Pyglet-gui Documentation

Release 0.1

Jorge C. Leitão

Sep 20, 2017

Contents

1 Pyglet-gui at a glance **3**
- 1.1 Hello world . . . 3
- 1.2 A Button . . . 4
- 1.3 Modifying the button . . . 5
- 1.4 This is just part of the whole . . . 5

2 Overview of the API **7**
- 2.1 Viewers . . . 7
- 2.2 Theme and Graphics . . . 8
- 2.3 Controllers . . . 8
- 2.4 Examples . . . 8
- 2.5 Existing user interfaces . . . 9

3 Viewers **11**
- 3.1 Managed . . . 11
- 3.2 Rectangle . . . 11
- 3.3 Viewer . . . 12

4 Containers **15**
- 4.1 Container . . . 15
- 4.2 Other containers . . . 15

5 Controllers **17**
- 5.1 Controller . . . 17
- 5.2 Two state controller . . . 18
- 5.3 Continuous state controller . . . 18
- 5.4 Options and selectors . . . 18

6 Managers **21**
- 6.1 Viewer Manager . . . 21
- 6.2 Controller Manager . . . 23
- 6.3 Manager . . . 24

7 Theme **25**
- 7.1 Graphic elements . . . 25
- 7.2 Templates . . . 26
- 7.3 Parser . . . 26

8 Button 29

Contents:

CHAPTER 1

Pyglet-gui at a glance

Pyglet gui was designed to make Graphical User Interfaces (GUI) in Pyglet. Here's an overview of how you can write a GUI in Pyglet-gui.

First, a minimal Pyglet:

```
import pyglet

window = pyglet.window.Window(640, 480, resizable=True, vsync=True)
batch = pyglet.graphics.Batch()

@window.event
def on_draw():
    window.clear()
    batch.draw()
```

Hello world

In Pyglet-gui, a GUI always need a `Theme`. Let's build one:

```
from pyglet_gui.theme import Theme

theme = Theme({"font": "Lucida Grande",
               "font_size": 12,
               "text_color": [255, 0, 0, 255]}, resources_path='')
```

Don't worry about the *resources_path*='' for now. With this theme, we can now create a simple `Label`:

```
from pyglet_gui.gui import Label

label = Label('Hello world')
```

Finally, we create a Manager to initialize a GUI and we run Pyglet app:

```
from pyglet_gui.manager import Manager

Manager(label, window=window, theme=theme, batch=batch)

pyglet.app.run()
```

A Button

Let's say we now want a Button. Using the same Pyglet's setup, we create a more complex Theme:

```
from pyglet_gui.theme import Theme

theme = Theme({"font": "Lucida Grande",
               "font_size": 12,
               "text_color": [255, 255, 255, 255],
               "gui_color": [255, 0, 0, 255],
               "button": {
                   "down": {
                       "image": {
                           "source": "button-down.png",
                           "frame": [8, 6, 2, 2],
                           "padding": [18, 18, 8, 6]
                       },
                       "text_color": [0, 0, 0, 255]
                   },
                   "up": {
                       "image": {
                           "source": "button.png",
                           "frame": [6, 5, 6, 3],
                           "padding": [18, 18, 8, 6]
                       }
                   }
               }
              }, resources_path='theme/')
```

This is assigning textures for the up and down state of the button.

Compared to the previous example, we added "gui_color" (color of non-text elements) and "button" to the root, and resources_path='theme/'. This assumes the image "button.png" and "button-down.png" are in the directory "theme/" (use Pyglet-gui ones for now).

Again:

```
from pyglet_gui.buttons import Button

# just to print something to the console, is optional.
def callback(is_pressed):
    print('Button was pressed to state', is_pressed)

button = Button('Hello world', on_press=callback)
```

and we run:

```
from pyglet_gui.manager import Manager

Manager(button, window=window, theme=theme, batch=batch)
```

```
pyglet.app.run()
```

This is the basic idea of Pyglet-gui: you set up a Theme and create the GUI.

The default path of the Pyglet-gui Button is “button”->”up” and “button”->”down”, which, in Pyglet-gui is, represented by lists: [”button”, “up”] and [”button”, “down”].

Modifying the button

Lets now assume we don’t want the paths [”button”, “up”] and [”button”, “down”], but we want the path [”my_path”, “up”] and [”my_path”, “down”]. We do:

```
from pyglet_gui.buttons import Button

class MyButton(Button):
    def get_path(self):
        path = ['my_path']
        if self.is_pressed:
            path.append('down')
        else:
            path.append('up')
        return path

button = MyButton('Hello world', on_press=callback)
```

Pyglet-gui is designed to be reusable. All elements in Pyglet-gui are designed to be subclassed to fulfill the developer’s need.

This is just part of the whole

This was a minimal overview of how you use Pyglet-gui, but Pyglet-gui is more. It provides a consistent API to define custom Themes, custom graphics, and, most importantly, user interfaces.

