simplequi

Release 1.0.7

Arthur Gordon-Wright

Dec 17, 2019

CONTENTS:

This package implements the Codeskulptor.org simplegui API in a desktop environment, running using PySide2/Qt. See the *README* for installation and usage instructions.

CHAPTER

ONE

README

1.1 simplequi

v1.0.7

Run codeskulptor.org programs on the desktop using Qt/PySide2

To run an existing codeskulptor script on your local machine, simply import simplequi as simplegui:

```
import simplequi as simplegui

# The rest of your script goes here unchanged
```

Nothing else should need changing!

1.1.1 Features

- Runs codeskulptor.org Python3 scripts using a Qt application
- The API matches simplegui exactly, so you should be able to run your script exactly as on codeskulptor.org after importing simplequi

1.1.2 Installation

Get simplequi from pip:

```
pip install simplequi
```

Or checkout the source code from https://github.com/ArthurGW/simplequi, then run:

```
pip install -r requirements.txt
```

1.1.3 Examples

Included in simplequi/examples are various scripts to show simple usages.

After installing simplequi, these can be run for example like this:

```
python -m simplequi.examples.codeskulptor_default
```

1.1.4 Known Issues

- Only supports the simplegui part of the codeskulptor API.
  - Does not support simplemap, simpleplot or other support functions.
  - Support for simplemap and simpleplot is planned in future.
- Execution happens by the simplequi Qt application running when the Python interpreter is ready to shutdown
  - This can cause problems with some debuggers, but is fine for normal use.
  - Please report any issues you find with this!
- For now, only supports PySide2/qt-for-python
  - Support for PyQt will hopefully be added in future.

1.1.5 Contribute

- Issue Tracker: https://github.com/ArthurGW/simplequi/issues
- Source Code: https://github.com/ArthurGW/simplequi

1.1.6 Support

If you are having issues, please let us know. The maintainers can be contacted at simplequi.codeskulptor@gmail.com

1.1.7 License

The project is licensed under the GPLv3 license.

OpenSSL

The distribution includes a couple of OpenSSL DLLs, which are necessary for getting images and sounds from HTTPS urls. This encryption may not be allowed in your country, please check local laws. These DLLs may only work on Windows, so you may have to install OpenSSL yourself on other systems.

This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit. (http://www.openssl.org/).

See simplequi/resources/ssllib/LICENSE.txt for the full OpenSSL licence details.

CHAPTER

TWO

SIMPLEQUI USAGE

See the documents below for everything you can currently do with simplequi. *Codeskulptor simplegui API* lists everything in the simplequi module itself, i.e. anything run by simplequi.function, e.g. `simplequi.create_frame()`. *Classes* describes the usage of the classes returned by the simplequi functions, such as `Frame`, `Timer` etc.

2.1 Codeskulptor simplegui API

The functions below implement the basic methods used to interact with simplegui and create objects: frames, timers, images and sounds.

2.1.1 Create a Frame

simplequi.**create_frame**(*title*, *canvas_width*, *canvas_height*, *control_width=None*)

Creates a new frame for interactive programs.

The frame's window has the given title, a string. The frame consists of two parts: a control panel on the left and a canvas on the right. The control panel's width in pixels can be specified by the number control_width. The canvas width in pixels is the number canvas_width. The height in pixels of both the control panel and canvas is the number canvas_height.

Parameters

- **title** (str) – the title of the frame window
- **canvas_width** (int) – the width of the drawing area, in pixels
- **canvas_height** (int) – the height of the drawing area, in pixels
- **control_width** (Optional[int]) – the width of the control area of the frame, in pixels

Return type `Frame`

Returns a `Frame` that can be used to setup (most of) the rest of the program

2.1.2 Create a Timer

simplequi.**create_timer**(*interval*, *timer_handler*)
Creates a timer.

Once started, it will repeatedly call the given event handler at the specified interval, which is given in ms. The handler should be defined with no arguments.

Parameters

- **interval** (int) – the interval at which to call the timer handler, in milliseconds
- **timer_handler** (Callable[[], None]) – a function to call after each interval

Return type *Timer*

Returns a *Timer* that calls timer_handler every interval ms (once started)

2.1.3 Load an Image from a URL

simplequi.**load_image**(*url*)
Loads an image from the specified URL.

The image can be in any format supported by PySide2. No error is raised if the file isn't found or is of an unsupported format.

Parameters **url** (str) – the URL of the image to load

Return type *Image*

Returns an *Image* that can be passed to *draw_image()*

2.1.4 Load a Sound from a URL

simplequi.**load_sound**(*url*)
Loads a sound from the specified URL.

Supports whatever audio formats that PySide2 supports. No error is raised if the file isn't found or is of an unsupported format.

Parameters **url** (str) – the URL of the sound to load

Return type *Sound*

Returns a *Sound* that can be played, paused etc.

