Using PyGUI

BigWorld Technology 2.1. Released 2012.

Software designed and built in Australia by BigWorld.

Level 2, Wentworth Park Grandstand, Wattle St
Glebe NSW 2037, Australia
www.bigworldtech.com

Table of Contents

Chapter 1. Introduction

1.1. Overview

PyGUI is an example high-level GUI framework that is built upon the lower level BigWorld GUI library. It is designed to complement the BigWorld GUI library (by providing a set of classes that can be attached to GUI components), rather than completely wrapping it. As such, it is highly recommended to have an understanding of how the `BigWorld.GUI` library works before starting to use PyGUI. See the Client Programming Guide chapter *Graphical User Interface (GUI)* for details.

The scripts for PyGUI can be found at `fantasydemo/res/scripts/client/Helpers/PyGUI`. Each individual script is not too long, so it is a good idea to read through the scripts as they are mentioned in this document to get a better handle on how they work. The PyGUI library is used by the FDGUI module in `fantasydemo/res/scripts/client/FDGUI`, which can be used as an example game-specific high-level GUI implementation.

While the library could be used as-is to build a game UI, it has been built specifically for the needs of the FantasyDemo and may need tweaking to suit other projects. There are two ways it could be utilised:

- Copy the PyGUI module from the FantasyDemo project and tweak it to suit any particular requirements.
- Use it as a point of reference for implementing a custom GUI framework from the ground up.

This document will describe the PyGUI module as if it were to be used as-is.

1.2. Features

The PyGUI library includes:

- Container windows (that can be dragged across the screen with the mouse).
- High-level event handling mechanisms (e.g. clicking on a button contained within the window).
- A generic "visual state" definition system for defining what components look like in different states.
- An input focusing system for mutually exclusive components (e.g. multiple edit fields on the screen at the same time).
- Tool-tips can be associated with any PyGUI component.
- A number of high-level GUI component scripts:
  - Button
  - Check-box
  - Radio-button
  - Edit field (including integration with IME)
  - Language indicator (usually used in conjunction with an edit field)
  - Draggable slider
  - Word wrapped, scrollable text buffer
  - Web browser (using the Mozilla integration)

Chapter 2. How it Works

2.1. PyGUIBase

All component scripts in PyGUI inherit from a common base class called `PyGUIBase`. This class provides common functionality to all PyGUI component scripts, including activation/deactivation management, event handler routing, and tool-tips.

Be sure to call all base class methods which are defined in `PyGUIBase` if any are overridden in the derived class, otherwise some functionality may break (e.g. input event handling methods).

2.2. .gui files

PyGUI is designed so that user interfaces can be defined in `.gui` files. To allow this, PyGUI component scripts have been setup with an appropriate factory string and can be configured by specifying named values within the `<script>` blocks in the .gui file.

Generally speaking, new GUI windows are assembled via the interactive Python console, and then saved to a `.gui` file. When the game scripts load up the window, they load from the .gui file rather than assembling the window programmatically. Existing windows can be tweaked either by modifying the .gui file externally, or by adjusting via the Python console and resaving the `.gui` file.

For example, the <script> block for a slider may look something like this:

```
<script>    "PyGUI.Slider"
    <toolTipInfo>
        <templateName>    tooltip1line    </templateName>
        <items>
            <text> The quick brown fox, etc    </text>
            <shortcut>    </shortcut>
        </items>
        <placement>    above    </placement>
    </toolTipInfo>
    <isHorizontal>    true    </isHorizontal>
    <minValue>    6.000000    </minValue>
    <maxValue>    40.000000    </maxValue>
    <stepSize>    1.000000    </stepSize>
</script>
```

Note

Some component scripts are made up of multiple components (e.g. the slider has a static background part as well as a child draggable part). The scripts do assume that the child components exist and have a certain name. To make this easier, complex components provide creator methods that create the required parts.

See the Client Programming Guide chapter *Graphical User Interface (GUI)* for detailed information about `.gui` files.

2.3. Visual states

The appearance of a component on the screen is defined by the relevant attributes on SimpleGUIComponent. By itself, this is fine for simple types of interface components that do not change state (e.g. the background texture of a window). However, most scripts (e.g. buttons) change state internally based on user input (e.g.

mouse hovering, mouse clicking) or game state (e.g. disabling a button) and these states need to be reflected to the user visually.

To help with this, the PyGUI framework allows "visual states" to be defined on the component script and to be saved with the .gui files. The component script will use information from these visual states to update attributes on SimpleGUIComponent when the internal state has changed.

For example, buttons have 'normal' and 'hover' as some of its possible states. The button will use 'normal' as the default state on creation, and the internal state changes to 'hover' when the mouse enters the region of the button. Each state has a visual state block which can be configured be configured to change attributes such as the texture, UV mapping, the colour of the component, or even change the way the component is blended onto the screen. The component script will automatically switch apply the appropriate visual state as the internal state changes. If a required visual state has not been defined for a component script, warnings will be printed to the Python console.

2.3.1. Visual state Python class

A visual state is represented in script by a specialised Python class which derives from the base `VisualState` class. For example, `ButtonVisualState` corresponds to the Button component script.