The next logical step is to have an overview of what Pyglet-gui allows you to do. Thanks for your interest!

CHAPTER 2

Overview of the API

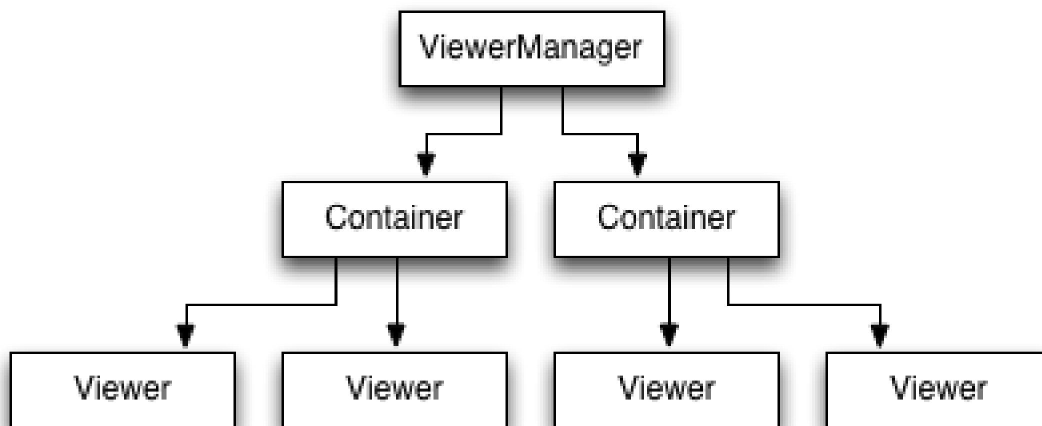
This document gives an overview of how Pyglet-gui works and what you can do with it.

Pyglet-gui uses *Viewers* for defining appearance and *Controllers* for defining behaviour. For instance, a *Button* is a subclass of a Viewer (for draw) and of a Controller (for behaviour).

Viewers

A Viewer is characterized by a rectangular bounding box that implements abstract methods to draw `Graphical Elements` such as textures, inside it.

In Pyglet-gui, a GUI organizes viewers in a tree: every viewer has a parent *Container* (a subclass of Viewer with children viewers) and the root of the tree is a *ViewerManager*, a special container without parent. This is small variation of the composite pattern.



This structure is essentially used to minimize the number of operations in the drawing Batch; Pyglet-gui provides two orthogonal ways to operate on the batch: the top-down and bottom-up:

- Top-down: when a container wants to reload itself in the batch (e.g. in the initialization of the *Manager*).
- Bottom-up: when a single *Viewer* wants to reload itself (e.g. when a *Controller* changed a viewer's state).

Pyglet-gui abstracts most of these concepts by a simple interface. The procedure can be decomposed in three steps, as exemplified in *Button* source code:

```
def change_state(self):
    self._is_pressed = not self._is_pressed
    self.reload()
    self.reset_size()
```

1. the state of the Viewer changes, and that requires a new appearance;
2. *reload()* the graphics of the Viewer;
3. *reset_size()* reset size of the viewer bounding box.

If the Viewer changed size when it became pressed, the method *reset_size()* is propagated to the parent container and in the tree up to the container that didn't changed size, which means that a relayout of the GUI is only made to a certain level in the tree, minimizing Batch operations. The complete references of this API can be found in *Viewer*.

Theme and Graphics

Pyglet-gui has a graphics API for handling vertex lists and vertex attributes: The developer defines a `Theme` from a dictionary, and viewers select the part of the theme they need using a path computed from the viewer's current state, *get_path()*.

This `Theme` is constructed out of a nested dictionary by having *Parsers* interpreting the dictionary's content and populating the Theme with *Templates*.

These templates are able to generate `Graphical Elements` that are used by *Viewers* to compose their appearance.

Controllers

A Controller represents something that can have behavior, such as something triggered by Pyglet events.

Pyglet-gui uses a *ControllerManager* for handling all window events in the GUI, and the manager uses these events to call the correct *Controllers'* handlers.

A handler in a controller is just a method "on_*": the ControllerManager only handles specific Pyglet events and uses `hasattr()` to check which controllers receive those events.

Examples

In the directory "examples" you can find examples of how to instantiate GUIs and how to use the Pyglet-gui to create elements with custom functionality.

In fact, all Pyglet-gui user interfaces are examples, since they are just subclasses of *Controller*, *Viewer*, or both, that implement custom methods:

- *get_path()*: used to select the path on the `Theme`;
- *load_graphics()* and *unload_graphics()*: used to load and unload `Graphical Elements`;

- `layout()`: used to position the `Graphical Elements` in the correct place;
- `compute_size()`: used to compute the size of the Viewer from the graphics it contains;
- *on_**: used to handle events.

