Simple Widgets & GUI

The `pyglet.gui` module provides a selection of widgets that can be used to add user interface elements to your game or application. The selection is limited, but should cover the most common use cases. For example: the configuration screen in a game, or a set of toolbar buttons for a visualization program.

Widgets are internally constructed from other high level pyglet objects, and are therefore fairly simple in design. They should blend in well with any pyglet project. For example, Widgets are event handlers that receive keyboard and mouse events from the Window. They can then in turn dispatch their own custom events, because they subclass `EventDispatcher`. Widgets can take a Batch and Group, similar to other pyglet objects, to allow for Batched rendering.

Before reading further, it is important to understand how event handling and dispatching work in pyglet. If you have not yet done so, it is recommended that you first read through the Event dispatching & handling section of the documentation. Widgets are by nature very tightly associated with input events, so this is necessary to fully grasp their usage.

Example code can be found in `examples/gui/widgets.py` in the pyglet source repository.

Creating a Widget

Included Widgets are `PushButton`, `ToggleButton`, `Slider`, and `TextEntry`. They each have different arguments, which will be shown in the API documentation. For our example, we will create a 'PushButton' widget, which requires you to provide at least two images. These two images will visually represent the "pressed" and "unpressed" states of the button. This widget can also take an optional image for 'hover', but we'll skip that for now:

```
pressed_img = pyglet.resource.image("button_pressed.png")
unpressed_img = pyglet.resource.image("button_unpressed.png")

pushbutton = pyglet.gui.PushButton(
    x=100, y=300, pressed=pressed_img,
    unpressed=unpressed_img, batch=batch,
)
```

;

We now have a PushButton widget, but it won't yet do anything. It will be drawn on screen, however, if included as part of a `Batch` as shown above. In order to get the widget to react to the mouse, we need to set it to handle events dispatched by the Window:

```
my_window.push_handlers(pushbutton)
```

The widget should now change appearance when you click on it. It will switch between the provided images (pressed and unpressed states). You can try adding the 'hover' image as well, for more visual feedback.

Now that our widget is receiving events, we can now take of the events that are produced *by* the widget. In this case, the PushButton widget dispatches two events: 'on_press' and 'on_release'. To wire these up, we simply set handlers for them:

```
def my_on_press_handler(widget):
    print("Button Pressed!")

def my_on_release_handler(widget):
    print("Button Released...")

pushbutton.set_handler('on_press', my_on_press_handler)
pushbutton.set_handler('on_release', my_on_release_handler)
```

If we try this code, we should see messages printed to the console whenever we press and release the button. You now have a way to make the widgets interact with code in your project.

Other widgets are used in a similar way, but have different arguments depending on their nature. Have a look at the API documentation for each one, and also see the example in the source repository.

Frame objects

pyglet also provides an optional `Frame` object. If you only need a few widgets at a time, then you can ignore this object. However, if you several dozen widgets at once, you might find that it's wasteful to have every widget receiving the Window events at the same time. This is where the Frame can be useful. Essentially, it acts as a "middle-man" between the Window events and the Widgets. The Frame implements a simple 2D spatial hash, which only passes on Window events to those Widgets that are actually near the mouse pointer. This works well for Widgets that generally care about mouse clicks and keyboard keys, but has some limitatic and-drop type events (more on that later).

Without a Frame, the general widget usage is:

1. Make one or more Widget instances.
2. Push the Wigets as event handlers on your Window.
3. All Widgets receives all Window events.

If a Frame is introduced, the following occurs:

1. Make a single Frame instance.
2. Set the Frame as a handler for Window events.
3. Make one or more Widget instances.
4. Add your widget instances to the Frame.
5. Only Widgets near the mouse pointer will recieve Window events.

This works quite well for most cases, but has some limitations. When using the TextEntry widget, for instance, the widget may become unresponsive if you use click-and-drag to select text, but your mouse pointer moves far enough away from the widget. For this reason, Frames may not be suitable.

The Frame concept may be developed further in a future release, but for now it serves a limited but useful purpose.

Custom widgets

For users who are interested in creating their own custom Widgets, the `WidgetBase` base class is available for subclassing. This base class has most of the relevant Window events pre-defined, and is ready to be pushed as a handler. Custom subclasses can then override whichever mouse or keyboard events they need, depending on the application. Some additional helper properties are also provided.

It is recommended look through the pyglet source code to have a better understanding of how this looks in practice. Because Widgets are made up of other high-level pyglet objects, you might find that it's not terribly complex. The PushButton Widget, for example, is less than 100 lines of code. This may be a good starting point to design a custom Widget for your specific use case.

This section may be expanded further in a future release.