There is one `VisualState` instance per visual state on a component script, and can be set by name by calling `componentScript.setVisualState( name, stateInstance )`.

The visual states will be automatically serialised to and from the .gui files.

2.3.2. Visual state XML schema

The exact visual state properties that are available depends on the script that it is attached to, however they all use the following basic schema within the `<script>` tag:

```
<script>
    ...
    <visualStates> PyGUI.VisualStateClassName
        <statename1>
            <property1>      value </property1>
            </property2>    value </property2>
            ...
        </statename1>
        <statename2>
            ...
        </statename2>
    </visualStates>
</script>
```

So for example, a button may look something like:

```
<visualStates>    PyGUI.ButtonVisualState
    <normal>
        <textStyle>    ButtonNormal    </textStyle>
        <icon>
            <textureName>    gui/maps/gui_general.dds    </textureName>
            <materialFX>    BLEND    </materialFX>
            <mapping>    PIXEL
                <coords0>    0.000000 110.000000    </coords0>
                <coords1>    0.000000 131.000000    </coords1>
                <coords2>    21.000000 131.000000    </coords2>
                <coords3>    21.000000 110.000000    </coords3>
            </mapping>
            <colour>    255.000000 255.000000 255.000000 255.000000    </
colour>
        </icon>
    </normal>
    <pressed>
        <textStyle>    ButtonPressed    </textStyle>
        <icon>
            <textureName>    gui/maps/gui_general.dds    </textureName>
            <materialFX>    BLEND    </materialFX>
            <mapping>    PIXEL
                <coords0>    22.000000 110.000000    </coords0>
                <coords1>    22.000000 131.000000    </coords1>
                <coords2>    43.000000 131.000000    </coords2>
                <coords3>    43.000000 110.000000    </coords3>
            </mapping>
            <colour>    255.000000 255.000000 255.000000 255.000000    </
colour>
        </icon>
    </pressed>
    ...
</visualStates>
```

2.3.3. External Visual States

To avoid duplication when the same visual state definitions are used across multiple instances of the same script (e.g. all buttons in the game may have the same look), visual states can be defined in external XML files and references from within the .gui file.

To reference an external visual state, use the `<external>` tag within the `<visualStates>` block. The external XML file would have a single root tag, as of the root tag is the `<visualStates>` block from above. For example,

```
<visualStates>
    <external>    gui/visual_states/standard_button.xml    </external>
</visualStates>
```

Tip

To load an existing visual state XML into a component, use the `loadVisualStates` method on the component script. For example,

```
>>> button = PyGUI.Button.create( "default_texture_name.tga",
 "Push Me" )
>>> button.script.loadVisualStates( "gui/visual_states/
standard_button.xml" )
```

2.4. Mutually Exclusive Input

Some component scripts rely on mutually exclusive keyboard input. In other words, in most cases there should only be one component that has keyboard focuse at any particular time (e.g. only one EditField should receive key events).

Since the low level BigWorld GUI library allows many component scripts to receive keyboard input at the same time (via the SimpleGUIComponent.focus property), PyGUI has a high level "focus manager". This manager makes sure only one component has their focus property set to true at any one time (as long as all component scripts grab focus via the manager).

The following functions are provided to maintain the mutually exclusive keyboard focus:

- `PyGUI.setFocusedComponent` — Takes a GUI component, removes focus from any previously focused component, sets the focus flag on the incoming component to True, and then stores a reference to this new component.
- `PyGUI.getFocusedComponent` — Returns a reference to the currently focused component (i.e. the component previously given to `setFocusedComponent`).
- `PyGUI.isFocusedComponent` — Given a component, determines if it currently is the focused component.

2.5. Tool Tips

PyGUI supports triggering tool tips on any arbitrary GUI component, since the trigger functionality is implemented in `PyGUIBase`. It also allows the definition of any number of tool tip templates to allow full control over the appearance of tool tips.

2.5.1. Tool Tip Manager

Since there can be only one tool tip active at any one time, this is managed by a singleton instance of the `ToolTipManager` class. This singleton should be initialised on application startup, simply by creating an instance of `ToolTipManager`. The scripts will raise an exception if there is already a manager instance created.

The `ToolTipManager` constructor expects two arguments: a parent GUI (tool tips will be added as children of this GUI, and is often a simple full screen invisible SimpleGUIComponent), and a Z-order to be used by the tool tips (this is usually a small number, close to zero, so that tool tips appear on top of everything else). For example,

```
toolTipMgr = PyGUI.ToolTipManager( self.guiRoot, 0.01 )
```

Once constructed, this singleton instance can also be accessed via `PyGUI.ToolTipManager.instance`.

2.5.2. Tool Tip Templates

Before tool tips can be used, one or more template GUI's need to be defined. A tool tip template has a name, and an associated .gui file, allowing much flexibility in the layout of specific tool tip types (e.g. you may have a simple one line tool tip, or you may have a complex tool tip which includes an icon image and multiple lines of text).

