Windward Game Library Documentation

Release 0.6.1

Gregory McWhirter

April 14, 2015

This library is a wrapper around Pyglet 1.2 to (hopefully) make it a bit easier to work with for students who are still learning python. There are almost certainly still bugs to fix and improvements to make.

The place to start with any game is a `wwgamelib.Game` object. The Game object controls starting/stopping the game and synchronizing what should be shown on the screen.

The Game object will have a collection of `wwgamelib.layer.Layer` objects where most of the work is done. Some Layer objects might be `wwgamelib.menu.Menu` objects, others might be `wwgamelib.map.Map` objects or a custom Layer that controls a scene.

Contents:

CHAPTER 1

wwgamelib package

The main wwgamelib package includes two main classes: `Game` and `GameObject`.

The `Game` class handles the main game functionality – running, stopping, pausing, etc. It also serves as a master control for the game's :py:class:'wwgamelib.layer.Layer's. It controls which layers are shown and which are hidden.

The `GameObject` class is a wrapper around python's standard object class. The purpose is to have subclasses that save a reference to the main `Game` object as a `weakref`. This makes the main game accessible from each of the derived objects, so that properties such as window size can be easily found without the use of global variables.

This package also has several submodules, which are listed below. Each provides a wrapper around various `pyglet` functionality in a way that is (hopefully) more cohesive and easier to understand.

1.1 Submodules

1.1.1 wwgamelib.label module

Label objects are used to put text into a game. This is a wrapper around the Pyglet labels that makes the interface much more similar to a Sprite than it would otherwise be. This module holds the code related to `Label` objects.

class `wwgamelib.label.`**`Label`**(*game*, *text='‘*, *font_name=None*, *font_size=None*, *bold=False*, *italic=False*, *color=(255*, *255*, *255*, *255)*, *x=0*, *y=0*, *width=None*, *height=None*, *align='left'*, *multiline=False*, *batch=None*, *group=None*)

Inherits from `GameObject` and `pyglet.text.Label`

Wrapper for pyglet labels. It adds some nice functions for hiding and showing labels.

Variables

- **`position`** – A tuple giving the current position by top-left corner.
- **`rgroups`** – A tuple of the render groups for this label.

Args: game: (`Game`) The game object that this label belongs to. It will be stored as a `weakref`

text: (string) The text to display

font_name: (string or list) Font family names to use. If it is a list, the first one available is used.

font_size: (float) Font size (in pt)

bold: (bool) Whether the font is bold or not

italic: (bool) Whether the font is italic or not

color: (tuple) An (int, int, int, int) tuple representing an RGBA color x: (int) The x-coordinate of the label

y: (int) The y-coordinate of the label

width: (int or None) The width of the label

height: (int or None) The height of the label

align: (string) Horizontal alignment of the text (if width is supplied). One of "left", "center", or "right".

multiline: (bool) Whether to word-wrap at newline characters or not. If true, a width must also be set.

batch: (`pyglet.graphics.Batch`) The render batch for the label

group: (`pyglet.graphics.Group`) The render group for the label

`_delete`()
Calls self.delete, but also logs the action.

`_get_rgroups`()
Property accessor for the rgroups property. Should not be called directly.

Returns: tuple: A tuple of the (`Label.top_group`, `Label.background_group`, `Label.foreground_group`, `Label.foreground_decoration_group`)

`_set_rgroups`(*top*, *bg*, *fg*, *fgd*)
Property accessor for the rgroups property. Should not be called directly.

`clear_batch`()
Clears the label's render batch, setting it to have its own batch.

Returns: `Label`: The current instance (for chaining method calls)

`get_position`()
Gets the position as a tuple – used to set up the position property

Returns: tuple: The current x,y pair of the upper-left corner.

`hide`()
Makes the label invisible.

Returns: `Label`: The current instance (for chaining method calls)

`position`
The current position of the top-left corner in (x,y) form.