2.1.5 Lookup the Value for a Key Press

class simplequi._keys.**KeyMap**
The keyboard event handlers receive the relevant key as an integer.

Because different browsers can give different values for the same keystrokes, simplequi provides a way to get the appropriate key integer for a given meaning. The acceptable strings for character are the letters 'a'...'z' and A'...'Z', the digits '0'...'9', 'space', 'left', 'right', 'up', and 'down'. Note that other keyboard symbols are not defined in simplequi.KEY_MAP.

__getitem__(*key*)
x.__getitem__('y') <==> x['y']

Get the value that is sent to a handler for a given key press.

This is usually called using square brackets, like `value = simplequi.KEY_MAP['space']`

Parameters **key** (`str`) – the key press to look up

Return type `int`

Returns the integer value for the key

Raises **KeyError** – if the key is not in the map for simplegui

Example usage:

```
def key_handler(self, key):
    if key == simplequi.KEY_MAP['left']:
        print('Left key pressed!')
        self.move_ship_left()
```

2.2 Classes

These are the classes returned by simplegui functions. They implement the rest of the simplegui API. The methods on these are responsible for most of the fun stuff that simplegui can do!

2.2.1 Frame

class simplequi._frame.**Frame**(*title*, *canvas_width*, *canvas_height*, *control_width=None*)

Frame that contains all other graphical widgets

Parameters

- **title** (`str`) – the text to use in the frame title bar
- **canvas_width** (`int`) – the width of the drawing canvas part of the frame
- **canvas_height** (`int`) – the height of the drawing canvas part of the frame
- **control_width** (`Optional[int]`) – the optional width of the controls (buttons etc.) part of the frame

add_button(*text*, *button_handler*, *width=None*)

Adds a button to the frame's control panel with the given text label.

The width of the button defaults to fit the given text, but can be specified in pixels. If the provided width is less than that of the text, the text overflows the button. The handler should be defined with no parameters.

Parameters

- **text** (`str`) – the button text
- **button_handler** (`Callable[[], None]`) – a function to call when the button is clicked
- **width** (`Optional[int]`) – the optional button width

Return type *Control*

Returns a handle that can be used to get and set the button text

add_input(*text*, *input_handler*, *width*)

Adds a text input field to the control panel with the given text label.

The input field has the given width in pixels. The handler should be defined with one parameter. This parameter will receive a string of the text input when the user presses the Enter key.

Parameters

- **text** (str) – the text of the input field label
- **input_handler** (Callable[[str], None]) – a function that is called when the user presses enter in the text input field
- **width** (int) – the width of the input field

Return type *Control*

Returns a handle that can be used to get and set the input field text

add_label(*text*, *width=None*)

Adds a text label to the control panel.

The width of the label defaults to fit the width of the given text, but can be specified in pixels. If the provided width is less than that of the text, the text overflows the label.

Parameters

- **text** (str) – the label text
- **width** (Optional[int]) – the optional label width

Return type *Control*

Returns a handle that can be used to get and set the label text

static get_canvas_textwidth(*text*, *size*, *face*)

Given a text string, a font size, and a font face, this returns the width of the text in pixels.

It does not draw the text. This is useful in computing the position to draw text when you want it centered or right justified in some region. The supported font faces are the default 'serif', 'sans-serif', and 'monospace'.

Parameters

- **text** (str) – the text to measure
- **size** (int) – the font size that would be drawn with
- **face** (str) – the font face that would be drawn with

Return type int

Returns the width of the text in pixels

set_canvas_background(*colour*)

Changes the background colour of the frame's canvas, which defaults to black.

..seealso:: get_colour() defines the allowed colour definitions

Parameters **colour** (str) – the background colour to set, accepts any valid CSS colour

Return type None

set_draw_handler(*draw_handler*)

Adds an event handler that is responsible for all drawing.

The handler should be defined with one parameter. This parameter will receive a *Canvas* object.

Parameters **draw_handler** (Callable[[*Canvas*], None]) – function to call every 1/60:sup:*th* of a second (actually 17ms), which gets a *Canvas* object as an argument

Return type None

set_keydown_handler(*key_handler*)
Adds a keyboard event handler waiting for keydown event.

When any key is pressed, the keydown handler is called once. The handler should be defined with one parameter. This parameter will receive an integer representing a keyboard character.

Parameters **key_handler** (`Callable[[int], None]`) – a function to call when a key is pressed down and the frame is focused

Return type `None`

set_keyup_handler(*key_handler*)
Adds a keyboard event handler waiting for keyup event.

When any key is released, the keyup handler is called once. The handler should be defined with one parameter. This parameter will receive an integer representing a keyboard character.