To register a template, do so on startup by calling `ToolTipManager.addToolTipTemplate` and provide a template name and a template .gui filename. For example,

```
toolTipMgr.addToolTipTemplate( "simple", "gui/simple_tooltip.gui" )
```

Component scripts will refer to the tool tip template by name.

The tool tip should have a script attached to it which is an instance of, or derives from, `PyGUI.ToolTip`. If deriving a new class, it can override the `doLayout` method to implement dynamic layouts for the tool tip content. For an example, see `FDGUIOneLineToolTip` in `fantasydemo/res/scripts/helpers/FDGUI/FDToolTip.py`.

2.5.3. Tool Tip Info

To associate a tool tip with a particular component, a ToolTipInfo instance needs to be attached to the PyGUIBase script. A ToolTipInfo has a number of different properties:

- `templateName` — A string name of a registered template
- `items` — A dictionary containing a set of named items. When a tool tip is activated, it will go through each key in this dictionary and look for a component on the template GUI with the key name. If the component it finds is a `Text` component it will set the text to the dictionary value, otherwise it will set the `textureName` property based on the value.
- `delayType` — A string containing the type of delay to use before displaying the tool tip. Specify 'delay' to show the tool tip only after half a second, or 'immediate' to show it immediately.
- `placement` — A string defining the placement type for this tool tip. This can be 'below', 'above', 'left', or 'right'.

This tool tip information is serialised to the .gui file once it has been attached to a `PyGUIBase` component. An example `toolTipInfo` serialisation is,

```
<script> &quot;PyGUI.Button&quot;
    ...
    <toolTipInfo>
        <templateName> tooltip1line </templateName>
        <items>
            <line1> Toggles the display of the inventory window. </line1>
            <shortcut> (I) </shortcut>
            <title> Inventory </title>
        </items>
    </toolTipInfo>
    ...
</script>
```

This example assumes that the 'tooltip1line' template has three Text component's on it: 'line1', 'shortcut', and 'title'.

2.6. Draggable Components

PyGUI provides generic functionality for components that can be dragged around the screen with the mouse. In PyGUI this is used by the DraggableWindow and the Slider scripts. In FDGUI this is also used for dragging inventory items out of the inventory window and across the screen.

There can only be one component being dragged at any one time, and this is transparently handled by `PyGUI.DraggableComponent.ComponentDragManager`.

Draggable functionality is implemented in `PyGUI.DraggableComponent`. Any component script that derives from `PyGUI.DraggableComponent` automatically becomes draggable, and can be configured to be either draggable only in the horizontal direction, only the vertical direction, or both. It can also be configured to restrict the component to be only draggable within the confines of the screen or it's parent Window.

These options are configured by passing in the appropriate parameters to the DraggableComponent constructor:

```
DraggableComponent.__init__( self, horzDrag, vertDrag, restrictToParent )
```

If the derived class implements any of the following methods, then it must call the methods on the `DraggableComponent` base class:

- `onSave( self, dataSection )`
- `onLoad( self, dataSection )`
- `handleMouseButtonEvent( self, comp, event )`

Chapter 3. Integration

There are a number of script hooks that need to be in place in order for PyGUI to function correctly.

3.1. Personality script integration

To allow the PyGUI framework to handle some logic at a global level, the personality script needs to call some PyGUI event handlers.

- In `BWPersonality.handleKeyEvent`, call `PyGUI.handleKeyEvent`:

```
def handleKeyEvent( event):
    ...
    handled = PyGUI.handleKeyEvent( event )
    if not handled:
        handled = GUI.handleKeyEvent( event )
    ...
```

- In `BWPersonality.handleMouseEvent()`, call `PyGUI.handleMouseEvent()`:

```
def handleMouseEvent( event ):
    ...
    handled = PyGUI.handleMouseEvent( event )
    if not handled:
        handled = GUI.handleMouseEvent( event )
    ...
```

- In `BWPersonality.handleIMEEvent`, call `PyGUI.handleIMEEvent` to allow edit fields to cope with IME state changes (see below for more details on IME).

```
def handleIMEEvent( event ):
    return PyGUI.handleIMEEvent( event )
```

- In `BWPersonality.handleLangChangeEvent`, call `PyGUI.handleInputLangChangeEvent` to update components such as the language indicator.

```
def handleInputLangChangeEvent():
    return PyGUI.handleInputLangChangeEvent()
```

- In `BWPersonality.onRecreateDevice()`, call `PyGUI.onRecreateDevice` to allow the GUI to cope with the screen size changing.

```
def onRecreateDevice():
    ...
    PyGUI.onRecreateDevice()
    ...
```

3.2. Factory strings

To allow the GUI system to successfully load .gui files and attach class instances to the components, the PyGUI module needs to be loaded into the main Python namespace. This can be done by putting the following lines at the top of the client personality script:

```
from Helpers import PyGUI
__import__('__main__').PyGUI = PyGUI
```

This is neccessary because PyGUI factory strings are of the form `PyGUI.ClassName` (e.g. `PyGUI.Button`) and the GUI system needs to be able to find the `PyGUI` module at the top level.