`rgroups`
A tuple of (`Label.top_group`, `Label.background_group`, `Label.foreground_group`, `Label.foreground_decoration_group`)

`set_batch`(*batch*)
Sets the label's render batch.

Args: batch: (`pyglet.graphics.Batch`) The render batch to add this label to.

Returns: `Label`: The current instance (for chaining method calls)

`set_group`(*group*)
Sets the label's render group.

Args: group: (`pyglet.graphics.Group`) The group to base things from. The actual render-group orders are automatically calculated.

Returns: `Label`: The current instance (for chaining method calls)

`set_position`(*x*, *y*)

Sets the position – useful for handling a `Label` in the same way as you can handle a `Sprite`.

Args: x: (int) The desired x coordinate.

y: (int) The desired y coordinate.

Returns: `Label`: The current instance (for chaining method calls)

`set_position_by_center`(*x*, *y*)

Sets the position of this label according to where the center of it should be.

Calculates where the upper-left corner should be based on the desired center coordinates and the current width and height, then calls `Label.set_position()` to actually set it.

Args: x: (int) The desired x coordinate of the center.

y: (int) The desired y coordinate of the center.

Returns: `Label`: The current instance (for chaining method calls)

`set_text`(*text*)

Sets the text for the label.

This does not recalculate the width and height.

Args: text: (string) The desired label text.

Returns: `Label`: The current instance (for chaining method calls)

`show`(*batched=False*)

Makes the label visible.

Args:

batched: (bool) Whether or not the drawing for this label is batched. If it is, then the label's `Label.draw()` method is not called.

Returns: `Label`: The current instance (for chaining method calls)

`wwgamelib.label.`**`mu_magic_numbers`**(*font*)

Returns the magic number mu based on the font

See: http://www.pearsonified.com/2012/01/characters-per-line.php

Args: font: (string) The name of the font

Returns: float: The magic number

Raises: KeyError: The font name is not recognized.

`wwgamelib.label.`**`text_height`**(*lines*, *font_size*, *padding=4*)

Calculates the height of some text given the number of lines and font size.

See: http://reeddesign.co.uk/test/points-pixels.html

Args: lines: (int) Number of lines

font_size: (int) The size of the font (in pt)

padding: (int) The number of pixels of padding to add (default 4)

Returns: int: The number of pixels high that the text is

`wwgamelib.label.`**`text_width`**(*text*, *font*, *font_size*, *cpl=72*)

Calculates the width of the text given the font size.

Args: text: (string) The text to get the size of

font: (string) The name of the font being used.

font_size: (integer) The font size (in pt)

cpl: (integer) Desired characters-per-line (default 72)

Returns: int: The width of the text in pixels

1.1.2 wwgamelib.layer module

The layer module contains the `Layer` class. Layers are a way for the game to tie related objects together. They could be used for different stages/maps (e.g., `wwgamelib.map.Map`), for menus (e.g., `wwgamelib.menu.Menu`), or for other such things.

Each layer is is a `Listener`, so it can be responsible for its own key bindings and handling its own mouse events. This is done by implementing the appropriate pyglet listener functions as methods (see, e.g., http://www.pyglet.org/doc-current/programming_guide/keyboard.html, http://www.pyglet.org/doc-current/programming_guide/mouse.html, http://www.pyglet.org/doc-current/programming_guide/input.html). This module is responsible for code for layers that should be displayed together.

It has one class: `wwgamelib.layer.Layer`.

class `wwgamelib.layer.`**`Layer`**(*game*, *hidden=False*)

Inherits from `wwgamelib.listener.Listener`

Represents a layer that can be shown/hidden all at once.

Variables

- **`_shown_components`** – A dictionary of components that are "shown" in the layer. Keys are string names.
- **`_hidden_components`** – A dictionary of components that are "hidden" in the layer. Keys are string names.
- **`_shown`** – A boolean that represents whether the layer is shown or not.