Existing user interfaces

Below is a list of the existing elements in Pyglet-gui. Elements that are not links are not documented yet and most probably are not yet covered by a Test Case.

Viewers:

- Graphics: a viewer with a graphic element from the theme.
- Spacer: an empty viewer for filling space in containers.
- Label: a viewer that holds text.
- Document: a viewer that holds Pyglet documents (optionally with a scrollbar).

Controllers:

- `TwoStateController`: a controller with two states.
- `ContinuousStateController`: a controller with a float value state.
- Slider: a `ContinuousStateController` with continuous or discrete states and 3 graphic elements: a bar, a knob and markers.

Containers:

- Vertical: widgets inside are arranged vertically.
- Horizontal: widgets inside are arranged horizontally.
- Grid: widgets inside are arranged in a grid (you provide a matrix of them).
- Frame: a wrapper that adds a graphical frame around a viewer.
- Scrollable: a wrapper with scrollable content.

End-user controllers:

- `Button`: a On/Off button with a label and graphics placed on top off each other.
- OneTimeButton: a Button which turns off when is released.
- Checkbox: a Button where the label is placed next to the graphics (and graphics is a checkbox-like button).
- FocusButton: a Button that can have focus and is selectable with TAB.
- HorizontalSlider: an concrete implementation of a Slider, in horizontal position.
- TextInput: a box for writing text.

CHAPTER 3

Viewers

This section describes how the viewer API works and how you can use it.

Managed

class `pyglet_gui.core.`**`Managed`**

A managed is an abstract class from where all GUI elements derive from. Like the name suggests, it is managed by a *`Manager`*. It is attached to a manager using

`set_manager``()`

Sets the manager of this class.

This class exposes important attributes of the manager such as the theme and (manager's) batch. It represents the idea that any controller or viewer in Pyglet-gui are managed by a *`Manager`*.

`get_batch``()`

Returns a dictionary of the form {'batch': batch, 'group': group} where *group* is a string from the available drawing *`groups`* of the manager.

`theme`

A read-only property that returns its manager's theme.

Rectangle

class `pyglet_gui.core.`**`Rectangle`**

A geometric rectangle represented by x, y, width and height. It is used for different operations in Pyglet-gui.

`x, y`

The position of the rectangle

`width, height`

The size of the rectangle

is_inside()

Parameters

- **x** –
- **y** –

Returns True if point (x,y) lies inside the rectangle

set_position()

Parameters

- **x** –
- **y** –

Setter for (x, y).

Viewer

class pyglet_gui.core.**Viewer**

A viewer, subclass of *Managed* and *Rectangle*, is generic way of displaying Pyglet-gui elements in a window.

Viewers are organized in a tree structure where the manager is always the root, the nodes are *Containers*, and viewers are leafs.

Viewers can have graphical elements that have to be defined by subclasses and are loaded by *load_graphics()*.

In Pyglet-gui, the viewer's appearance is defined by the path it chooses from the Theme, defined in *get_path()*.

get_path()

Returns the viewer's path on the theme.

get_path() can return a different path depending on the viewer's state, for example, in pyglet-gui's Button:

```
def get_path(self):
    path = ['button']
    if self.is_pressed():
        path.append('down')
    else:
        path.append('up')
    return path
```

leads to a different appearance depending on whether the button is pressed or not.

To draw elements, a viewer assigns graphical elements to its manager's batch using *get_batch()* This is done by calling generate() for each of its graphics in the method

load_graphics()

Method used to generate() graphics this viewer owns. It normally calls *get_path()* to retrieve the specific subset of theme it needs:

```
theme = self.theme[self.get_path()]
```

followed by calls of the form:

```
self._button = theme['image'].generate(color=theme['gui_color'], **self.get_
↪batch('background'))
# _button is now a loaded graphic element.
```

Analogously, a viewer has to define the method *unload_graphics()* to deconstruct the generated graphics from load.

unload_graphics()

Method used to unload graphics loaded in *load_graphics()*.

Example:

```
# _button is a loaded graphic element.
self._button.unload()
```

Most of the times, load and unload are called consecutively: when the viewer wants to change its appearance, e.g. because it changed its state, it has to unload itself to remove the graphics from the batch, and load them again using the new path. Pyglet-gui provides the method *reload()* for that:

reload()

Calls unload followed by load. Used in the bottom-up drawing scheme when the element change its state (e.g. by an event).

This is used to update the graphics whenever the Viewer changed state.

One important feature of a viewer is that it is not supposed to overlap with other viewers from the same GUI. This means that is its parent who decides its position. The method *compute_size()* returns the computed size of the viewer from the Graphics it has.

compute_size()

Computes the size of the viewer and returns the tuple (width, height). Implementation is made by subclasses.

The size must include all graphics and possible children the viewer has; this is the bounding box of the viewer to avoid overlaps.

The default implementation returns (self.width, self.height).