In order to be able to load your own classes derived from `PyGUI.Window`, you will need to also add your game specific GUI module in a smiliar fashion. For example, if you place all your game specific GUI scripts into a module called GameGUI, then you would put this near the top of the personality script:

```
import GameGUI
__import__('__main__').GameGUI = GameGUI
```

3.3. Input Method Editor

PyGUI includes an IME presentation layer implementation, which is contained within the `PyGUI.IME` module. This module implements the IME display as a set of component singletons, which will display themselves at a specified location (e.g. at the cursor carat of the EditField) based on the current IME state. The EditField script uses this module.

To allow IME to work, `PyGUI.IME.init()` should be called on startup. This will initialise the singleton components, which are initially hidden until the user starts using an IME enabled EditField.

Call `PyGUI.IME.fini()` on shutdown to clean up these components.

Note

See the Client Programming Guide chapter *Input Method Editors (IME)* for detailed information about IME integration.

3.4. Testing

To test if the integration is working, use the PyGUI test window script:

```
>>> import Helpers.PyGUI.Tests
>>> Helpers.PyGUI.Tests.Window.test()
```

This will clear the existing GUI and display a draggable window with a number of basic components. If all is well, then you will be able to interact with these components.

Note

The test window script assumes the following resources exist:

- `gui/tests/window.gui`
- `gui/visual_states/button_test.xml`
- `gui/visual_states/checkbox_test.xml`

Chapter 4. Constructing an Interface

This section outlines the typical process of creating a new user interface with PyGUI. An individual user interface will usually be a window that encapsulates some part of the game, for example the chat window or the graphics settings window.

4.1. Window script

The first step in creating a new window is to derive a new class which will be used to handle events generated by the components on the window. A minimal window that has no logic would look like this:

```
from Helpers.PyGUI import Window

class MyFirstWindow( Window ):
    factoryString = "GameGUI.MyFirstWindow"

    def __init__( self, component ):
        Window.__init__( self, component )
```

The class should include a factory string class property, which is used by the engine to know how to re-attach the class when deserialising from a .gui file. Note the `GameGUI` module name here is just for illustrative purposes - this is a module created by you and is specific to your game (for example FantasyDemo has a module called `FDGUI`).

Tip

To make a Window draggable (rather than static), derive from `DraggableWindow` rather than `Window`.

4.2. Window component

Once there is a script to attach to the window component, it can be created at the interactive Python console in the client.

```
>>> wnd = GameGUI.MyFirstWindow.create( "gui/maps/window_background.tga" )
>>> ... # setup properties like .width and .height as required for your window
```

Once the appopriate properties have been set on the `WindowGUIComponent` as required, save it to a .gui file.

```
>>> wnd.save( "gui/myfirstwindow.gui" )
```

4.3. Adding components to the window

Adding components to the window such as buttons and edit boxes is simply a case of adding child GUI components and attaching the appropriate script. Some scripts require a specific structure of multiple components with the correct naming scheme to allow them to work (e.g. a slider consists of a background component and a "thumb" slider component), so to ease creation most component scripts provide helper "create" functions which assemble the required components in the correct way.

For example, the Button class has a static create method which creates a parent component that acts as the background image and input event handler, and a child text component for its label. As such, a Button can be created with a single call:

```
>>> button = PyGUI.Button.create( "gui/maps/button-background.tga", "Button
 Text!" )
>>> wnd.addChild( button, "someButtonName" )
>>> button.position.x = 0.2
>>> button.position.y = -0.1
>>> ...
>>> wnd.save( "gui/myfirstwindow.gui" )
```

See *Component Script Reference* on page 19 for each script to see how they can be created.

4.3.1. Attaching existing visual states

To attach a pre-existing external visual state XML definition to a component, use the `loadVisualStates` method on the component script. For example,

```
>>> button.script.loadVisualStates( "gui/visual_states/
ingame_menu_button.xml" )
```

4.4. Handling events

Events generated by component scripts (e.g. button click, slider movement) are handled by the window script. A Python decorator called `PyGUIEvent` is provided to make this as simple as possible.

The `PyGUIEvent` decorator takes two arguments, the name of the child component that generates the event and the name of the event to handle. For example, the `onClick` event for the button created in the previous section can be handled:

```
class MyFirstWindow( Window ):
    ...
    @PyGUI.PyGUIEvent( "someButtonName", "onClick" ):
    def theButtonGotClicked( self ):
        print "HELLO!"
```

Note

Use dotted notation if the component that generates the event is several children deep in the GUI hierarchy. For example, if the button is contained within a sub-window named "buttonContainer", then the decorator would be written as,

```
@PyGUI.PyGUIEvent( "buttonContainer.someButtonName",
 "onClick"
```

This can be any number of children deep.

`PyGUIEvent` supports partial method parameter completion, which can be put to effective use by stacking multiple decorators onto the same method. A simple example of this would be if you had multiple buttons on the window that perform similar operations:

```
class MyFirstWindow( Window ):
    ...
    @PyGUI.PyGUIEvent( "helloButton1", "onClick", 1 ):
    @PyGUI.PyGUIEvent( "helloButton2", "onClick", 2 ):
```

```
    @PyGUI.PyGUIEvent( "helloButton3", "onClick", 3 ):
    def onHelloButtonClick( self, numTimes ):
        for x in range(numTimes):
            print "HELLO!"
```

Any number of arguments and key word arguments can be pre-filled in this way.