Args: game: (`wwgamelib.Game`) The game object that is host to this layer. This is stored as a `weakref`.

hidden: (bool) Whether the layer starts hidden or not (default False)

`_get_batch`(*batch*)

Gets the batch by name.

Args: batch: (string) The name of the batch to find.

Raises: KeyError: The batch with that name was not found.

`add_component`(*component*, *key=None*, *hidden=False*, *batch='default'*)

Adds the component to the appropriate list. If key is None, a uuid1 is generated as the key.

Args: component: The component to add. It will likely be a `wwgamelib.label.Label` or `wwgamelib.sprite.Sprite`.

Returns: string: the key of the component.

Raises: KeyError: the key is already in use in either shown or hidden components.

`delete_component`(*key*)

Deletes the component (or, more specifically, the reference to it) from the layer.

Args: key: (string) The name of the component to delete.

Returns: `Layer`: The current instance (for method chaining)

Raises: KeyError: no component is found with that name.

`get_component`(*key*)
Finds the component that has the given key (whether shown or hidden).

Args: key: (string) The name of the component to find.

Returns: object: The component

Raises: KeyError: no component is found with that key.

`hide`()
Hides the layer and all of its shown components.

Returns: `Layer`: The current instance (for method chaining)

`hide_component`(*compkey*)
Hides the selected component and moves it to the hidden components.

Args: compkey: (string) The name of the component to hide.

Returns: `Layer`: The current instance (for method chaining)

Raises: KeyError: The component with that name is not found.

ValueError: The component with that name is already hidden.

`is_shown`()
Returns whether or not the layer is shown.

Returns: bool: Whether the layer is shown or not.

`show`()
Shows the layer and all of its components.

Returns: `Layer`: The current instance (for method chaining)

`show_component`(*compkey*)
Shows the selected component and moves it to the shown components.

Args: compkey: (string) The name of the component to show.

Returns: `Layer`: The current instance (for method chaining)

Raises: KeyError: The component with that name is not found.

ValueError: The component with that name is already shown.

1.1.3 wwgamelib.listener module

Base class for event listeners

class `wwgamelib.listener.`**`Listener`**(*game*)
Inherits from `GameObject`

Base class for objects that listen for mouse or keyboard events.

Args: game: (`wwgamelib.Game`) The game that the listener belongs to. Stored as a `weakref`.

1.1.4 wwgamelib.map module

The map module contains 2 classes and 1 function.

The `Map` class represents a tiled m x n grid map. It uses `networkx` to store the graph representation of the grid, and uses a dictionary to hold the cells corresponding to each grid.

The cells in the map are, by default, `Cell` objects. :py:class:'~wwgamelib.map.Cell's are :py:class:'wwgamelib.GameObject's that hold a collection of objects within themselves. Additionally, they keep track of their own position, both in terms of rows and columns within the grid and in terms of pixels on the screen.

If additional functionality is required (for instance, determining whether a `Cell` should be considered blocked when a new object is added to the cell), a subclass should be declared from `Cell` and the functionality added there. When constructing a `Map`, the derived cell class can be supplied as a parameter to the constructor and it will be used instead of `Cell`.

Finally there is the function `graph_shortest_path()`. This calls the networkx function `networkx.algorithms.shortest_paths.weighted.single_source_dijkstra()`, but also takes into account that paths of infinite length are not legitimate paths. This module will provide an interface to represent a tiled map.

class `wwgamelib.map.`**`Cell`**(*game*, *parmap*)

Inherits from `wwgamelib.GameObject`

Represents a cell on the game map.

Variables

- **`_objects`** – (dict) The objects that have been attached to this cell.
- **`_blocked`** – (boolean) Whether the cell is considered blocked or not (for path-finding).
- **`_position`** – (tuple) The position (x,y) of the top-left corner of the cell.
- **`_coords`** – (tuple) The position (x,y) of the cell in the map grid (not pixels).

