sgl Documentation

Release 0.0.1

m48

August 02, 2016

Contents

Welcome to the documentation for SGL! Soon, this will actually contain, like, useful things.

Until then, please look over the source code to figure out what's going on. I think I've made it logical enough to understand. There's going to be a lot of this going to change, though. Just warning you.

Contents:

CHAPTER 1

Installing

One of these days, you should be able to install SGL by going:

```
pip install sgl
```

We're not quite there yet, though.

1.1 Other methods of installing

For the record, no, I am not expecting random members of the public actually go through this amount of effort to test a crappy, half completed game development framework. This is mostly here because I'm currently forcing some friends to set the framework for me in an early stage of development, and want them to have some good instructions to follow to do so.

1.1.1 Install in pip's developer mode

1. First, download the repository in its current state. You can either do this with the git client, or the lazy way–by downloading the zip file Github provides.
2. Open a command line window, and navigate to the root folder of the repository.
3. Run the following command:

```
    pip install -e .
```

4. Go into the examples folder, and run `sgl-core-demo/demo.py` to make sure SGL is working correctly.

This will make pip add a temporary link to the folder where SGL is located to your `Python/Lib/site-packages` directory. If you look in there after doing this, you should find a `sgl.egg-link` file that contains the path to the SGL folder.

So, make sure to be careful moving around your SGL folder after this–if you move it without updating this file, SGL will stop in your programs working.

When I update SGL, clear out the contents of this folder, and replace it with the contents of the latest Github zip file of my repository.

When I finally get sgl on the real Python package directory, you will want to uninstall this local copy. To do this, just run the following command in any folder:

```
pip uninstall sgl
```

Then reinstall it with the real pip command at the top of the page.

1.1.2 Don't install it at all

If you're really lazy, download the Github zip file of my repository., and then copy and paste the `sgl` folder into any folder in which you want to have access to sgl.

So, for example, to make the example work, you would copy the `sgl` folder into the `examples/sgl-core-demo/` folder, so you would end up with a `examples/sgl-core-demo/sgl/` folder. If you have all the dependencies installed correctly, the example should work like that.

This is, however, a bit inconvenient–you will have to replace this SGL folder whenever it is upgraded. This may be what you're after, though–this early in development, SGL may change enough that your programs may depend on specific versions, and it may be convenient to be able to use multiple versions simultaneously without dealing with virtualenvs for a package that isn't even published anywhere.

1.2 Dependencies

This should be automatically managed, but things may get screwy, so I will say it explicitly.

SGL depends on PyGame 1.9.0+ to work correctly. It may work on earlier versions, but I have not tested this, and some advanced features may behave unexpectedly.

Some functions of SGL depend on NumPy being installed. I think nearly any version will work fine.

To render videos from SGL, you will need to install MoviePy. In my experience, on Windows, this is a fairly smooth, but annoying and long process. You have been warned. Installing MoviePy is not necessary for normal SGL development, though.

To the best of my knowledge, SGL only will work on Python 2.7. I am pretty sure I have done some stuff that will make it broken on Python 3, like using the old-style of print statements. You're welcome to give it a shot on Python 3, though.

Install these dependencies however you wish. Some of them offer Windows installers–if they do, prefer these. At the moment, this usually works a bit more smoothly than using pip. I know some libraries, such as MoviePy, do not support this, though, and others are slowly moving away from it, so you might have no choice but to use pip in some cases.

CHAPTER 2

Examples

Currently there is only one extremely pathetic example.

2.1 SGL Core Demo

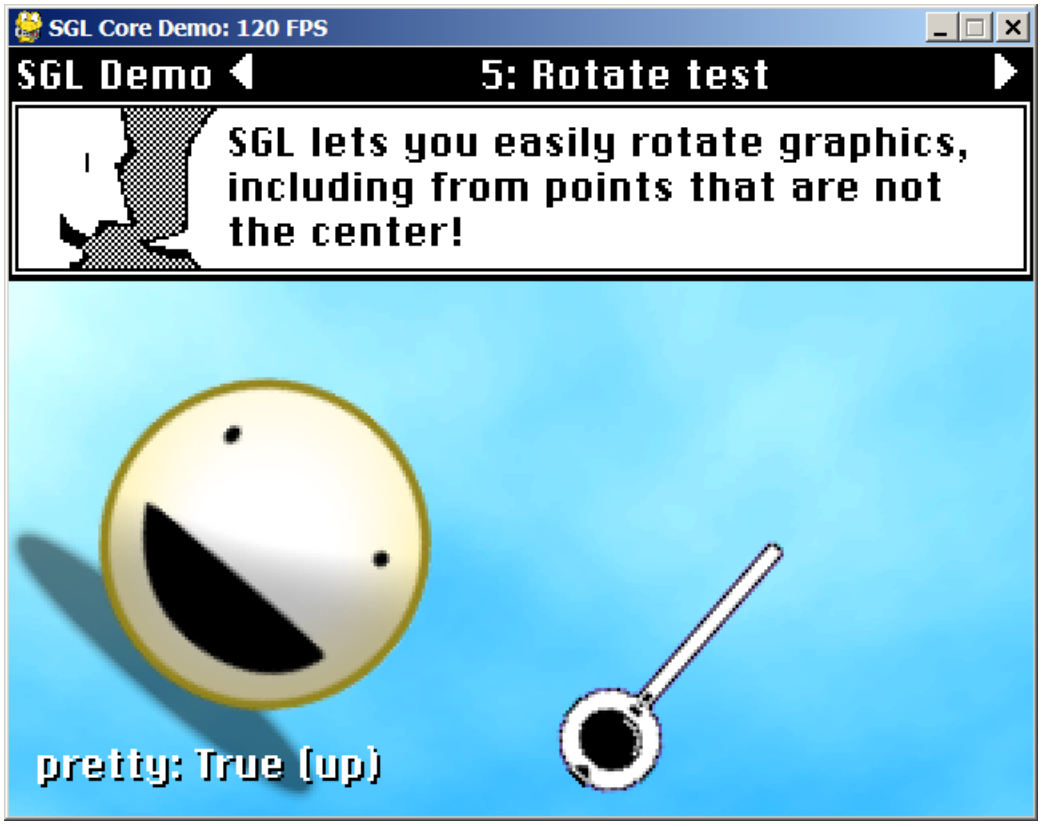

Fig. 2.1: SGL demonstrates its incredible graphic effects.

This is a demo for various features in `sgl.core`. It is used internally for me to develop new features and make sure they are working correctly. Anything that does not appear in this demo has not been tested.

This means, for example, that I am still not sure if any of the audio commands work or not.

To navigate through the demos, press the left and right arrow keys. Most of the demos what you alter their parameters by pressing the up and down arrow keys.

The coding style of this demo is not necessarily indicative of what a normal SGL game would look like, as it intentionally avoids using any functions from `sgl.lib`. So, it manages things manually that usually SGL would handle for you.

Also, it runs a series of separate demos as different classes, which are pretty much each separate SGL games, which a normal SGL game would have no need to do.