When the parent has the size of all its children, it sets the position of the Viewer, using *set_position()*:

set_position()

Parameters

- **x** –
- **y** –

A setter for the position of the viewer. Calls *layout()* after to ensure the graphics are also set.

layout()

Places graphical elements in the correct positions in relation to the viewer's position.

Default implementations does nothing.

What defines the functionality of the viewer is the method *reset_size()*, which is worth transliterating:

```
def reset_size(self, reset_parent=True):
    width, height = self.compute_size()

    # if out size changes
    if self.width != width or self.height != height:
        self.width, self.height = width, height
```

```
        # This will eventually call our layout
        if reset_parent:
            self.parent.reset_size(reset_parent)
    # else, the parent is never affected and thus we layout.
    else:
        self.layout()
```

reset_size()

Parameters **reset_parent** – A boolean, see below.

The case *reset_parent* = *False* updates the viewer size and `layout()` if the size changed. This call is what we call a top-down draw: it is called when it was the parent's initiative to reset_size of the viewer.

The *reset_parent* = *True* does the same but, if the size changes, it also calls the parent's reset_size. This call is the bottom-up draw: the child decided to trigger a reset_size.

In the button-up, the parent will re-calculate its own size, and calls reset_size of all children, with *reset_parent* = *False*. This ensures that all its children are affected by the size change of one of them.

This call can be further propagated to the parent's parent in order to accommodate the size changes of all elements.

In situations where an event was triggered (e.g. by a `Controller`), the bottom-up is the correct way, thus *reset_size()* should be called after `reload()`. For example, Pyglet-gui's `pyglet_gui.button.Button` uses:

```
def change_state(self):
    self._is_pressed = not self._is_pressed
    self.reload()
    self.reset_size()
```

Finally, the viewer implements a `delete()`, used for deleting the element

delete()

Used to delete the viewer: calls `unload_graphics()` and undo initialization.

CHAPTER 4

Containers

Container

class `pyglet_gui.containers.`**`Container`**

A *`Viewer`* that contain other viewers. This is an abstract and base class of all containers in Pyglet-gui and is used to group viewers and position them in specific ways.

In the *`Viewer`* API, a container is a node in the tree of viewers.

While viewers only have to load graphics, a container has to load both its graphics and its content. Thus, the container provides two aditional methods:

`load_content``()`
Loads all viewers in the container

`unload_content``()`
Unloads all viewers in the container

Both these methods are already correctly called during a *`reload()`*.

The getters and setters of content are:

`content`
A read-only property returning the content of the container.

`add``(`*viewer*`)`
Adds the viewer to the container's content.

`remove``(`*viewer*`)`
Removes the viewer from the container's content.

Other containers

class `pyglet_gui.containers.`**`Wrapper`**

A wrapper is a container that contains one and only one Viewer. It follows the decorator pattern.

It does not have any graphical appearance and is used by Pyglet-gui for creating more interesting elements such as the *ViewerManager*.

CHAPTER 5

Controllers

A viewer, by itself, cannot be interacted by events; it is a static element. On the other hand, a *Controller* is a dynamic element that does not have a geometric representation and is not able to draw itself on the screen.

To provide functionality to a viewer, or to provide drawing features to a controller, Pyglet-gui mixes both.

This section introduces the API for controllers.

Controller

class pyglet_gui.core.**Controller**

A controller is an abstract class that represents something that can be controlled.

The main functionality of a controller is to attach itself to a list of controllers of the *ControllerManager*.

set_manager()

Sets its manager and calls:

```
manager.add_controller(self)
```

When mixing a Controller with a Viewer, the controller has to be the first parent-class or you have to write a custom *set_manager()*.

This way, the *ControllerManager* can dispatch calls when it handles events from the Pyglet's window. To a controller receive events (e.g. "on_press"), it has to have the method implemented (i.e. the manager uses `hasattr()` to decide if it sends the event to the controller or not). The signature of the method must be the same same as of Pyglet (or, if you want, the one *ControllerManager* calls).

To receive mouse events, the controller has to define `hit_test(x, y)()`, which returns True if the point (x, y) is inside the controller and False otherwise.

Two state controller

The simplest example of a controller is one that flips between two states. Pyglet-gui provides a simple abstraction of such behavior in the *TwoStateController*.

class pyglet_gui.controllers.**TwoStateController**

A Controller with two possible values characterized by the read-only property *is_pressed*. This controller accepts the following arguments:

Parameters

- **on_press** – An optional callback function of one boolean argument that is called when the controller changes state.
- **is_pressed** – An optional boolean for deciding the state on initialization.

is_pressed

True if in one state, False in the other.

This controller has the method

change_state()

Flips the state of the controller and calls on_press if it is defined.

Continuous state controller

Another example of a useful controller is a controller with a continuous set of values within an interval. Pyglet-gui provides a simple abstraction of such behavior in the *ContinuousStateController*.

class pyglet_gui.controllers.**ContinuousStateController**

A Controller with a state in a continuous interval [min_value, max_value] characterized by the read-only property *value*.