Args: game: (`Game`) The game object that the sprite belongs to. Stored as a `weakref`.

parmap: (`Map`) The map object that the cell belongs to. Stored as a `weakref`.

`add_object`(*name*, *object*)

Adds an object to this cell with the given name.

Args: name: (string) The name of the object.

object: (object) The object to add to the cell.

Raises: KeyError: The cell already has an object with that name.

Returns: `Cell`: The current instance (for method chaining)

`get_cell_coords`()

Gets the current position of the cell in the grid (not pixels)

Returns: tuple: The (x, y) coordinates

`get_objects`()

Returns the objects dictionary for this cell.

Returns: dict: The objects in this cell.

`get_position`()

Gets the current position of the cell (top-left corner)

Returns: tuple: The (x, y) coordinates

has_object(*name*)
Returns whether or not the cell contains an object with the specified name.

Returns: boolean: True if the cell has the named object.

is_blocked()
Returns whether the cell is considered blocked or not.

Returns: boolean: Whether the cell is considered blocked.

remove_object(*name*)
Removes the object with the given name from the cell.

Args: name: (string) The name of the object to remove.

Raises: KeyError: The object with that name is not found

Returns: object: The removed object

set_blocked(*send_to_map=True*)
Sets the cell to be blocked.

Args:

send_to_map: (bool) Whether or not to send the blocked data to the map to update the edge weights.
Should be True except if the cell is being created and set blocked before being added to the map (or, more specifically, before `Cell.set_cell_coords()` is called).

Returns: `Cell`: The current instance (for method chaining)

set_cell_coords(*x*, *y*)
Sets the grid indices for the current cell.

Args: x: (int) The x-coordinate in the grid (not pixels)

y: (int) The y-coordinate in the grid (not pixels)

Returns: `Cell`: The current instance (for method chaining)

set_position(*x*, *y*)
Sets the position of the cell (top-left corner)

Args: x: (int) The x-coordinate

y: (int) The y-coordinate

Returns: `Cell`: The current instance (for method chaining)

set_unblocked(*send_to_map=True*)
Sets the cell to be unblocked.

Args:

send_to_map: (bool) Whether or not to send the blocked data to the map to update the edge weights.
Should be True except if the cell is being created and set unblocked before being added to the map (or, more specifically, before `Cell.set_cell_coords()` is called).

Returns: `Cell`: The current instance (for method chaining)

class `wwgamelib.map.`**Map**(*game*, *width=20*, *height=15*, *size=32*, *cell_klass=<class 'wwgamelib.map.Cell'>*)
Inherits from `wwgamelib.layer.Layer`

The main game layer for the game.

Variables

- **`_width`** – (int) The width of the map (number of cells)
- **`_height`** – (int) The height of the map (number of cells)
- **`_size`** – (int) The width and height of the cell squares (pixels)
- **`_grid_width`** – (int) The width of the map (pixels)
- **`_grid_height`** – (int) The height of the map (pixels)
- **`_grid`** – (`networkx.Graph` from `networkx.generators.classic.grid_graph()`) The graph representation of the grid
- **`_grid_data`** – (dict) A map of (row,col) -> cell (cells hold the data for that position in the grid)
- **`_cell_class`** – The class to use for the cells. Defaults tp `wwgamelib.map.Cell`, but could straightforwardly be changed to any subclass.

Args: game: (`wwgamelib.Game`) The game object that is host to this layer. This is stored as a `weakref`.

width: (int) The width of the map (default 20)

height: (int) The height of the map (default 15)

size: (int) The width and height of a cell square (default 32)

cell_klass: The class to use for map cells

`_get_col_data`(*col*)
This returns the column data for the map as a list.

Args: col: (int) The column index

Returns: list: The list of cells in the row (in order)

`_get_row_data`(*row*)
This returns the row data for the map as a list.