Hopefully even this complicated example showcases SGL's ability to be simple and easy to understand, though.

CHAPTER 3

SGL Core Reference

SGL divided into two components–`sgl.lib` and `sgl.core`. `sgl.core` is the part of the library that provides the low-level drawing commands, similar to what you might find built into BlitzBasic or Processing. While ideally you should be able to spend most of your time in `sgl.lib`, it is essential to use `sgl.core` to make even the simplest SGL program.

To import `sgl.core`, currently this is all that is required:

```
import sgl
```

Then, to call any of the functions defined in `sgl.core`, simply prefix them with `sgl.`, such as this:

```
sgl.init(640, 480)
graphic = sgl.load_image("smiley.png")
```

Warning: This may change in the future. I may make it so to import `sgl.core`, you must do it like this:

```
import sgl.core as sgl
```

The benefits of this approach:

- It will make the internal structure of the library easier to maintain
- It will make it more obvious that `sgl.core` is actually called `sgl.core`.

The detriments of this approach:

- It is more typing for little benefit for the end-user.
- The user must explicitly rename `sgl.core` to `sgl` to keep their command invocations short.

I still haven't made up my mind which approach to take. Keep in mind you may have to change your import statements later. The command invocation will remain the same regardless, though–it is just the importing syntax that may change.

3.1 Concepts

3.1.1 Philosophy

Most of `sgl.core` is fairly simple–it is a non-object-oriented module with various functions. This may irritate people who like object-oriented programming, and does create some clumsiness. For example, to get the width of a drawing surface, you cannot do something like this:

```
graphic = sgl.load_image("smiley.png")
print(graphic.width)
```

You must instead do this:

```
graphic = sgl.load_image("smiley.png")
with sgl.with_buffer(graphic):
   print(sgl.get_width())
```

People coming to SGL from non-object-oriented languages, like BlitzBasic, may be used to this, but this will undoubtedly bother Python programmers. I assure you, however, this is an intentional choice. In SGL, you have the freedom to choose how you want to program your game–whether you want to use object-oriented programming or not, and even how you organize the objects and classes of your game. `sgl.lib` is but one approach of organizing classes to wrap the internals of SGL. You can easily make your own if you wish, and I encourage you to do so.

I feel like existing Python game development frameworks are a little too dogmatic in how they force a certain structure on your programs, and I want to avoid that at all costs, even if my approach is a bit excessive. If you can wrap your head around this philosophy, most of `sgl.core` should make sense to you.

3.1.2 Fake types

In the function reference, you may see references to types such as "SGL Surface" or "SGL Sound." These are not real types, however–whenever a function claims to accept or return values of these types, in reality, they are dealing with *classes of whatever the current backend is.* So, if you're using the Pygame backend, and you examine the type of an "SGL Surface", you will find that it is, in reality, a pygame.Surface.

So, hypothetically, there's nothing stopping you from calling `pygame.Surface` commands on an SGL surface, like this:

```
graphic = sgl.load_image("smiley.png")
graphic.set_at((3, 3), (255, 0, 0))
```

Please don't do this, though. For one, this will obviously not work when different backends are introduced. And also, I may eventually actually wrap SGL Surfaces and such in classes like `sglSurface` or something, which will make your code broken on *all* backends.

Ideally, your code should be completely ignorant of what backend SGL is currently using. This'll make your game/program more portable, which is one of the main points of SGL existing.

3.1.3 Color arguments

Color arguments in SGL are kind of ridiculously flexible, and take some influence from how Processing works.

Whenever you see an argument named `*colors`, it can accept the values in the following formats:

- `number` - If you specify a single value as a color, it will specify shade of gray with that value used for the R, G, and B values of that color.
- `number, number` - If you specify to values as a color, the first number will specify the shade of gray, as it would with a single number, but the second number will specify the *transparency* of that shade of gray.
- `number, number, number` - If you specify three values, it will be interpreted as a normal RGB color.
- `number, number, number, number` - If you specify for values, it will be interpreted as a normal RGBA color.
- `tuple/list` - If you specify color as a tuple or list, it will be unpacked and interpreted as the arguments would.

In addition, for each number, the following rules apply:

- If the number is an integer, it will be interpreted as a normal RGB value that ranges from 0 to 255.

- If any numbers outside of this range, it will be clamped within that range.
- If the number is a float, it will be interpreted as a *percentage* of 255. So, for example, 0.5 would be half of 255.

This is perhaps a little complicated, but allows you to do some very convenient things. For example, if you want to clear the screen with 50% gray, it is as simple as doing this:

```
sgl.clear(0.5)
```

If you want to make a surface that is black and 75% transparent, you can do this:

```
surface = sgl.make_surface(320, 32, 0.0, 0.75)
```

And, if you feel it is confusing that the color data is getting mixed in with the rest of their arguments, you can pass in tuples as well:

```
surface = sgl.make_surface(320, 32, (0.0, 0.75))
```

And, of course, you can use plain RGB values for everything:

```
sgl.clear(100, 175, 93)
```

3.2 Function reference

This is a reference of every single function in `sgl.core`. It is not complete yet, but all of the most essential functions have been documented in some detail. Hopefully this is helpful.

3.2.1 Base functions

These functions have to do with the base functioning of your program, and are essential for nearly everything you'd want to do with SGL.

`sgl.`**`init`**(*width*, *height*, *scale=1*, *fullscreen=False*, *backend="pygame"*)
Must be called before any other SGL functions to initiate the drawing surface.

Parameters

- **`width, height`** (*int*) – Specifies the size of the display surface.
- **`scale`** (*int*) – Specifies the scaling factor. Width and height will be multiplied by this to form the actual window size. This is useful for working with small resolutions, such as 320x240.
- **`fullscreen`** (*bool*) – On desktop platforms, whether the display

should be windowed or fullscreen.

Parameters **`backend`** (*str*) – Which backend to use. Currently, the only supported option is "pygame".

`sgl.`**`run`**(*update=None*, *draw=None*)
Starts an SGL program that automatically manages the event loop. SGL will call the specified callbacks every frame, and it is in these that you will specify your game logic.

Parameters

- **`update`** (*function*) – A function to call every frame to update your game's state.

- **draw** (*function*) – A function to call every frame to draw the current frame. Do not put any game logic in this function. No backends currently do this, but in the future, some might take advantage of the difference between update and draw to, for example, pause the game by calling update but not draw.

sgl.**make_movie**(*file=""*, *update=None*, *draw=None*, *duration=0*, *fps=24*, *realtime=False*, *display=True*, ***extra*)

Similar to *sgl.run()*, but uses MoviePy to render each frame of your program to a video file or animated GIF.