4.5. Loading and displaying the window

To display the window at run-time, load the window from disk and then activate it. Keep in mind that since it can take some time to load the .gui file and related resources (such as textures), it is better to load it once on startup and keep initially hidden until required. For example:

```
...
# on GameGUI initialisation
self.myFirstWindow = GUI.load( "gui/myfirstwindow.gui" )

...
# somewhere else in a key event handler
if event.key == Keys.ESCAPE:
    self.myFirstWindow.script.active( True )
```

4.6. Tweaking existing windows

There are two ways you can modify existing user interfaces.

- Load the .gui file in the client, modify it on the Python console, then resave it. This is often the easiest method when making more extensive modifications.
- Edit the .gui file in a text editor. This is good for small tweaks, such as moving the position of a component slightly.

Chapter 5. Component Script Reference

5.1. Window

Since a group of user interface functionality is usually encapsulated within its own window (e.g. chat window, options window, etc), typical usage of PyGUI is to have the Window class at the top level of a particular .gui file. The .gui file will then be loaded and added to the GUI hierarchy at the appropriate time in the game scripts (e.g. show the options window when a button is pressed).

There is no helper create method for window, since there is no specialised hierarchy. Just create a `WindowGUIComponent` and attach a Window script instance to it.

Tip

Derive your Window script from `DraggableWindow` rather than Window in order to make it draggable with the mouse.

5.2. Button

The Button script provides standard push-button functionality. In addition, they can be configured so that they can be toggled between an active state and an inactive state.

5.2.1. Component hierarchy

```
WindowGUIComponent — background texture, handles input events
    -> TextGUIComponent — optional text label, must be named "label"
```

5.2.2. Creation

A create method is supplied and takes the following keyword arguments:

```
Button.create( [texture], [visualStates], [label], [toggle] )
```

5.2.3. Properties

Buttons have a number of properties which control it's behaviour, and are serialised to the GUI XML file.

- `buttonStyle` — by default this is 'pressbutton', but if it set to 'togglebutton' it will become a toggle-type button.
- `buttonDisabled` — determines if this button is disabled (i.e. cannot be pressed by the user). Use the `setDisabledState(state)` method to change this state.
- `buttonActive` — if this is a toggle style button, then this is True when toggled to the active state. Use the `setToggleState(state)` method to change this state.

5.2.4. Events

Button can raise the following events:

- `onClick()` — the user has clicked the button.
- `onActivate()` — the toggle button has been activated.
- `onDeactivate()` — the toggle button has been deactivated.

5.2.5. Visual states

Button has the following visual states:

- `normal` — the button is idle, this is the default state.
- `hover` — the user is hovering the mouse over the button.
- `pressed` — the user is currently pressing down on the button.
- `disabled` — the button is currently disabled and can't be used by the user.
- `active` — the toggle button is idle, but is currently in it's active state.
- `hover_active` — the user is hovering the mouse over the toggle button while it is active.
- `pressed_active` — the user is currently pressing down the button while it is active.
- `disabled_active` — the toggle button is disabled, but is active.

The button uses the `ButtonVisualState` class to determine its appearance and has the following properties:

- `textStyle` — chooses a font via a style name defined in `TextStyles.py`.
- `icon` — defines how the background texture gets applied, and has the following sub-properties:
  - `textureName` — the name of the texture resource.
  - `materialFX` — the blending type used to display the texture (uses the same value names as `SimpleGUIComponent.materialFX`).
  - `mapping` — the rectangle within the texture, specified by four UV coordinates.
  - `colour` — colour tint, corresponds to the `SimpleGUIComponent.colour` property.
- `pressed` — the user is currently pressing down on the button.
- `disabled` — the button is currently disabled and can't be used by the user.
- `active` — the toggle button is idle, but is currently in it's active state.
- `hover_active` — the user is hovering the mouse over the toggle button while it is active.
- `pressed_active` — the user is currently pressing down the button while it is active.
- `disabled_active` — the toggle button is disabled, but is active.

5.2.6. Example

```
import Helpers.PyGUI as PyGUI

class TestWindow( PyGUI.Window ):

    @PyGUI.PyGUIEvent( "button", "onClick" )
    def buttonClicked( self ):
        print "Button Pushed"


def test():
    wnd = TestWindow.create( "system/maps/col_white.bmp", bind=False )
    wnd.colour = (128,128,128,255)
    wnd.materialFX = "BLEND"
    wnd.width = 1.0
    wnd.height = 1.0
    wnd.button = PyGUI.Button.create( label="Push Me",
visualStates="button_test.xml" )
    wnd.script.onBound()
    return wnd
```

5.3. CheckBox

A check box is similar to a Button (in fact, it derives from the `Button` class), however it's hierarchy is slightly modified to cope with the different layout of a checkbox.