Parameters

- **value** – The initial value. Default to 0
- **min_value** – Default to 0.0
- **max_value** – Default to 1.0
- **on_set** – An optional callback function of one float argument that is called when the controller changes value.

value

The value of the controller. A read-only property.

This controller has the method

set_value()

The setter for the value. Calls on_set if it is defined. The value must belong to [min_value, max_value].

Options and selectors

One useful GUI less trivial example of a controller is selector: a menu with a set of options, and the user can choose one and only one. Pyglet-gui provides an abstraction of such behavior in the *Option* and *Selector*.

class `pyglet_gui.controllers.`**`Option`**

A `Controller` with a name and a parent selector. The name is used as an id in the parent selector. This controller is initialized by a name and a parent:

Parameters

- **`name`** – Mandatory string
- **`parent`** – Mandatory *`Selector`*.

and has one method

`select``()`

Makes him the current selection of the parent.

class `pyglet_gui.controllers.`**`Selector`**

An abstract class with a set of options labeled by a string. The arguments are

Parameters

- **`options`** – Mandatory list of strings identifying the options.
- **`labels`** – Optional list of strings with the same length of options labeling the options.
- **`on_select`** – Optional callback function that will receive one argument, the selected option name.
- **`selected`** – Optional string (belonging to "options" setting a initially-selected item.

This class has two methods:

`select``()`

Selects the option name and, if defined, calls on_select.

`deselect``()`

Deselects the current selected option, if any is selected.

CHAPTER 6

Managers

In Pyglet-gui, each independent GUI is a *Manager*, a subclass of both *ViewerManager* and *ControllerManager*.

This section provides the relevant references for understanding how *Manager* works and how you can use it.

This section is the most complex of this documentation because it glues different APIs together. The references of the classes are themselves divided in APIs, so it is hopefully easier to understand.

Viewer Manager

ViewerManagerGroup

Each *Manager* is independent of each other, but they are drawn on the same window, so, they need different vertex groups to know which one is drawn on top. A `ViewerManagerGroup` is defined for that:

class `pyglet_gui.manager.`**`ViewerManagerGroup`**

A Pyglet's ordered group, i.e. a drawing group that preserves ordering with a unique ordering on instances of ViewerManagerGroup.

This group uses its own order, *own_order*, to distinguish itself from other Pyglet's Ordered groups.

`own_order`

The same value as `order`, used for comparisons between ViewerManagerGroup.

This group defines *__eq__*, *__lt__* and *__hash__* that compare against ViewerManagerGroup using *own_order* and against other ordered groups using `order`.

The different ViewerManagerGroup don't know each other, but always know if they are on top of all.

`is_on_top`()

Returns true if the particular instance is on top amongst all instances of ViewerManagerGroup.

To set this group to be the top group, use *pop_to_top()*:

pop_to_top()

Sets *own_order* to the highest value amongst all instances of ViewerManagerGroup, ensuring the instance becomes the top.

ViewerManager

class pyglet_gui.manager.**ViewerManager**

A manager of *Viewers*. A ViewerManager is a subclass of *pyglet_gui.containers.Wrapper* that exposes important features of Pyglet-gui.

Because it is a container, it is part of the tree structure used by Pyglet-gui to draw viewers. However, this container is special in the sense that it does not have a parent, and thus it only *pyglet_gui.core.Viewer.reset_size()* with reset_parent=False, i.e. it only uses the top-down drawing.

One consequence is that because no one sets its position, it sets its own position, from a position computed from *get_position()*.

Because it is the root of the tree, it exposes attributes required for drawing to its viewers. They are the pyglet_gui.theme.theme.Theme, the Batch and batch groups.

theme

The pyglet_gui.theme.theme.Theme of this manager. A read-only property defined in the initialization.

One theme can be shared among different ViewerManagers.

batch

The Batch of the manager. A read-only property defined on the initialization.

If no batch is provided in initialization, this Manager defines its own batch and exposes a draw() method.

A Pyglet Batch can be shared among ViewerManagers and is exposed by each viewer by the method *pyglet_gui.core.Managed.get_batch()*.

Because Pyglet-gui Theme API uses groups for drawing, the ViewerManager is responsible for defining such groups to its viewers.

The first group required is for the ViewerManager itself, such that different ViewerManagers can be drawn in the same window. This is implemented in the *root_group*:

root_group

A *ViewerManagerGroup* used by ViewerManagers to decide which manager is on top of each other (on drawing). It is exposed as a read-only property.

Because there can be several managers on the same window, the viewer implements the method *pop_to_top()*:

pop_to_top()

Calls *ViewerManagerGroup.pop_to_top()*.

For drawing viewers, this manager has 4 sub-groups exposed by the attribute *group*:

group

A dictionary of 4 key-strings: 'panel', 'background', 'foreground', 'highlight' mapping to 4 pyglet.graphics.OrderedGroup with orders 10, 20, 30 and 40 respectively.