Parameters

- **file** (*str*) – The filename of the movie you wish to output.
- **update, draw** (*function*) – Works the same as with *sgl.run()*.
- **duration** (*number*) – The amount of your program's execution you want to record, in seconds.
- **fps** (*number*) – What the frame rate of the resulting video will be. `make_movie` will force your program's frame rate to match with *sgl.set_fps_limit()*.
- **realtime** (*boolean*) – If the video renders faster than the frame rate you specify, this will slow down the speed of the video rendering to match. This is useful when you have "display" enabled and you want to, say, use your program while recording it. By default it is false.
- **display** (*boolean*) – Whether to draw the current frame to the screen, in addition to rendering into the video file. Useful if you want to interact with your program as it is rendering to video.
- **extra** (*various*) – Additional keyword arguments are passed to MoviePy's `write_videofile` or `write_gif` functions. You can use this to specify additional settings about which codecs to use and such.

sgl.**set_fps_limit**(*fps*)

Sets the highest frame rate your program will allow. More specifically, it makes your program assume that the frame rate is *always* this value, as opposed to what it is in reality. This can make programming animation more convenient, but comes with some side effects–such as that, if the frame rate of your program drops, it will start running in slow motion.

Parameters **fps** (*int*) – The desired framerate. Set to 0 to disable framerate limiting.

sgl.**get_fps_limit**()

Gets the framerate limit. If there is no framerate limiting, will return 0.

Returns The framerate limit

Return type int

sgl.**get_fps**()

Gets the current framerate. This will return the *actual* frame rate, even if the limit has been set with *sgl.set_fps_limit()*.

Returns The current framerate

Return type float

sgl.**get_scale**()

Gets the window's scaling factor. SGL automatically takes into account the scaling factor when returning, say, mouse coordinates, so there shouldn't be any reason to call this–it's just for completeness sake.

Currently, is no setting the scaling factor–if you want to make a program in which the scaling factor appears to change, use surfaces and end the scaling abilities of *sgl.blit()* to simulate this. This functionality may be added in the future, though.

Returns The current scaling factor

Return type int

sgl.**has**(*ability*)
Returns whether the current backend has a given ability. The abilities are specified in the enum-like class sgl.abilities. There are currently only four abilities you can test for:

- `sgl.abilities.software`: Whether the current backend is powered by software rendering, and thus will be slow at special effects such as rotating and scaling.
- `sgl.abilities.numpy`: Whether the current backend supports exporting and importing surfaces to NumPy arrays.
- `sgl.abilities.save_buffer`: Whether the current backend supports saving surfaces to image files.

The Pygame backend supports all of these.

Parameters **ability** (*int*) – The ability to test

Returns Whether the current backend supports this ability

Return type bool

sgl.**set_title**(*title*)
Sets the text in the title bar of the current window.

Parameters **title** (*str*) – The text to put in the title bar

sgl.**get_title**(*title*)
Gets the text currently in the title bar of the current window.

Returns The text in the title bar

Return type str

sgl.**get_actual_screen_width**()
Gets the width of the current window after applying the scaling factor. You should not need to get this. I should probably get rid of this function.

Returns The window width

Return type int

sgl.**get_actual_screen_height**()
Gets the height of the current window after applying the scaling factor. You should not need to get this. I should probably get rid of this function.

Returns The window height

Return type int

sgl.**get_dt**()
Gets *delta time,* or the time that has passed since the last frame is been rendered. If you do not have framerate limiting enabled for your program, you must multiply every animation value by this value if you expect your program to work consistently on different types of computers.

Unlike some other game libraries, this value is returned in *seconds,* not milliseconds.

Returns The time that has passed since the last frame in seconds

Return type float

sgl.**is_running**()
Returns whether your program is running. Useful for when you are manually managing the event loop, and you want your program to end under the same conditions an automatic SGL program would.

Returns Whether your program is running

Return type Bool

sgl.**end**()
Halts execution of the program, and closes the window.

3.2.2 Drawing commands

These commands have to do with drawing shapes.

sgl.**set_smooth**(*smooth*)
Sets whether lines for shapes should be anti-aliased or not. Currently no backend supports this.

Parameters **smooth** (*bool*) – Whether anti-aliasing should be enabled

sgl.**get_smooth**()
Returns whether anti-aliasing is enabled or not.

Returns Whether anti-aliasing is enabled

Return type bool

sgl.**set_fill**(**color*)
Sets the color with which shapes are filled. Also affects what color fonts are rendered in.

sgl.**get_fill**()
Gets the current fill color.

Returns The current fill color, or `None` if fill is disabled

Return type tuple

sgl.**set_stroke**(**color*)
Sets the color in which shapes are outlined.

sgl.**get_stroke**()
Gets the current stroke color.

Returns The current stroke color, or `None` if stroke is disabled

Return type tuple

sgl.**set_stroke_weight**(*weight*)
Sets how thick the lines outlining shapes will be. Setting this to 0 will disable stroke rendering.

sgl.**get_stroke_weight**()
Gets the current stroke weight.

Returns The current fill weight

Return type int

sgl.**no_stroke**()
Turns off stroke rendering.

sgl.**no_fill**()
Turns off fill rendering.

sgl.**push**()
Pushes the current graphics state to the stack. So, the next time you call *sgl.pop*, it will restore this state. You can use this to change the drawing colors and what current surface is, and reset them later.

sgl.**pop**()
Pops the current graphics state from the stack. This will make all the drawing settings return to what they were the last time *sgl.push* was called.

sgl.**with_state**()
A context manager that saves the current drawing state and restores it when the enclosed operations are finished. Useful to avoid manually having to call *sgl.push* and *sgl.pop*.

sgl.**clear**(**color*)
Completely fills the current surface with the specified color.

sgl.**draw_line**(*x1*, *y1*, *x2*, *y2*)
Draw the line between the specified coordinates.

sgl.**draw_rect**(*x*, *y*, *width*, *height*)
Draws a rectangle in the specified area.

sgl.**draw_ellipse**(*x*, *y*, *width*, *height*, *from_center=False*)
Draws an ellipse in the specified area.

sgl.**draw_circle**(*x*, *y*, *radius*, *from_center=True*)
Draws a circle in the specified area.

3.2.3 Text commands

These commands have to do with rendering text.

sgl.**set_font_smooth**(*smooth*)
Specifies whether text is anti-aliased or not.

sgl.**get_font_smooth**()
Returns whether text is anti-aliased or not.

Returns Whether font antialiasing is enabled

Return type bool

sgl.**load_font**(*file*, *size*)
Loads a font from a font file in your program folder.

Returns The loaded font

Return type SGL font object

sgl.**load_system_font**(*font_name*, *size*)
Loads a font from the user's system via the font's name.

Returns The loaded font

Return type SGL font object

sgl.**set_font**(*font*)
Sets what font is used for all future drawing operations.

sgl.**draw_text**(*text*, *x*, *y*)
Draws the specified text at the specified coordinates.

sgl.**get_text_width**(*text*)
Gets how wide the specified string will be when rendered in the current font.

Returns The width of the rendered text

Return type int

`sgl.`**`get_text_height`**(*text=""*)
Gets how tall the specified string will be when rendered in the current font. Usually will just return the height of the current font.

Returns The height of the text

Return type int

3.2.4 Image commands

These commands have to do with loading images and stuff.