Args: row: (int) The row index

Returns: list: The list of cells in the row (in order)

`_position`(*row, col*)
Determines the pixel position of a certain row-column grid square

Treats the grid as centered in the game window

Args: row: (int) The row index

col: (int) The column index

Returns: tuple: The pixel coordiates of the grid square

`_reset_cell`(*row, col*)
Totally resets a cell

Args: row: (int) the index of the row to reset

col: (int) the index of the column to reset

Returns: `Map`: The current instance (for method chaining)

`_reset_column`(*col*)
Totally resets a column of the grid

Args: col: (int) the index of the column to reset

Returns: `Map`: The current instance (for method chaining)

`_reset_grid`()
Totally resets the grid

Returns: `Map`: The current instance (for method chaining)

`_reset_row`(*row*)
Totally resets a row of the grid

Args: row: (int) the index of the row to reset

Returns: `Map`: The current instance (for method chaining)

`dump_visual`(*marks=None*, *mark_char='*'*)
This is a debug routine to print out a visual representation of the information stored in the map.

Args: marks: (list) A list of row-col pairs to put a special mark on

mark_char: (char) The character to use for the special mark

`get_cell_coords_from_pixel_coords`(*x*, *y*)
Gets the cell coordinates (row, col) from the pixels coordinates.

Args: x: (int) the x-coordinate (pixels)

y: (int) the y-coordinate (pixels)

Returns: tuple: The (row, col) cell coordinates or None if the pixels are out of bounds.

`set_blocked`(*blocked*, *row*, *col*)
Sets or unsets whether a cell is blocked by modifying the edge weights on the grid graph.

Args: blocked: (bool) Whether the cell is blocked or not.

row: (int) The row index of the cell

col: (int) The column index of the cell

Returns: `Map`: The current instance (for method chaining)

`shortest_path`(*from_pt*, *to_pt*)
Finds the shortest path between two cells (calls `wwgamelib.map.graph_shortest_path()`).

Uses North-South-East-West neighbors only

Args: from_pt: (tuple) The (x,y) pair (in the grid, not by pixels) of the starting point of the path

to_pt: (tuple) The (x,y) pair (in the grid, not by pixels) of the ending point of the path

Returns: list or None: The nodes in a shortest path, or None if no path is possible

`would_allow_path`(*from_pt*, *to_pt*, *new_block*)
Determines whether blocking a cell would still allow a path between two others without changing the actual map (uses the Dijkstra algorithm via `networkx`).

Uses NSEW neighbors only

Args: from_pt:

to_pt:

new_block:

Returns: boolean: Whether there is still a path or not `wwgamelib.map.`**`graph_shortest_path`**(*graph*, *from_pt*, *to_pt*)

Finds the shortest path between two cells

Uses the Dijkstra algorithm via `networkx.algorithms.shortest_paths.weighted.single_source_dijkstra`

Uses North-South-East-West neighbors only

Args: from_pt: (tuple) The (x,y) pair (in the grid, not by pixels) of the starting point of the path

to_pt: (tuple) The (x,y) pair (in the grid, not by pixels) of the ending point of the path

Returns: list or None: The nodes in a shortest path, or None if no path is possible

1.1.5 wwgamelib.menu module

The menu module contains two classes: `wwgamelib.menu.Menu` and `wwgamelib.menu.MenuButton`.

The `wwgamelib.menu.Menu` class is a `wwgamelib.layer.Layer` that handles placing several menu buttons. Classes derived from it will likely also implement a listener to determine when each button is clicked and what to do when each is.

:py:class:‘wwgamelib.menu.Menu‘s also have functionality to look up which button(s) attached to it overlaps a certain set of pixels.

The `wwgamelib.menu.MenuButton` class represents one of those buttons. It is a `wwgamelib.sprite.Sprite` (the background image) with an attached `wwgamelib.label.Label` (the button text). The label is not always necessary if it is desired to put the button text directly into the image asset. This module provides base classes for game menus and menu buttons.