Parameters **key_handler** (`Callable[[int], None]`) – a function to call when a key is released and the frame is focused

Return type `None`

set_mouseclick_handler(*mouse_handler*)
Adds a mouse event handler waiting for mouseclick event.

When a mouse button is clicked, i.e., pressed and released, the mouseclick handler is called once. The handler should be defined with one parameter. This parameter will receive a pair of screen coordinates, i.e., a tuple of two non-negative integers.

Parameters **mouse_handler** (`Callable[[tuple], None]`) – a function to call when the mouse is clicked

Return type `None`

set_mousedrag_handler(*mouse_handler*)
Adds a mouse event handler waiting for mousedrag event.

When a mouse is dragged while the mouse button is being pressed, the mousedrag handler is called for each new mouse position. The handler should be defined with one parameter. This parameter will receive a pair of screen coordinates, i.e., a tuple of two non-negative integers.

Parameters **mouse_handler** (`Callable[[Tuple[int, int]], None]`) – a function to call when the mouse is clicked and dragged

Return type `None`

start()
Commences event handling on the frame (actually on the canvas that handles the events)

2.2.2 Canvas

class simplequi._canvas.**Canvas**(*drawing_area*)
Wrapper for the drawing area, implementing the codeskulptor canvas API.

This class is passed to the `draw_handler` set by *set_draw_handler()* to allow calls to draw on the canvas to be made in draw events. These events occur roughly 60 times per second. Note that users should not create instances of this class, creation is handled internally.

Parameters **drawing_area** (`DrawingArea`) – the actual widget that will be drawn on

draw_arc(*center_point*, *radius*, *start_angle*, *end_angle*, *line_width*, *line_color*, *fill_color=None*)
Draws an arc at the given center point having the given radius.

The point is a 2-element tuple or list of screen coordinates. The starting and ending angles indicate which part of a circle should be drawn. Angles are given in radians, clockwise starting with a zero angle at the 3 o'clock position. The line's width is given in pixels and must be positive. The fill color defaults to None. If the fill color is specified, then the interior of the circle is colored.

Parameters

- **center_point** (`Tuple[int, int]`) – the center of the arc
- **radius** (`int`) – the radius of the arc
- **start_angle** (`float`) – the start angle of the arc
- **end_angle** (`float`) – the end angle of the arc
- **line_width** (`int`) – the line width to draw with
- **line_color** (`str`) – the line colour to draw with
- **fill_color** (`Optional[str]`) – the colour to fill the arc with (optional, defaults to transparent)

Return type `None`

draw_circle(*center_point*, *radius*, *line_width*, *line_color*, *fill_color=None*)

Draws a circle at the given center point having the given radius.

The point is a 2-element tuple or list of screen coordinates. The line's width is given in pixels and must be positive. The fill color defaults to None. If the fill color is specified, then the interior of the circle is colored.

Parameters

- **center_point** (`Tuple[int, int]`) – the center of the circle
- **radius** (`int`) – the radius of the circle
- **line_width** (`int`) – the line width to draw with
- **line_color** (`str`) – the line colour to draw with
- **fill_color** (`Optional[str]`) – the colour to fill the circle with, optional, defaults to transparent

Return type `None`

draw_image(*image*, *center_source*, *width_height_source*, *center_dest*, *width_height_dest*, *rotation=0.0*)

Draw an image that was previously loaded by simplequi.load_image.

center_source is a pair of coordinates giving the position of the center of the image, while center_dest is a pair of screen coordinates specifying where the center of the image should be drawn on the canvas. width_height_source is a pair of integers giving the size of the original image, while width_height_dest is a pair of integers giving the size of how the images should be drawn. The image can be rotated clockwise by rotation radians.

You can draw the whole image file or just part of it. The source information (center_source and width_height_source) specifies which pixels to display. If it attempts to use any pixels outside of the actual file size, then no image will be drawn.

Specifying a different width or height in the destination than in the source will rescale the image.

Parameters

- **image** (*Image*) – the *Image* to render

- **center_source** (Tuple[int, int]) – the center point of the portion of the original image to render
- **width_height_source** (Tuple[int, int]) – the width and height of the portion of the original image to render
- **center_dest** (Tuple[int, int]) – the location to render the image on the canvas
- **width_height_dest** (Tuple[int, int]) – the size to render the image on the canvas
- **rotation** (float) – amount in radians to rotate the image, with 0.0 at the 3 o'clock position, increasing clockwise

Return type None

draw_line(*point1*, *point2*, *line_width*, *line_color*)

Draws a line segment between the two points, each of which is a 2-element tuple or list of screen coordinates. The line's width is given in pixels and must be positive.

Parameters

- **point1** (Tuple[int, int]) – the coordinate of the start point of the line
- **point2** (Tuple[int, int]) – the coordinate of the end point of the line
- **line_width** (int) – the width of the line to draw
- **line_color** (str) – the colour of the line to draw

Return type None

draw_point(*point*, *color*)

Draws a 1×1 rectangle at the given point in the given color. The point is a 2-element tuple or list of screen coordinates.