`sgl.`**`set_transparent_color`**(**color*)
Sets what color will be considered transparent in images without an alpha channel. By default, this color is set to (255, 0, 255), or "magic magenta."

`sgl.`**`load_image`**(*file*, *use_transparent_color=True*)
Loads an image without an alpha channel from the hard drive.

Returns A surface containing the image loaded

Return type SGL surface

`sgl.`**`load_alpha_image`**(*file*)
Loads an image with an alpha channel from the hard drive.

Returns A surface containing the image loaded

Return type SGL surface

3.2.5 Surface commands

These commands have to do with using surfaces and changing the current drawing buffer.

`sgl.`**`blit`**(*thing*, *x*, *y*, *alpha=255*, *flip_v=False*, *flip_h=False*, *angle=0*, *width=None*, *height=None*, *scale=1*, *a_x=0*, *a_y=0*, *src_x=0*, *src_y=0*, *src_width=None*, *src_height=None*, *blend_mode=0*, *pretty=False*)
Draws one surface to another. This function is kind of the motherlode of SGL, and provides most of the library's drawing functionality.

Since this function takes so many arguments, it is recommended you use keyword arguments for everything beyond `x` and `y`. For example, a typical call to this function might look like this:

```
sgl.blit(self.player, 16, 16, flip_h=True, angle=45, a_x=0.5, a_y=0.5)
```

Parameters

- **`thing`** (*`SGL surface`*) – The thing that should be drawn (or blitted) to the current surface.

  Is an SGL surface value.

- **`x, y`** (*`int/float`*) – Specifies the coordinates on the current surface that `thing` should be drawn.

  Currently, passing in floats will just result in them being rounded to integers.

- **alpha** (*int/float*) – A number specifying how transparent `thing` should be drawn.

  If it is an integer, this should be a value between 0 and 255, with 0 being invisible and 255 being completely opaque.

  If it is a float, this should be a value between 0.0 and 1.0, with zero being invisible and 1.0 being completely opaque.

- **flip_v, flip_h** (*bool*) – Specifies whether `thing` should be horizontally or vertically flipped.

  Setting `width` or `height` to negative values will also flip a graphic.

- **angle** (*int*) – The angle to which `thing` should be rotated. The angle should be in degrees, and should be a value between 0 to 360 (although the code currently does not check this. It probably should).

  Note that on software renderers, such as Pygame, rotation is fairly slow. You can rotate a few graphics, such as the player graphic, in real time, but it is recommended you cache rotated sprites if you plan to rotate many things at once.

- **width, height** (*int*) – Specifies the width and height to resize `thing` to.

  If either of these values is not specified, it will be filled in with the original width/height of `thing`.

  If either of these values is 0, `thing` will not be drawn.

  If either is negative, `thing` will be drawn backwards in whatever direction resizing it takes.

- **scale** (*float*) – Enables you to scale `thing` by a ratio, keeping its aspect ratio. 1.0, the default, will draw `thing` at its normal size, 0.5 will draw it half as big, and so on.

  This can be combined with width and height, and is applied after those values are calculated.

- **a_x, a_y** (*int/float*) – Specifies the anchor point of `thing`–the coordinates of the graphic that is considered (0, 0). Useful in conjunction with `angle` to rotate about different points of the graphic than the top left corner.

  If these values are integers, they will be interpreted as exact coordinates on `thing`.

  If these values are floats, they will be interpreted as a percentage of the size of `thing`. (As in, setting both anchor points to 0.5 will make the anchor point the center of the image.)

- **src_x, src_y, src_width, src_height** (*int*) – Makes the function apply to the specified rectangle of `thing`. This is applied before the other functions are. This is a convenience to avoid having to do *sgl.get_chunk* whatever you want to draw a small chunk of a larger image.

  On hardware accelerated backends, this may be faster than using *sgl.get_chunk*.

- **blend_mode** (*int*) – Specifies the blending mode to use when drawing `thing`. Should be a constant from `sgl.blend`.

  Currently, the only available blending values are:

  – `sgl.blend.add` - Adds the colors of `thing`'s pixels to the pixels behind it. So, black pixels will become transparent, white pixels will stay white, and everything in between will make the image behind slightly brighter.

  – `sgl.blend.subtract` - Subtracts the colors of `thing`'s pixels from the pixels behind it. So, black pixels will become transparent, and white pixels will invert the background to certain degree.

  - `sgl.blend.multiply` - Multiply the colors of `thing`'s pixels from the pixels behind it. So, white pixels will become transparent, black pixels will stay black, and everything else will be make the image behind slightly darker.

- **pretty** (*boolean*) – Specifies whether the results of scaling and/or rotating `thing` should be smoothed out or not.

  This will slow down rendering in most cases.

`sgl.`**`blitf`**(*thing*, *x*, *y*)
A lightweight version of *sgl.blit* that does not perform any special effects on the surface being drawn. May be slightly faster.

Parameters

- **thing** (*SGL surface*) – The thing that should be drawn to the current surface.
- **x, y** (*int*) – Specifies the accordance on the current surface that `thing` should be drawn.

`sgl.`**`make_surface`**(*width*, *height*, **color*)
Makes a new blank surface of the specified color. If no color specified, the new surface will be blank and transparent.

Returns The created surface

Return type SGL surface

`sgl.`**`get_chunk`**(*x*, *y*, *width*, *height*)
Takes a chunk out of the current surface and returns it as a new copy surface. Useful for separating out, say, individual frames from spritesheets.

Parameters

- **x, y** (*int*) – The coordinates to start extracting
- **width, height** (*int*) – The width and height of the rectangle to extract

Returns The extracted surface

Return type SGL surface

`sgl.`**`set_clip_rect`**(*x*, *y*, *width*, *height*)
Sets the clipping rectangle–makes it so all future rendering operations will only affect the specified rectangle of the current surface.

`sgl.`**`get_clip_rect`**()
Returns the size of the current clipping rectangle.

Returns A four element tuple, or None

Return type tuple

`sgl.`**`no_clip_rect`**()
Turns off rendering clipping.

`sgl.`**`set_buffer`**(*surface*)
Sets which surface is the *drawing buffer*–the surface on which all future drawing operations will take place. You can also think of this as the "current surface," and many parts of the documentation refer to it like this).

Parameters **surface** (*SGL surface*) – The surface that will become the current surface

`sgl.`**`reset_buffer`**()
Sets the drawing buffer to refer to the screen buffer. If you manage the stack correctly, you shouldn't need to use this, but this is here just in case.

sgl.**with_buffer**(*buffer*)
A context manager that makes all enclosed drawing operations apply to a given buffer. Useful to avoid manually dealing with the stack.

Parameters **buffer** (*SGL buffer*) – The buffer to draw on

sgl.**get_width**()
Gets the width of the current surface.

Returns The width of the current surface, in pixels

Return type int

sgl.**get_height**()
Gets the height of the current surface.