5.3.1. Component hierarchy

```
WindowGUIComponent – invisible background, handles input events
    -> SimpleGUIComponent – check icon, must be named "box"
    -> TextGUIComponent – text label, must be named "label"
```

5.3.2. Creation

A create method is supplied and takes the following keyword arguments:

```
CheckBox.create( [texture], [visualStates], [label] )
```

5.3.3. Properties

The CheckBox inherits the properties of Button. The `buttonStyle` property must be set to 'checkbox' to access checkbox functionality.

5.3.4. Events

CheckBox has the same events as Button.

5.3.5. Visual states

CheckBox has the same visual states as Button.

5.3.6. Example

```
import Helpers.PyGUI as PyGUI


class TestWindow( PyGUI.Window ):

    @PyGUI.PyGUIEvent( "check1", "onActivate", True )
    @PyGUI.PyGUIEvent( "check1", "onDeactivate", False )
    def check1Activated( self, activated ):
        print "Checkbox 1 Clicked", activated

    @PyGUI.PyGUIEvent( "check2", "onActivate", True )
    @PyGUI.PyGUIEvent( "check2", "onDeactivate", False )
    def check2Activated( self, activated ):
        print "Checkbox 2 Clicked", activated
```

```
def test():
    wnd = TestWindow.create( "system/maps/col_white.bmp", bind=False )
    wnd.colour = (128,128,128,255)
    wnd.materialFX = "BLEND"
    wnd.width = 1.0
    wnd.height = 1.0

    wnd.check1 = PyGUI.CheckBox.create( label="Check Box 1",
 visualStates="checkbox_test.xml" )
    wnd.check2 = PyGUI.CheckBox.create( label="Check Box 2",
 visualStates="checkbox_test.xml" )

    wnd.check1.position.y = +0.25
    wnd.check2.position.y = -0.25

    wnd.script.onBound()
    return wnd
```

5.4. RadioButton

A RadioButton behaves and looks similar to the CheckBox, however they can be placed in named groups so that only one radio box within the same group can be checked at any one time. The group is determined by a unique group name among its siblings.

5.4.1. Component hierarchy

RadioButton has the same component hierarchy as CheckBox.

5.4.2. Creation

A create method is supplied and takes the following keyword arguments:

```
RadioButton.create( [texture], [label], [groupName], [visualStates] )
```

5.4.3. Properties

The RadioButton inherits the properties of Button. The `buttonStyle` property must be set to 'radiobutton' to access checkbox functionality. In addition to the base class properties, the RadioButton uses some additional properties:

- `groupName` — this defines the mutual exclusivity group. All sibiling radio buttons with the same group name will be mutually exclusive to one another.

5.4.4. Events

RadioButton has the same events as Button.

5.4.5. Visual states

RadioButton has the same visual states as Button.

5.4.6. Example

```
import Helpers.PyGUI as PyGUI
```

```
class TestWindow( PyGUI.Window ):

    @PyGUI.PyGUIEvent( "radio1A", "onActivate", "A" )
    @PyGUI.PyGUIEvent( "radio1B", "onActivate", "B" )
    @PyGUI.PyGUIEvent( "radio1C", "onActivate", "C" )
    def group1OptionSelected( self, item ):
        print "Selected item", item, "from group 1."

    @PyGUI.PyGUIEvent( "radio2A", "onActivate", "A" )
    @PyGUI.PyGUIEvent( "radio2B", "onActivate", "B" )
    @PyGUI.PyGUIEvent( "radio2C", "onActivate", "C" )
    def group2OptionSelected( self, item ):
        print "Selected item", item, "from group 2."


def test():
    wnd = TestWindow.create( "system/maps/col_white.bmp", bind=False )
    wnd.colour = (128,128,128,255)
    wnd.materialFX = "BLEND"
    wnd.width = 1.0
    wnd.height = 1.5

    wnd.radio1A = PyGUI.RadioButton.create( label="Group 1, Item A",
 groupName="group1", visualStates="checkbox_test.xml" )
    wnd.radio1B = PyGUI.RadioButton.create( label="Group 1, Item B",
 groupName="group1", visualStates="checkbox_test.xml" )
    wnd.radio1C = PyGUI.RadioButton.create( label="Group 1, Item C",
 groupName="group1", visualStates="checkbox_test.xml" )

    wnd.radio1A.verticalPositionMode = "CLIP"
    wnd.radio1B.verticalPositionMode = "CLIP"
    wnd.radio1C.verticalPositionMode = "CLIP"

    wnd.radio1A.position.y = +0.75
    wnd.radio1B.position.y = +0.50
    wnd.radio1C.position.y = +0.25


    wnd.radio2A = PyGUI.RadioButton.create( label="Group 2, Item A",
 groupName="group2", visualStates="checkbox_test.xml" )
    wnd.radio2B = PyGUI.RadioButton.create( label="Group 2, Item B",
 groupName="group2", visualStates="checkbox_test.xml" )
    wnd.radio2C = PyGUI.RadioButton.create( label="Group 2, Item C",
 groupName="group2", visualStates="checkbox_test.xml" )

    wnd.radio2A.verticalPositionMode = "CLIP"
    wnd.radio2B.verticalPositionMode = "CLIP"
    wnd.radio2C.verticalPositionMode = "CLIP"

    wnd.radio2A.position.y = -0.25
    wnd.radio2B.position.y = -0.50
    wnd.radio2C.position.y = -0.75

    wnd.script.onBound()
    return wnd
```

5.5. Slider

The Slider class allows the user to interpolate between a minimum value and a maximum value. Both horizontal and vertical sliders are supported. It consists of a parent component which acts as the main container, and has a child component which acts as the draggable "thumb".