When a graphic element is generated by the Viewer, the viewer has to decide which group to use to that element. This property is exposed in each viewer by the method *pyglet_gui.core.Managed.get_batch()*.

window
A Pyglet window where the ViewerManager lives, exposed as a property.

The manager uses Pyglet's window to know where it has to be positioned, and to assign itself as an handler.

This property is writable to assign another window to the manager.

get_position()
Computes and returns its position (x, y) on its window.

Used with *pyglet_gui.core.Viewer.set_position()* to set the position of this manager in the window.

Controller Manager

class pyglet_gui.manager.**ControllerManager**
A controller manager is the class responsible for managing Pyglet-gui *Controllers*.

It has a list of controllers assigned to him and is responsible for calling its handlers.

controllers
The list of controllers assigned to him. Exposed as a read-only property.

add_controller()
Appends the controller to *controllers*.

remove_controller()
Removes the controller from *controllers*.

This manager assumes the user is only interested in using one controller at the time. It tracks down the mouse position and tests when the mouse entered in a controller bounding box, saving that controller as the current "hovering" controller.

When the mouse is pressed, the "hovering" controller also becomes the "focus" controller. These are unique within a manager because *Containers* don't overlap viewers.

The class exposes two methods for this behaviour:

set_focus()
Sets the controller to be the focus of the manager. If controllers have the method, it calls on_lose_focus and on_gain_focus of the old focus and new focus respectively.

set_hover()
Sets the controller to be the hover of the manager. If controllers have the method, it calls on_lose_highlight and on_gain_highlight of the old hover and new hover respectively.

The focus controller is the only controller to receive keystrokes and other events.

In this manager, the keystroke TAB and SHIFT+TAB are handled to navigate (to the front and to the back) in the list of controllers, to give focus to them. This is useful for keyboard driven GUIs.

This manager has two other special controllers, the "wheel target" and "wheel hint" (in case wheel target don't handle the event), used to handle mouse wheel events. This is useful for allowing scrollbars to receive wheel events without requiring the user to click on them to "focus it".

set_wheel_target()
Sets the wheel target to be the controller. The controller has to have the method on_mouse_scroll.

set_wheel_hint()
Sets the wheel hint to be the controller. The controller has to have the method on_mouse_scroll.

Manager

class `pyglet_gui.manager.`**`Manager`**

The manager is the Pyglet-gui main element for initializing a new GUI in Pyglet-gui. It is a subclass of both *`ViewerManager`* and *`ControllerManager`* which overrides some of the on_* methods to give some functionality to the *`ViewerManager`*.

Parameters

- **`content`** – The content of this manager. An instance of *`Viewer`*.
- **`theme`** – The Theme of this manager. An instance of `Theme`.
- **`window`** – The window of this manager. An instance of Pyglet Window.
- **`batch`** – An optional Batch for this manager. If set, must be an instance of Pyglet Batch.
- **`group`** – An optional Group, parent of the group this manager uses. Must be a Pyglet Group
- **`is_movable`** – If False, this manager is not movable.
- **`anchor`** – A anchor option to position this manager in relation to the window. Default to ALIGN_CENTER.
- **`offset`** – The offset of this manager in relation to the anchor point.

Besides the implementation of ViewerManager and ControllerManager, the manager implements its own movability: it can be dragged if the parameter 'is_movable' is true.

CHAPTER 7

Theme

Pyglet-gui Theme API defines a systematic approach for mapping a set of resources (e.g. "image.png") and attributes (e.g. color, padding) to lists of vertices and vertex attributes.

The API works as follows:

- The user defines a set of attributes and sources of static resources in a JSON file;
- A set of *Parsers* translate that to *Templates*;
- A `theme.Theme`, a nested dictionary, holds these templates with a unique identifier by a path (e.g. ['button', 'up'])
- A `theme.Theme` is passed to the *pyglet_gui.manager.ViewerManager*,

and *Viewers* load concrete graphical elements, `GraphicElement` using the path.

This document explains how this API works in detail. It starts by explaining Graphic elements, goes to Templates, Parsers, Theme, and ends in the JSON file.

Graphic elements

class `elements.`**`GraphicElement`**

A graphical element is a subclass of *pyglet_gui.core.Rectangle* and an abstract class that represents something with a set of vertices and a set of rules to assign a set of attributes (e.g. color, texture coordinate) to those vertices.

A GraphicalElement is normally instantiated by a *templates.Template*. The initialization needs a batch and a group to assign its vertices to a group in the batch.

A graphical element provides three methods for accessing its size:

`get_content_region``()`

Returns the tuple (x, y, width, height) with its region.

`get_content_size``()`

Returns the tuple (width, height) with the size this element.

get_needed_size()
Returns the tuple (width, height) with the size required for this element.