Returns The height of the current surface, in pixels

Return type int

sgl.**save_image**(*file*)
Saves the image in the current surface to the specified filename.

Parameters **file** (*str*) – The filename of the image to save to. The extension will determine the file type.

sgl.**to_numpy**()
Exports the current surface as a NumPy array. On some back ends, such as Pygame, this might return a "live" NumPy array that is linked the original surface–as in changing the array will instantly change the surface. This is much more efficient than the alternative, but can be unexpected.

Returns The pixel data of the current surface, in a three-dimensional array

Return type NumPy array

sgl.**from_numpy**(*array*)
Creates a new surface from a NumPy array.

Parameters **array** (*NumPy array*) – A three-dimensional array consisting of RGB values.

Returns The surface represented by the array

Return type SGL surface

3.2.6 Special Effect commands

These commands do cool special effects.

sgl.**invert**(*surface=None*)
Inverts the colors of either the current surface or whatever surface is passed in.

Returns A new surface, with the effect applied, or nothing

Return type SGL surface

sgl.**grayscale**(*surface=None*)
Turns to grayscale either the current surface or whatever surface is passed in.

Returns A new surface, with the effect applied, or nothing

Return type SGL surface

3.2.7 Audio functions

These functions have to do with playing music and sound effects.

sgl.**load_sound**(*file*)
Loads a sound file from the hard drive.

Returns An object representing the loaded sound

Return type SGL sound

sgl.**play_sound**(*sound*, *volume=1.0*, *loops=0*)
Plays a previously loaded sound.

Parameters

- **sound** (*SGL sound*) – The sound object to play
- **volume** (*float*) – The volume to play the sound at, specified as a float. (So, 1.0 would be full volume, 0.5 would be half volume, and so on.)
- **loops** (*int*) – How many times to loop playback of the sound. If this is 0, the default, the sound will only play once. If it is -1, it will play forever, until *sgl.stop_sound* is called on it.

sgl.**stop_sound**(*sound*)
Stops the specified sound.

Parameters **sound** (*SGL sound*) – The sound object to stop

sgl.**stop_all_sounds**()
Stops all currently playing sounds.

sgl.**is_sound_playing**(*sound*)
Determines whether the specified sound is currently playing.

Parameters **sound** (*SGL sound*) – The sound object to query

Returns Whether the sound is playing

Return type bool

sgl.**play_music**(*file*, *volume=1.0*, *loops=-1*)
Plays, by default, infinitely looping background music. Music, unlike sounds, do not need to be loaded in advance. They are loaded when you call this function. Because of this, only one music track can be playing at once.

Parameters

- **file** (*str*) – The filename of the music to play
- **volume** (*float*) – The volume to play the music at, specified as a float. (So, 1.0 would be full volume, 0.5 would be half volume, and so on.)
- **loops** (*int*) – How many times to loop playback of the music. By default, this is -1, which will look the music forever, until *sgl.stop_music()* is called.

sgl.**pause_music**()
Pauses the currently playing piece of music. This is different from *sgl.stop_music()* in that you can later call *sgl.resume_music()* to resume the song from the exact point at which you paused it. With stopping, you have no choice but to restart the song.

sgl.**resume_music**()
Resumes a piece of music that has been paused with *sgl.pause_music()*.

sgl.**set_music_volume**(*volume*)
Sets the volume of the currently playing music.

Parameters **volume** (*float*) – The volume to play the music at, specified as a float. (So, 1.0 would be full volume, 0.5 would be half volume, and so on.)

sgl.**stop_music**()
Stops the currently playing music. This music will not be able to be resumed with *sgl.pause_music()*.

sgl.**is_music_playing**()
Returns whether any music is currently playing.

Returns Whether any music is playing

Return type bool

3.2.8 Input commands

These commands have to do with getting input.

sgl.**add_input**(*type*)
Initializes support for a specified type of input.

Parameters **type** (*int*) – The type of input to add. Should be a constant from `sgl.input`.

sgl.**supports_input**(*type*)
Returns whether the backend supports the specified input type. Might be integrated into *sgl.has()* later.

Parameters **type** (*int*) – The type of input to test. Should be a constant from `sgl.input`.

Returns Whether the specified type of input is supported by this backend

Return type bool

sgl.**remove_input**(*type*)
Removes support for a specified type of input. It does not matter whether this input is real or fake–it will remove it regardless.

Parameters **type** (*int*) – The type of input to remove. Should be a constant from `sgl.input`.

sgl.**has_input**(*type*)
Returns whether a specified input type is handled or not. Intentionally does not distinguish between real and fake inputs.

Parameters **type** (*int*) – The type of input to test. Should be a constant from `sgl.input`.

Returns Whether the specified type of input is currently handled

Return type bool

sgl.**on_key_down**(*key*)
Briefly returns *True* on the frame a keyboard key is pressed down.

Parameters **key** (*int*) – The key code of the key to test. Should be a constant from `sgl.key`.

Returns Whether a key has just been pressed down

Return type bool

sgl.**on_key_up**(*key*)
Briefly returns *True* on the frame a keyboard key is released.

Parameters **key** (*int*) – The key code of the key to test. Should be a constant from `sgl.key`.

Returns Whether a key has just been released

Return type bool

sgl.**is_key_pressed**(*key*)
Returns *True* if the specified keyboard key is currently pressed.

Parameters **key** (*int*) – The key code of the key to test. Should be a constant from sgl.key.

Returns Whether a key is currently down

Return type bool

sgl.**get_keys_pressed**()
Returns a list of all the keyboard keys currently pressed.

Returns A list of all pressed keys

Return type list of ints

sgl.**get_letters_pressed**()
Returns a string containing all the characters typed in the last frame. Handles capitalizing letters and changing characters when shift is pressed.

Returns The text typed in the last frame

Return type str

sgl.**show_mouse**()
Shows the system mouse cursor.

sgl.**hide_mouse**()
Hides the system mouse cursor.

sgl.**get_mouse_x**()
Returns what the mouse cursor's x position on the current frame.

Returns The mouse's x coordinates

Return type int

sgl.**get_mouse_y**()
Returns what the mouse cursor's y position on the current frame.

Returns The mouse's y coordinates

Return type int

sgl.**get_prev_mouse_x**()
Returns what the mouse cursor's x position was on the previous frame. Exists for convenience and because when managing the event loop automatically, it is impossible for your program to retrieve this value manually.

Returns The mouse's x coordinates on the previous frame

Return type int

sgl.**get_prev_mouse_y**()
Returns what the mouse cursor's y position was on the previous frame.

Returns The mouse's y coordinates on the previous frame

Return type int

sgl.**on_mouse_down**(*button=1*)
Briefly returns *True* on the frame the specified mouse button is pressed down. By default, it tests for the left mouse button–button #1. Does not intelligently determine what the left mouse button is if the user has used their system settings to swap the functions of the mouse buttons.

Parameters **button** (*int*) – The number of the mouse button to test. Is 1 by default.

Returns Whether a mouse button has just been pressed down

Return type bool

sgl.**is_mouse_pressed**(*button=1*)
Returns *True* if the specified mouse button is currently pressed down.