5.5.1. Component hierarchy

```
WindowGUIComponent — background texture, handles input events
    -> SimpleGUIComponent — draggable thumb, must be named "thumb"
```

5.5.2. Creation

A create method is supplied and takes the following keyword arguments:

```
Slider.create( [texture], [thumbTexture], [isHorizontal], [minValue],
 [maxValue], [visualStates] )
```

5.5.3. Events

- `onValueChanged( value )` — called whenever the slider has moved.
- `onBeginDrag( value )` — called when the user has just started dragging the slider.
- `onEndDrag( value )` — called when the user has finished dragging the slider.

5.5.4. Visual states

- `normal` — the slider is idle, this is the default state.
- `hover` — the user is hovering the mouse over the slider thumb.
- `pressed` — the user is currently pressing down on the slider thumb.
- `disabled` — the slider is currently disabled and can't be used by the user.

The slider uses the `SliderVisualState` class to determine its appearance and has the following properties:

- `background` - defines how the background texture gets applied, and has the following sub-properties:
  - `textureName` — the name of the texture resource.
  - `mapping` — the rectangle within the texture, specified by four UV coordinates.
  - `colour` — colour tint, corresponds to the `SimpleGUIComponent.colour` property.
- `thumb` — defines how the "thumb" appears, and has the following sub-properties:
  - `textureName` — the name of the texture resource.
  - `mapping` — the rectangle within the texture, specified by four UV coordinates.
  - `colour` — colour tint, corresponds to the `SimpleGUIComponent.colour` property.

5.5.5. Example

```
import Helpers.PyGUI as PyGUI


class TestWindow( PyGUI.Window ):
    @PyGUI.PyGUIEvent( "slider", "onValueChanged" )
    def sliderValueChanged( self, value ):
        print "Slider Value Changed", value

    @PyGUI.PyGUIEvent( "slider", "onBeginDrag" )
    def sliderBeginDrag( self, value ):
        print "Slider begin drag", value

    @PyGUI.PyGUIEvent( "slider", "onEndDrag" )
    def sliderEndDrag( self, value ):
        print "Slider end drag", value


def test():
    wnd = TestWindow.create( "system/maps/col_white.bmp", bind=False )
    wnd.colour = (128,128,128,255)
    wnd.materialFX = "BLEND"
    wnd.width = 1.0
    wnd.height = 1.0

    wnd.slider = PyGUI.Slider.create( visualStates="slider_test.xml" )

    wnd.script.onBound()
    return wnd
```

5.6. EditField

The EditField is a single line text input edit box. It supports advanced keyboard navigation, horizontal scrolling, Unicode text input and can make use of the PyGUI IME framework. It also supports automatically choosing a new font based on a provided list of possible fonts which best matches the current screen resolution.

The EditField makes use of the PyGUI keyboard input mutual exclusivity framework.

Use the `EditField.setText` and `EditField.getText` methods to access the current text in the edit field.

5.6.1. Component hierarchy

```
WindowGUIComponent – background texture, horizontal scrolling, handles input
 events
   -> TextGUIComponent – the current text content, must be named "text"
```

5.6.2. Creation

A create method is supplied and takes the following arguments:

```
EditField.create( backgroundTextureName, [activeColour], [inactiveColour] )
```

5.6.3. Properties

The edit field has a number of properties which control it's behaviour, and are serialised to the GUI XML file.

- `activeColour` — this defines the colour of the text when the edit field currently has focus.
- `inactiveColour` — this defines the colour of the text when the edit field does not have focus.

- `enabled` — if this is False, then the user cannot focus the edit field and input text.
- `maxLength` — the maximum length of the edit field, in characters.
- `enableIME` — if True, then IME functionality is enabled on the edit field.
- `focusViaMouse` — if True, then the edit field can be brought into focuse by clicking on it with the mouse.
- `idealVisibleCharacters` — the number of characters which should be ideally visible and is used to determine the best font to use from the autoFont list.
- `autoSelectionFonts` — a list of font names that are chosen based on the current resolution and idealVisibleCharacters. This is serialised as a set of `<autoFont>` tags.

5.6.4. Events

- `onEnter( text )` — called whenever the user presses enter with the EditField in focus.
- `onEscape`() — called whenever the user presses escape with the EditField in focus.
- `onChangeFocus( state )` — called whenever the edit field component changes input focus.

5.6.5. Visual states

The edit field does not currently use the visual state system.

