts you are actually still using in your project after you've fiddled around with the text a lot by printing out a list of fonts that have not been used yet at the time this function is called.

Of course if you don't run the code path in which a particular font is used before calling this function then it won't be of much use, so take its results under advisement rather than as gospel.

process_events(*event: Event*)

This is the top level method through which all input to UI elements is processed and reacted to.

One of the key things it controls is the currently 'focused' element of which there can be only one at a time. It also manages 'consumed events' these events will not be passed on to elements below them in the GUI hierarchy and helps us stop buttons underneath windows from receiving input.

Parameters
event -- pygame.event.Event - the event to process.

Returns
A boolean indicating whether the event was consumed.

set_active_cursor(*cursor: Cursor*)

This is for users of the library to set the currently active cursor, it will be currently only be overridden by the resizing cursors.

The expected input is a pygame.cursors.Cursor:

```
manager.set_active_cursor(pygame.cursors.Cursor(pygame.SYSTEM_CURSOR_ARROW))
```

set_focus_set(*focus: IUIElementInterface | Set[IUIElementInterface] | None*)

Set a set of element as the focused set.

Parameters
focus -- The set of element to focus on.

set_locale(*locale: str*)

Set a locale language code to use in the UIManager

Parameters
locale -- A two letter ISO 639-1 code for a supported language.

TODO: Make this raise an exception for an unsupported language?

set_text_hovered(*hovering_text: bool*)

Set to true when hovering an area containing selectable text.

Currently, switches the cursor to the I-Beam cursor.

Parameters
hovering_text_input -- set to True to toggle the I-Beam cursor

set_ui_theme(*theme: IUIAppearanceThemeInterface*, *update_all_sprites: bool = False*)

Set ui theme.

Parameters

- **theme** -- The theme to set.
- **update_all_sprites** --

set_visual_debug_mode(*is_active: bool*)

Loops through all our UIElements to turn visual debug mode on or off. Also calls print_layer_debug()

Parameters
is_active -- True to activate visual debug and False to turn it off.

set_window_resolution(*window_resolution: Tuple[int, int]*)

Sets the window resolution.

Parameters
window_resolution -- the resolution to set.

update(*time_delta: float*)

From here all our UI elements are updated and which element is currently 'hovered' is checked; which means the mouse pointer is overlapping them. This is managed centrally, so we aren't ever overlapping two elements at once.

It also updates the shape cache to continue storing already created elements shapes in the long term cache, in case we need them later.

Finally, if live theme updates are enabled, it checks to see if the theme file has been modified and triggers all the UI elements to rebuild if it has.

Parameters
time_delta -- The time passed since the last call to update, in seconds.

class pygame_gui.**UITextEffectType**(*name*)

Bases: `object`

A Type for Text effect constants, so we can mess with them later if needs be

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X