Parameters **button** (*int*) – The number of the mouse button to test. Is 1 by default.

Returns Whether a mouse button is pressed

Return type bool

sgl.**get_mouse_buttons_pressed**()
Returns a list of all the currently pressed mouse buttons.

Returns A list of all the pressed mouse buttons

Return type list of ints

sgl.**on_joy_down**(*button*)
Briefly returns *True* on the frame the specified joystick button is pressed down.

Parameters **button** (*int*) – The number of the joystick button to test

Returns Whether a joystick button has just been pressed down

Return type bool

sgl.**on_joy_up**(*button*)
Briefly returns *True* on the frame the specified joystick button is released.

Parameters **button** (*int*) – The number of the joystick button to test

Returns Whether a joystick button has just been pressed released

Return type bool

sgl.**is_joy_pressed**(*button*)
Returns *True* if the specified joystick button is currently pressed down.

Parameters **button** (*int*) – The number of the joystick button to test

Returns Whether a joystick button is pressed

Return type bool

sgl.**get_joy_axis**(*axis*)
Returns the value of a joystick's given "axis". There is usually a separate axis for each direction of each stick. For example, there might be an axis for the horizontal motion of the left stick, the vertical motion of the left stick, and axes for both on right stick. Some joysticks, however, use axes for other things as well, such as pressure sensitive buttons. Experiment to figure out which is which.

Parameters **axis** (*int*) – The axis to get the value from

Returns The value reported by the current axis

Return type float

sgl.**get_joy_num_axes**()
Returns the amount of axes this joystick reports having.

Returns The number of axes on this joystick

Return type int

3.2.9 Fake input commands

These commands have to do the fake input system, in which on platforms without certain types of input, such as smart phones, you can simulate unavailable input devices with the ones that are available.

> **Warning:** I don't think this API actually makes any sense. I might remove or change it later.

`sgl.`**`add_fake_input`**(*type*)
Adds a fake input. There must not be an equivalent real input defined.

`sgl.`**`got_key_down`**(*key*)
Simulates a key on the keyboard being pressed down.

`sgl.`**`got_key_up`**(*key*)
Simulates a key on the keyboard being released. If you do not call this function, SGL will think the key has been pressed down forever.

`sgl.`**`got_mouse_move`**(*x*, *y*)
Simulates the mouse moving to a new point.

`sgl.`**`got_mouse_down`**(*button=1*)
Simulates a mouse button being pressed.

`sgl.`**`got_mouse_up`**(*button=1*)
Simulates a mouse button being released. Similarly to *`sgl.got_key_up()`*, this is necessary for SGL to ever consider a mouse button released.

CHAPTER 4

SGL Library Reference

`sgl.lib` is where a lot of the functions useful for making games will live. Not much of it is written right now, and you should probably avoid using most of what's there, as it might change. Here's a brief summary of the modules it currently includes, though:

- `sgl.lib.Script` - Provides a small domain specific markup language for writing in game cut scenes. Is designed to be easy for writers to understand and for programmers to implement support for. The syntax of this language is currently undergoing a great revision. This is the module that you should avoid using the most right now.
- `sgl.lib.Time` - Provides the ability to register functions to happen at specific times or intervals. Useful for making pieces of code happen at a specific frame rate, for animated sprites and such.
- `sgl.lib.Tween` - Provides "tweening" functionality, which lets you specify start and end values for any numerical property of any object, and animate them moving from point A to point B in a given amount of time with the given interpolation. Makes it much easier to add dynamic animations to in game GUIs and game elements.

Sound exciting? Hopefully, because there will be even more in the future! :p

4.1 Actual Reference

4.1.1 sgl.lib.Sprite

This module provides the bulk of the functionality of *sgl.lib*. Nearly every other class in *sgl.lib* inherits from or uses *`sgl.lib.Sprite`* somehow.

class `sgl.lib.Sprite.`**`AnimatedSprite`**

A subclass of *`Sprite`* that provides extra functions useful for managing frame-based animations.

`animation`

string: Property containing the name of the currently playing animation. This name should correspond to one of the keys in `animations`.

If you set this property, it will immediately switch to the animation of that name. The current playing state (whether this sprite is playing whatever the current animation is, or paused), however, will remain unchanged.

`pause``()`

Pauses playment that the current point. You can call *`play`* to resume playment from this point.

`play``()`

Starts playing the active animation.

playing
bool: Returns whether this sprite is currently playing whatever the active animation is.

preupdate()
Overridden to handle animation logic, in addition to what *Sprite.preupdate* doas.

Todo:

- It might be useful to be able to restrict how many times an animation can loop, so you can, say, only play an animation once. Currently you can simulate this callbacks, but that's a little cumbersome.
- Maybe make this attempt to make up for lost time, like `sgl.lib.Time` does.

stop()
Stops playment. This is the same as calling *pause*, except the animation is reset to the first frame before stopping.

class sgl.lib.Sprite.**App**(*first_scene*)
A very simple object that wraps over scenes in order to let you easily switch between multiple scenes.

Parameters **first_scene** (*Scene*) – The scene that will be active by default. The scene will be activated by *switch_scene*.

draw()
A draw function you can pass to *sgl.run*.

switch_scene(*scene*)
Changes the active scene to another one. Also updates that scene's `Sprite.app` reference to refer to this object, so you can switch to other scenes from that scene easily.

Parameters **scene** (*Scene*) – The scene to switch to.

update()
An update function you can pass to *sgl.run*.

class sgl.lib.Sprite.**Camera**
A small object to hold the coordinates of the camera in a given scene.

position
tuple: The position of the camera—or, in other words, the coordinates in the scene that will be displayed at the top left corner of the window.

Can also be accessed through the `x` and `y` attributes.

class sgl.lib.Sprite.**EllipseSprite**
A subclass of *ShapeSprite* that draws an ellipse (or circle, if `width` and `height` are the same) in the bounding box of the sprite.

class sgl.lib.Sprite.**PerspectiveGroup**
A subclass of *SpriteGroup* that disregards the usual sprite rendering order and instead draws every sprite inside in the order of their y coordinates. This means that sprites positioned higher on the screen are drawn below sprites positioned lower on the screen. In most two-dimensional projections of perspective, this provides the illusion that some sprites are positioned "behind" other sprites.

By default, it completely flattens the sprite hierarchy inside when drawing sprites. This can be customized with the `max_level` attribute, or by setting a `no_perspective` attribute to `True` on a given sprite instance—this will make it so children of that sprites are not flattened out, and are drawn in their intended order.

draw_children()
You should not need to deal with this function yourself, but it is useful to note—*PerspectiveGroup*'s implementation of draw_children reimplements most of *Sprite.draw_children* without reusing

code. The two method implementations may become the synchronized. If you experience bugs in *PerspectiveGroup* that do not occur in normal sprites, this function is likely the cause.

class sgl.lib.Sprite.**RectSprite**
A subclass of *ShapeSprite* that draws a rectangle in the bounding box of the sprite.

class sgl.lib.Sprite.**Scene**
A special kind of *Sprite* designed to provide a few additional functions useful in managing larger game levels.

class sgl.lib.Sprite.**ShapeSprite**
A subclass of *Sprite* designed to handle shapes drawn with the SGL drawing commands instead of blitting surfaces to the screen.