After the element is initialized, its size and position can be updated using *update()*:

update(*x*, *y*, *width*, *height*)
Updates the position and size of the graphics, updating its vertex list in the Batch.

When it is no longer needed, it can be destroyed using *unload()*:

unload()
Removes the vertex list from the Batch.

Pyglet-gui provides two concrete implementations of a Graphical element:

class elements.**TextureGraphicElement**
A subclass of GraphicElement representing a rectangle of vertices with a texture.

class elements.**FrameTextureGraphicElement**
A subclass of GraphicElement representing 9 rectangles, as represented in the figure

Fig. 7.1: How the FrameTextureGraphicElement maps an image into a rectangle. Notice that if the rectangle changes size, each of the 9 rectangles will increase independently, and the image will be stretched on each one independently.

The *elements.GraphicElement.get_content_size()* is overridden to return the size of the inner rectangle.

Templates

For generating graphical elements, Pyglet-gui uses the concept of template.

class templates.**Template**
An abstract class that provides the method *generate()* to return a new instance of a elements.GraphicalElement (or subclass of).

A template is normally instantiated by a Parser, when the Theme is being loaded.

generate(*color*, *batch*, *group*)
Returns a new instance of a elements.GraphicalElement. It is an abstract method.

Pyglet-gui provides two concrete implementations of templates:

class templates.**TextureTemplate**
A *Template* that generates a *elements.TextureGraphicElement*.

class templates.**FrameTextureTemplate**
A *TextureTemplate* that generates a *FrameTextureGraphicElement*.

Parser

class parsers.**Parser**
A parser is a class responsible for parsing elements during the Theme loading. The Theme has a set of parsers and they read "string-keys" and interpret the values of those keys into a *Template*.

condition_fulfilled(*key*)
This abstract method receives a string and returns a boolean value when it is able to interpret that key. If two parsers accept the same key, the first in the list of parsers in the Theme is chosen.

parse_element(*element*)
This abstract method receives a dictionary and returns a *Template*, effectively interpreting the element.

class parsers.**TextureParser**
A concrete parser that accepts the key "image", and interprets it into a *TextureTemplate* or a *FrameTextureTemplate*.

CHAPTER 8

Button

Pyglet-gui ships a standard button and two variations of it. A Button is a mixing of:

- `TwoStateController`
- *`Viewer`*

because it is a controllable viewer with two states ("is pressed" and "is not pressed").

class `pyglet_gui.buttons.`**`Button`**

A `TwoStateController` and *`Viewer`* represented as a label and texture drawn on top of each other.

Parameters

- **`label`** – The string written in the button.
- **`is_pressed`** – True if the button starts pressed
- **`on_press`** – A callback function of one argument called when the button is pressed (optional).

Attributes:

`label`
The label of the button (a string).

Accepted events:

`on_mouse_press``()`
Switches the state of the button.

`[button, down], [button, up]:`
default path in the theme.

class `pyglet_gui.buttons.`**`OneTimeButton`**

A *`Button`* that changes back to its original state when the mouse is released.

Parameters

- **`label`** – The string written in the button.

- **on_release** – A callback function of one argument called when the button is released (optional).

Accepted events:

on_mouse_release()
Switches the state back and calls the callback if the mouse was released inside the button.

[button, down], [button, up]
default path in the theme.

class pyglet_gui.buttons.**Checkbox**

A button drawn as a checkbox icon with the label on the side.

Parameters

- **label** – A string written in the button graphics.
- **is_pressed** – True if the button starts pressed
- **on_press** – A callback function of one argument called when the button is pressed (optional).
- **align** – Whether the label is left or right of the checkbox.
- **padding** – The distance from the label to the checkbox.

['checkbox', 'checked'], ['checkbox', 'unchecked']
default path in the theme.

class pyglet_gui.buttons.**FocusButton**
A *Button* that is focusable and thus can be selected with TAB.

Parameters

- **label** – The string written in the button.
- **is_pressed** – True if the button starts pressed
- **on_press** – A callback function of one argument called when the button is pressed (optional).

Accepted events:

on_mouse_press()
Switches the state of the button.

on_key_press()
If the Button have focus and ENTER is pressed the state of the button is switched.

[button, down], [button, up]
default path in the theme.