5.6.6. Example

```
import Helpers.PyGUI as PyGUI

class TestWindow( PyGUI.Window ):

    @PyGUI.PyGUIEvent( "editField", "onEnter" )
    def editFieldEnterPressed( self, text ):
        print "You entered:", text
        self.component.editField.script.setText( "" )

    @PyGUI.PyGUIEvent( "editField", "onChangeFocus" )
    def editFieldOnChangeFocus( self, state ):
        print "Edit Field Focus Changed", state


def test():
    wnd = TestWindow.create( "system/maps/col_white.bmp", bind=False )
    wnd.colour = (128,128,128,255)
    wnd.materialFX = "BLEND"
    wnd.width = 1.0
    wnd.height = 1.0

    wnd.editField = PyGUI.EditField.create( texture="system/maps/
col_black.bmp" )
    wnd.editField.script.enableIME = True

    wnd.editField.indicator = PyGUI.LanguageIndicator.create()
    wnd.editField.indicator.horizontalPositionMode = "CLIP"
    wnd.editField.indicator.horizontalAnchor = "RIGHT"
    wnd.editField.indicator.position.x = 1

    wnd.script.onBound()
    return wnd
```

5.7. LanguageIndicator

A language indicator is a specialised non-interactive component. The job of the indicator is to display to the user the name of the current input locale (e.g. "EN" for English, "FR" for French, "CN" for Chinese, etc). It is often used in conjunction with an EditField.

5.7.1. Component hierarchy

```
WindowGUIComponent – background texture
    -> TextGUIComponent – text showing the language indication, must be named
 "label"
```

5.7.2. Creation

A create method is supplied and takes the following arguments:

```
LanguageIndicator.create( autoSize=True, padding=5, square=True )
```

5.7.3. Properties

The language indicator has a number of properties which control it's behaviour, and are serialised to the GUI XML file.

- `autoSize` — if this is True, then the indicator will automatically resize itself to fit around the indication label.
- `padding` — only effective if autoSize is True, this will add some padding (in pixels) around the indication label.
- `square` — only effective if autoSize is True, this will maintain a square aspect ratio at all times.

5.7.4. Events

LanguageIndicator does not raise any events.

5.7.5. Visual states

LanguageIndicator does not have any visual states.

5.7.6. Example

See the EditField section for a usage example of the LanguageIndicator.

5.8. ScrollableText

The ScrollableText class implements a wod wrapped, vertically scrolling buffer of text. It supports colour formatting (using the formatting tags supported by TextGUIComponent) and automatically chooses the most appropriate font out of a list that best matches the current screen resolution.

Use the `ScrollableText.appendLine( str )` method to add a new line. If the given string contains line breaks, it will automatically break the line into multiple lines at those points.

5.8.1. Component hierarchy

```
WindowGUIComponent – background texture, implements vertical scrolling
```

```
    -> TextGUIComponent — a multiline text component which implements the
 buffer, must be named "text"
```

5.8.2. Creation

A create method is supplied and takes the following arguments:

```
ScrollableText.create()
```

5.8.3. Properties

ScrollableText has a number of properties which control it's behaviour, and are serialised to the GUI XML file.

- `autoSelectionFonts` — a list of font names that are chosen based on the current resolution and idealCharactersPerLine. This is serialised as a set of `<autoFont>` tags.
- `idealCharactersPerLine` — the number of characters which should be ideally visible and is used to determine the best font to use from the autoFont list.
- `maxLines` — maximum number of lines to store before clipping off the oldest line
- `wordWrap` — boolean whether or not to word wrap the text

5.8.4. Events

ScrollableText does not raise any events.

5.8.5. Visual states

ScrollableText does not have any visual states.

5.8.6. Example

```
import Helpers.PyGUI as PyGUI
import Keys

class TestWindow( PyGUI.Window ):

    def __init__( self, component ):
        PyGUI.Window.__init__( self, component )
        component.focus = True

    def handleKeyEvent( self, event ):
        handled = False
        if event.isKeyDown():
            if event.key == Keys.KEY_PGUP:
                self.component.textArea.script.scrollUp()
                handled = True
            elif event.key == Keys.KEY_PGDN:
                self.component.textArea.script.scrollDown()
                handled = True
        return handled


def test():
    wnd = TestWindow.create( "system/maps/col_white.bmp", bind=False )
    wnd.colour = (128,128,128,255)
    wnd.materialFX = "BLEND"
    wnd.width = 1.0
    wnd.height = 1.0

    wnd.textArea = PyGUI.ScrollableText.create()
    wnd.textArea.colour = (0,0,0,255)

    wnd.textArea.script.appendLine( "This is a line of text." )
    wnd.textArea.script.appendLine( "The quick brown fox jumped over the lazy
dog." )
    wnd.textArea.script.appendLine( "A red line", (255,0,0,255) )
    wnd.textArea.script.appendLine( "\c00FF00FF;Multi \c00FFFFFF;coloured
\cFFFF00FF;line!" )
    wnd.textArea.script.appendLine( "A bit more text." )
    wnd.textArea.script.appendLine( "And even more." )
    wnd.textArea.script.appendLine( "Press PGUP and PGDN to scroll..." )

    wnd.script.onBound()
    return wnd
```