This will often be faster than drawing the equivalent surface.

draw_shape()
Override this function to specify what shape this sprite should draw. The stroke and fill properties will have already been set by the other functions of this class, and will automatically be restored to their previous values after this function has been executed.

class sgl.lib.Sprite.**Sprite**(*graphic=None*)
Provides a class to represent drawable objects.

Parameters **graphic** (*SGL Surface*) – A graphic that will be loaded by *Sprite.load_surface* during initialization.

add(*sprite*)
Adds a child sprite to this end of this sprite's sprite list.

Parameters **sprite** (*Sprite*) – A *Sprite* instance to add.

anchor
tuple: The position of the anchor point of this sprite. This determines the coordinates that is considered (0, 0) for this sprite, and impacts which point it is rotated around if you use that effect.

If either dimension is a floating-point number, they'll be interpreted as a percentage of the sprite's width or height. So, setting the anchor point to (0.5, 0.5) will placed in the center of the sprite's graphic.

Can also be accessed through the a_x and a_y attributes.

autosize()
Updates this sprite's size to match the size of its surface. Automatically called by *Sprite.load_surface* and when sprites are initialized with surfaces.

center()
Utility function to place this sprite in the center of its parent sprite. If it has no parent sprite, it will be placed in the center of the screen.

centre()
Like *center*, but British.

collision_rect
sgl.lib.Rect.Rect: The bounding box that the collision functions will use to determine when this sprite is overlapping with others. If none is specified, it will assume you want the entire sprite to be collidable.

draw()
Handle drawing this sprite and its children. Should not be overwritten to define custom drawing logic, unless you want to break drawing child sprites. Instead, override *Sprite.draw_self*.

draw_children()
Loops through and draws child sprites. Ideally, should not be called manually.

Currently, this updates each sprite's screen position one more time, to make them slightly more predictable.

draw_self()
Handle drawings this sprite. If you want to customize how sprite drawing works, override this function.

fill()
Utility function to make this sprite completely fill its parent sprite. If it has no parent sprite, it will fill the screen.

is_being_collided(*other*)
Returns whether this sprite is colliding with another one.

Parameters **sprite** (*Sprite*) – Another *Sprite* instance to test.

Returns Whether this sprite is colliding with the other one.

Return type bool

is_colliding_with(*other*)
Returns whether this sprite is colliding with another one. Currently just passes responsibility to the other sprite by calling its *Sprite.is_being_collided* function.

Parameters **sprite** (*Sprite*) – Another *Sprite* instance to test.

Returns Whether this sprite is colliding with the other one.

Return type bool

is_mouse_over()
Returns whether the mouse cursor is currently inside this sprite's bounding box.

Returns Whether the mouse is inside.

Return type bool

Todo:

- Come up with some type of GUI-like event system. Simple functions like these will not help determine whether there's a sprite above this one that should get the focus first and such.

kill()
Marks this sprite to be deleted on the next frame.

load_surface(*surface*)
Sets the graphic for this sprite, and makes the size match the size of this graphic.

Parameters **surface** (*SGL Surface*) – A surface containing the graphic you want this sprite to draw.

on_add()
Exectued when a sprite is added. It's recommended to put initialization code here instead of `__init__`, so that the sprite will have proper access to its parent elements during and after initialization. (Not all the examples have been updated to work like this, though.)

position
tuple: The local position of this sprite (before any transformations have been applied).

Can also be accessed through the `x` and `y` attributes.

preupdate()
Called before this sprite updates. Currently, contains logic to save the previous position and update the screen position.

prev_position
tuple: Read-only property returning the local position of this sprite on the previous frame.

Can also be accessed through the `prev_x` and `prev_y` attributes.

real_anchor
tuple: Read-only property returning the position of the anchor point of this sprite after percentage values (such as 0.5) have been converted to real coordinates.

rect
`sgl.lib.Rect.Rect`: Read-only property returning the bounding box of this sprite in local coordinates.

screen_collision_rect
`sgl.lib.Rect.Rect`: Read-only property returning the bounding box of this sprite in screen coordinates (after parent and camera transformations have been applied).

screen_position
tuple: Read-only property returning the screen position of this sprite (after parent and camera transformations have been applied). Can also be accessed through the `screen_x` and `screen_y` attributes.

screen_rect
`sgl.lib.Rect.Rect`: Read-only property returning the bounding box of this sprite in screen coordinates (after camera and parent transformations have been applied).

size
tuple: The size of this sprite. Used for positioning the anchor point, determining whether a sprite is visible and should be drawn, and as a default size for the sprite's collision bounding box.

update()
Called every frame to update this sprite and its child sprites' logic. To define logic for your sprite, override this function. If you want child sprite updating to work properly, though, make sure to call the *Sprite* base classes' update function before yours.

update_screen_positions()
Screen positions are slightly broken. By default, screen coordinates are only updated in the *Sprite.preupdate* function. This means that sometimes they will update a frame late if you change a parent sprite's position during its *Sprite.update* phase. In addition, even when it does update, sometimes position changes will not propagate through the hierarchy correctly.

This function can help you work around that by forcing all of this sprite and all of its child sprites to recursively update their screen coordinates to the correct value. If you see child sprites behaving strangely, try adding a call to this in the parent sprite's code.

I realize this is a bit of a hack, but it is the lesser evil. I tried to update all screen positions on every frame, but that slowed the sprite system down significantly and created new positioning problems that, unlike these, could not be solved with a single extra function call.

world_collision_rect
`sgl.lib.Rect.Rect`: Read-only property returning the bounding box of this sprite in world coordinates (after parent, but not camera transformations have been applied).

Todo:

- Make this property only disregard camera transformations, instead of all of them.

world_to_screen(*x*, *y*)
Converts local coordinates in this sprite's space to screen coordinates.

Parameters

- **x** (*number*) – The x coordinates to convert.

- **y** (*number*) – The y coordinates to convert.

Returns The new coordinates.

Return type tuple

class sgl.lib.Sprite.**SpriteGroup**
A subclass of *Sprite* designed to do nothing but hold other sprites.

Identical to a plain *Sprite*, except `infinite_space` is turned on by default.

sgl.lib.Sprite.**Spritesheet**(*surface*, *frame_width=0*, *frame_height=0*)
Takes a surface containing a spritesheet and extracts each individual frame from it. Often necessary to use *AnimatedSprite* effectively.

Parameters

- **surface** (*SGL Surface*) – A graphic containing a spritesheet with equally sized frames starting from (0,0). This function does not do anything in regards to transparency. You must load your graphic with *sgl.set_transparent_color* or *sgl.load_alpha_image* for transparency to work.
- **frame_width** (*int*) – How wide each frame is.
- **frame_height** (*int*) – How high each frame is.