Parameters

- **point** (Tuple[int, int]) – the coordinates of the point
- **color** (str) – the colour to draw the point

Return type None

draw_polygon(*point_list*, *line_width*, *line_color*, *fill_color=None*)

Draws a sequence of line segments between each adjacent pair of points in the non-empty list, plus a line segment between the first and last points.

It is an error for the list of points to be empty. Each point is a 2-element tuple or list of screen coordinates. The line's width is given in pixels, and must be positive. The fill color defaults to None. If the fill color is specified, then the interior of the polygon is colored.

Parameters

- **point_list** (Iterable[Tuple[int, int]]) – the coordinates of the polygon's vertices (the final point is always joined to the first)
- **line_width** (int) – the line width to draw with
- **line_color** (str) – the line colour to draw with
- **fill_color** (Optional[str]) – the colour to fill the polygon with, optional, defaults to transparent

Return type None

draw_polyline(*point_list*, *line_width*, *line_color*)
Draws a sequence of line segments between each adjacent pair of points in the non-empty list.

It is an error for the list of points to be empty. Each point is a 2-element tuple or list of screen coordinates. The line's width is given in pixels and must be positive.

Parameters

- **point_list** (Iterable[Tuple[int, int]]) – the coordinates of the line's segment start/finish points, in order
- **line_width** (int) – the line width to draw with
- **line_color** (str) – the line colour to draw with

Return type None

draw_text(*text*, *point*, *font_size*, *font_color*, *font_face='serif'*)
Writes the given text string in the given font size, color, and font face.

The point is a 2-element tuple or list of screen coordinates representing the lower-left-hand corner of where to write the text. The supported font faces are 'serif' (the default), 'sans-serif', and 'monospace'.

Parameters

- **text** (str) – the text to draw
- **point** (Tuple[int, int]) – the point to draw at
- **font_size** (int) – the font size of the text
- **font_color** (str) – the colour of the text
- **font_face** (str) – the font face of the text (one of *serif*, *sans-serif* or *monospace*)

Return type None

2.2.3 Control

class simplequi._widgets.**Control**(*widget*)
A control that lives in the control area of a *Frame*, and allows getting and setting of its display text.

Parameters **widget** (QWidget) – the widget that this control wraps

get_text()
Returns the text in a label, the text label of a button, or the text in the input field of a text input.

For an input field, this is useful to look at the contents of the input field before the user presses Enter.

Return type str

Returns the current text of the control

set_text(*text*)
Changes the text in a label, the text label of a button, or the text in the input field of a text input.

For a button, it also resizes the button if the button wasn't created with a particular width. For an input field, this is useful to provide a default input for the input field.

Parameters **text** (str) – the new text to set

Return type None

2.2.4 Timer

class simplequi._timer.**Timer**(*interval*, *timer_handler*)
Creates a timer.

Once started, it will repeatedly call the given event handler at the specified interval, which is given in milliseconds. The handler should be defined with no arguments.

Parameters

- **interval** (int) – how often to call the timer_handler
- **timer_handler** (Callable[[], None]) – a function to call every interval milliseconds

is_running()
Returns whether the timer is running, i.e., it has been started, but not stopped.

Return type bool

start()
Starts or restarts the timer.

stop()
Stops the timer. It can be restarted.

2.2.5 Image

class simplequi._image.**Image**(*url*)
Loads an image from the specified URL.

The image can be in any format supported by PySide2. No error is raised if the file can't be loaded for any reason, it will simply not draw when asked to.

Parameters **url** (str) – the URL to load the image from, can be a local file

get_height()
Returns the height of the image in pixels. While the image is still loading, it returns zero.

get_width()
Returns the width of the image in pixels. While the image is still loading, it returns zero.

2.2.6 Sound

class simplequi._sound.**Sound**(*url*)
Loads a sound from the specified URL.

Supports whatever audio formats that PySide2 supports (depending on locally-installed codecs). No error is raised if the file isn't found or is of an unsupported format.

Sounds are tracked

Parameters **url** (str) – the URL to load the sound from, can be a local file

pause()
Stops the playing of the sound. Playing can be restarted at the stopped point with *play()*.

play()
Starts playing a sound, or restarts playing it at the point it was paused.

rewind()
Stops playing the sound, makes it so the next *play()* will start playing the sound at the beginning.

set_volume(*volume*)
Changes the volume for the sound to be the given level on a 0 (silent) – 1.0 (maximum) scale. Default is 1.

Parameters **volume** (float) – the volume to set

Return type None

CHAPTER

THREE

INDEX

- genindex

INDEX

Symbols

A

C

D

F

G

I

K

L

P

R

S

T