The menus are `Layer` objects and the buttons are `Sprite` objects.

class `wwgamelib.menu.`**`Menu`**(*game*, *title*, *buttons*, *icon=None*)

Inherits from `wwgamelib.layer.Layer`

Abstract class for a menu.

Variables **`_buttons`** – (list) The names of the buttons for the menu

Args: game: (`wwgamelib.Game`) The game object that is host to this layer. This is stored as a `weakref`.

title: (string) The title of the menu. This is shown as a `wwgamelib.label.Label`.

buttons: (list of (name, `MenuButton`) tuples) The buttons that belong to the menu.

icon: (`wwgamelib.sprite.Sprite` or None) The logo/icon for the menu.

`button_lookup`(*x*, *y*)

Finds the name of one button from this menu that is under the given coordinates.

Note: If more than one button is at those coordinates, the one that is selected is at the mercy of python's dict iteration ordering.

Args: x: (int) The x-coordinate to look for y: (int) The y-coordinate to look for

Returns: string or None: The name of a button at those coordinates or None.

class `wwgamelib.menu.`**`MenuButton`**(*game*, *identifier*, *text*, *image*, *font_name='Arial'*, *font_size=20*)

Inherits from `wwgamelib.sprite.Sprite`

The base class for a menu button

Variables

- **`_identifier`** – The button's identifier

- **_label** – The button's label

Args: game: (wwgamelib.Game) The game object that is host to this layer. This is stored as a weakref.

identifier: (string) The name of the button

text: (string) The button's text, which is displayed in a Label

image: (pyglet.image.AbstractImage or pyglet.image.Animation or string) Either the image to set as the background or a path to the image asset.

font_name: (string) The name of the font to use for the button's text.

font_size: (int) The size of the button's text (in pt)

Raises: ValueError: A suitable image is not provided.

get_id()

Returns the button's identifier

Returns: string: The identifier of the button

set_label(*text*)

Sets the text of the button's label

Args: text: (string) The new text of the button

Returns: MenuButton: The current instance (for method chaining)

set_label_group(*group*)

Sets the label's render group.

Args: group: (pyglet.graphics.Group) The new base render group for the button's label.

Returns: MenuButton: The current instance (for method chaining)

1.1.6 wwgamelib.sprite module

This module provides a wrapper around pyglet.sprite.Sprite objects to enable various niceties.

class wwgamelib.sprite.**Sprite**(*game*, *image*, *x=0*, *y=0*, *batch=None*, *group=None*)

Inherits from GameObject and pyglet.sprite.Sprite

A wrapper for sprites that add some additional magic.

Variables **_attachments** – (list) The objects that have been attached to this sprite.

Args: game: (Game) The game object that the sprite belongs to. Stored as a weakref.

image: (pyglet.image.AbstractImage or pyglet.image.Animation or string) Either the image to use or a path to the image asset.

x: (int) The initial x-coordinate of the top-left corner

y: (int) The initial y-coordinate of the top-left corner

batch: (pyglet.graphics.Batch or None) The default render batch of the sprite.

group: (pyglet.graphics.Group or None) The default render group of the sprite.

Raises: ValueError: No suitable image is supplied.

attach(*obj*)

Attaches another sprite to this one. Attached sprites will move and be shown/hidden with this one.

Args: obj: The object to attach

Returns: `Sprite`: The current instance (for method chaining)

`contains` (*x*, *y*)
Determines if the sprite contains the coordinates x,y

Args: x: (int) The x-coordinate of the target point

y: (int) The y-coordinate of the target point

Returns: boolean: Whether the point (x,y) is within the bounds of the sprite.

`hide` ()
Sets the sprite to be invisible.

Returns: `Sprite`: The current instance (for method chaining)

`is_shown` ()
Returns whether or not the sprite is visible

Returns: boolean: Whether the object is visible or not

`set_attachment_positions_by_center` (*x*, *y*)
Sets the position of all attachments by their centers.