Index

A

add() (pyglet_gui.containers.Container method), 15
add_controller() (pyglet_gui.manager.ControllerManager method), 23

B

batch (pyglet_gui.manager.ViewerManager attribute), 22
Button (class in pyglet_gui.buttons), 29

C

change_state() (pyglet_gui.controllers.TwoStateController method), 18
Checkbox (class in pyglet_gui.buttons), 30
compute_size() (pyglet_gui.core.Viewer method), 13
condition_fulfilled() (pyglet_gui.theme.parsers.Parser method), 26
Container (class in pyglet_gui.containers), 15
content (pyglet_gui.containers.Container attribute), 15
ContinuousStateController (class in pyglet_gui.controllers), 18
Controller (class in pyglet_gui.core), 17
ControllerManager (class in pyglet_gui.manager), 23
controllers (pyglet_gui.manager.ControllerManager attribute), 23

D

delete() (pyglet_gui.core.Viewer method), 14
deselect() (pyglet_gui.controllers.Selector method), 19

E

elements.FrameTextureGraphicElement (class in pyglet_gui.theme), 26
elements.GraphicElement (class in pyglet_gui.theme), 25
elements.TextureGraphicElement (class in pyglet_gui.theme), 26

F

FocusButton (class in pyglet_gui.buttons), 30

G

generate() (pyglet_gui.theme.templates.Template method), 26
get_batch() (pyglet_gui.core.Managed method), 11
get_content_region() (pyglet_gui.theme.elements.GraphicElement method), 25
get_content_size() (pyglet_gui.theme.elements.GraphicElement method), 25
get_needed_size() (pyglet_gui.theme.elements.GraphicElement method), 25
get_path() (pyglet_gui.core.Viewer method), 12
get_position() (pyglet_gui.manager.ViewerManager method), 23
group (pyglet_gui.manager.ViewerManager attribute), 22

I

is_inside() (pyglet_gui.core.Rectangle method), 11
is_on_top() (pyglet_gui.manager.ViewerManagerGroup method), 21
is_pressed (pyglet_gui.controllers.TwoStateController attribute), 18

L

label (pyglet_gui.buttons.Button attribute), 29
layout() (pyglet_gui.core.Viewer method), 13
load_content() (pyglet_gui.containers.Container method), 15
load_graphics() (pyglet_gui.core.Viewer method), 12

M

Managed (class in pyglet_gui.core), 11
Manager (class in pyglet_gui.manager), 24

O

on_key_press() (pyglet_gui.buttons.FocusButton method), 30
on_mouse_press() (pyglet_gui.buttons.Button method), 29

on_mouse_press() (pyglet_gui.buttons.FocusButton method), 30
on_mouse_release() (pyglet_gui.buttons.OneTimeButton method), 30
OneTimeButton (class in pyglet_gui.buttons), 29
Option (class in pyglet_gui.controllers), 18
own_order (pyglet_gui.manager.ViewerManagerGroup attribute), 21

P

parse_element() (pyglet_gui.theme.parsers.Parser method), 27
parsers.Parser (class in pyglet_gui.theme), 26
parsers.TextureParser (class in pyglet_gui.theme), 27
pop_to_top() (pyglet_gui.manager.ViewerManager method), 22
pop_to_top() (pyglet_gui.manager.ViewerManagerGroup method), 22

R

Rectangle (class in pyglet_gui.core), 11
reload() (pyglet_gui.core.Viewer method), 13
remove() (pyglet_gui.containers.Container method), 15
remove_controller() (pyglet_gui.manager.ControllerManager method), 23
reset_size() (pyglet_gui.core.Viewer method), 14
root_group (pyglet_gui.manager.ViewerManager attribute), 22

S

select() (pyglet_gui.controllers.Option method), 19
select() (pyglet_gui.controllers.Selector method), 19
Selector (class in pyglet_gui.controllers), 19
set_focus() (pyglet_gui.manager.ControllerManager method), 23
set_hover() (pyglet_gui.manager.ControllerManager method), 23
set_manager() (pyglet_gui.core.Controller method), 17
set_manager() (pyglet_gui.core.Managed method), 11
set_position() (pyglet_gui.core.Rectangle method), 12
set_position() (pyglet_gui.core.Viewer method), 13
set_value() (pyglet_gui.controllers.ContinuousStateController method), 18
set_wheel_hint() (pyglet_gui.manager.ControllerManager method), 23
set_wheel_target() (pyglet_gui.manager.ControllerManager method), 23

T

templates.FrameTextureTemplate (class in pyglet_gui.theme), 26
templates.Template (class in pyglet_gui.theme), 26
templates.TextureTemplate (class in pyglet_gui.theme), 26
theme (pyglet_gui.core.Managed attribute), 11
theme (pyglet_gui.manager.ViewerManager attribute), 22
TwoStateController (class in pyglet_gui.controllers), 18

U

unload() (pyglet_gui.theme.elements.GraphicElement method), 26
unload_content() (pyglet_gui.containers.Container method), 15
unload_graphics() (pyglet_gui.core.Viewer method), 13
update() (pyglet_gui.theme.elements.GraphicElement method), 26

V

value (pyglet_gui.controllers.ContinuousStateController attribute), 18
Viewer (class in pyglet_gui.core), 12
ViewerManager (class in pyglet_gui.manager), 22
ViewerManagerGroup (class in pyglet_gui.manager), 21

W

window (pyglet_gui.manager.ViewerManager attribute), 22
Wrapper (class in pyglet_gui.containers), 15