Args: x: (int) The new x-coordinate of the centers

y: (int) The new y-coordinate of the centers

Returns: `Sprite`: The current instance (for method chaining)

`set_batch` (*batch*)
Sets the sprite's render batch and also sets the batch of all the sprite's attachments.

Args: batch: (`pyglet.graphics.Batch`) The render batch to set

Returns: `Sprite`: The current instance (for method chaining)

`set_group` (*group*)
Sets the sprite's render group and also sets the group of all the sprite's attachments

Args: group: (`pyglet.graphics.Group`) The render group to set

Returns: `Sprite`: The current instance (for method chaining)

`set_position` (*x*, *y*)
Sets the position of this sprite and all attached objects.

Args: x: (int) The new x-coordinate

y: (int) The new y-coordinate

Returns: `Sprite`: The current instance (for method chaining)

`set_position_by_center` (*x*, *y*)
Sets the position of this sprite and all attached objects by specifying the center.

Args: x: (int) The new x-coordinate of the center

y: (int) The new y-coordinate of the center

Returns: `Sprite`: The current instance (for method chaining)

`show` (*batched=False*)
Sets the sprite to be visible

Args: batched: (bool) Whether the drawing is batched (default False). If False, then `pyglet.graphics.Sprite.draw()` is called on the current instance.

Returns: `Sprite`: The current instance (for method chaining)

1.2 Module contents

This is the base module for the Windward Game Library. It contains the basic `Game` class, as well as the `GameObject` class.

class `wwgamelib.`**`Game`**(*title='My Game'*, ***kwargs*)

Inherits from `object`

This class controls the game. It encapsulates a window object and controls scene switching.

Variables

- **`logger`** – (`logging.Logger`) The logging utility (instead of print statements)
- **`title`** – (string) The title of the game.
- **`window`** – (`pyglet.window.Window`) The window instance that the game runs in.
- **`_layers`** – (dict) A dictionary of layers. The keys are strings naming `Layer` objects.
- **`_current_layer`** – (string) A string naming the current layer to be shown. This should be a key of `Game._layers`

Args: title: (string) The name of the game. Defaults to "My Game".

assetpath: (string or None) a path to add to the `pyglet.resource.path` list.

Keyword Args: logger_name: (string) The namespace of the logger (default __name__, optional)

logger_level: (enum value) The logging level to display (one of logging.DEBUG, logging.INFO, logging.WARNING, logging.ERROR, logging.CRITICAL, default logging.WARNING, optional)

logger_handler: (`logging.Handler`) A handler for the log (defaults to a `logging.StreamHandler` outputting to sys.stdout; optional) See `logging.FileHandler` and other entries in the `logging.handlers` module

`_get_layer`(*target*)

Gets the layer specified by the key target.

Args: target: (string) The name of the layer to get.

Raises: ValueError

`_hide_all_layers_except`(*target*)

Hides all layers that aren't the target one (in case that is already shown).

Args: target: (string or None) The key for the layer not to hide. Does not check to ensure the layer actually exists.

`_show`(*target*)

Really shows the target layer. Called by an event handler on the `Game.window` property.

Args: target: (string) The name of the layer to show.

`push_handlers`(**args*, ***kwargs*)

Delegates handlers to the window object. Calls push_handlers on the `pyglet.window.Window` instance stored in the `Game.window` attribute.

quit()
Shuts down the game (calls `pyglet.app.exit()`).

run()
Actually starts the game (calls `pyglet.app.run()`).

show(*target*)
Directs the game to show the target layer.

Args: target: (string) The name for the layer to show.

Raises: ValueError

class wwgamelib.**GameObject**(*game*, **args*, ***kwargs*)
This is a base class for all game objects. It handles saving the parent game as a weakref (and nothing else).

Args: game: (`Game`) The game object to save a reference to. It is stored as a `weakref`.

CHAPTER 2

wwgamelib examples

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*

Python Module Index

Index