Returns A list of each frame in this spritesheet.

Return type list

Todo:

- This may eventually be a class, so that spritesheets can blit chunks from a single surface instead of initializing possibly hundreds of tiny little surfaces for each frame. I think that will be more efficient, particularly with hardware accelerated backends. I'll try to make this class behave identically to a list in most cases, but keep in mind that code depending on modifying the frames of a spritesheet like a list could break when this change happens.

class sgl.lib.Sprite.**Viewport**
A *Sprite* that behaves identically to a *Scene* in most circumstances, except it is only rendered in the bounds of its bounding box.

Todo:

- Make scenes and viewports the same thing. Like, a scene automatically become a viewport if its size does not fill the screen, it will be drawn bigger if at a scale, etc.
- When this happens, force all the examples to run in wiggling (or at least bouncing) viewports. It should be possible to embed scenes in other scenes arbitrarily. The ability to make game anthologies, or browsable demo packages, like the WxPython demo, should be nearly automatic.

4.1.2 sgl.lib.Collision

This module provides some functions for working with bounding box based collisions. It is fairly basic, and not very optimized, but for most use cases, it works out of the box and does not let objects randomly get stuck in and fall through each other, something a disturbing amount of game engines lack.

class sgl.lib.Collision.**CollisionChecker**(*object1*, *object2*, *callback*)
An object that checks whether sprites are overlapping every frame, and takes user-specified action as a result.

This is mostly used by the backend of *sgl.lib.Collision*. From an end user's perspective, this class will mostly be used to stop and resume collision checking.

pause()
Temporarily pauses this collision checker, without deleting it.

resume()
Resumes a previously paused collision checker.

stop()
Stops a collision checker entirely by deleting the object.

class sgl.lib.Collision.**SlidingCollisionChecker**(*object1*, *object2*, *callback*)
Identical to a normal *CollisionChecker*, except that its update function provides additional logic to move objects outside of each other.

sgl.lib.Collision.**add**(*object1*, *object2*, *callback*)
Creates a basic collision checker, which will call a callback on every frame two objects overlap.

Will use a sprite's visible rectangle for collisions, unless a Sprite has a custom *Sprite.collision_rect* defined.

Parameters

- **object1** (*Sprite*) – The first object to test. This must be a single sprite—subsprites will not be tested for collision.
- **object2** (*Sprite*) – The second object to test. This can be any kind of sprite, and subsprites will be tested for collision.
- **callback** (*function*) – The function to call when the objects overlap. This function will have different information passed to it depending on how many arguments it accepts.
  - If it accepts no arguments, nothing special will be passed to it.
  - If it accepts 1 argument, it will be passed a reference to the subsprite of `object2` that `object1` collided with on that frame.
  - If it accepts 2 arguments, it will be passed a reference to `object1`, and then the subsprite of `object2` that `object1` collided with on that frame.
  - If it accepts 3 arguments, it will be passed all of the above, and then a `sgl.lib.Rect.Rect` representing the intersected area between the two sprites.

Returns An object that can be used to pause or stop this collision checking if necessary.

Return type *CollisionChecker*

sgl.lib.Collision.**add_sliding**(*object1*, *object2*, *callback=None*)
Creates a sliding collision checker, which will prevent `object1` from going inside any of the sprites of `object2`. It is called a "sliding" collision checker because if an object is moving diagonally, it will continue to move in the free direction even if it collides against a wall. Because the collision algorithms are bounding box based, though, diagonal obstacles are not supported.

The sliding collision algorithm checks every pixel position between `object1`'s position on the current and previous frame to check collision, so object should not be of the slide through each other when they are moving fast or if the frame rate gets low. A consequence of this, however, is that this means manual position updates are subject to collision checking as well–if you want to teleport an object through a wall, you must temporarily turn off sliding collision checking for that to work.

Because of this, it's recommended you always save a reference to the *SlidingCollisionChecker* object this function returns.

Will use a sprite's visible rectangle for collisions, unless a Sprite has a custom *Sprite.collision_rect* defined.

Parameters

- **object1** (*Sprite*) – The first object to test. This must be a single sprite—subsprites will not be tested for collision.
- **object2** (*Sprite*) – The second object to test. This can be any kind of sprite, and subsprites will be tested for collision.
- **callback** (*function*) – The function to call when the objects overlap. This function will have different information passed to it depending on how many arguments it accepts.
  - If it accepts no arguments, nothing special will be passed to it.
  - If it accepts 1 argument, it will be passed a list of strings that can contain various permutations of `"left"`, `"right"`, `"top"`, and `"bottom"`. The strings present in this list will tell you on what sides `object1` was hit before its position was corrected by the collision checker.

Returns An object that can be used to pause or stop this collision checking if necessary.

Return type *SlidingCollisionChecker*

`sgl.lib.Collision.`**update**`()`
Must be called every frame for any of the collision checkers to work.

Todo:

- It might be handy to have this and the other continuously executing libraries (tween, time...) automatically inject themselves into the SGL event loop instead of making the user manually call their update functions every frame. Currently it is slightly annoying that one has to plan out a way to make sure these update functions are called *exactly* once every frame in their entire program. I'm not sure if handling this automatically is too "magic," though...

4.1.3 sgl.lib.Layout

This module mainly provides one class, *FlowLayout*, that lets one arrange sprite similarly to the automatic layout functions of some GUI frameworks. This can be useful for laying out game GUIs in a resolution independent way.

class `sgl.lib.Layout.`**FlowLayout**(*horizontal=False*)
A Sprite that provides similar functionality to wxWidget's BoxSizer.

Parameters **horizontal** (*bool*) – Whether this layout runs horizontally or not. If this is False, the default, it will run vertically. This can be set later with the *horizontal* property.

add(*sprite*, *proportion=0.0*, *proportion_other_way=0.0*, *align=0.0*)
Behaves similarly to *Sprite.add*, but with additional parameters to determine how the added sprite will be fit into the layout.

Parameters

- **sprite** (*Sprite*) – A *Sprite* instance to add.
- **proportion** (*float*) – The amount of space this sprite will take up in *the direction that the layout flows* (which is set by *horizontal*). If this value is 0, this sprite will merely take up its original size. If it is anything else, its size will be calculated relative to the size of the layout and the other sprites.

  Basically, the way it works is this: take all of the sprites with proportion values, and add those numbers up. That will be the *total proportion value.* The size of each sprite will be its own proportion value over the total proportion value.

  So, if the proportion value is 1 for every sprite in a layout, their sizes will be distributed evenly across the layout.

If one of those sprites has a proportion value of 2, however, it will be twice as large as the other ones, and the other sprites' sizes will be twice as small. It all balances out in the end.

It would be equally valid to set the larger sprite's proportion value to 1, and the smaller sprites' to 0.5 for this—the exact values do not matter as much as how the proportion values relate to *each other.* This is slightly different from wxWidgets, in which proportion values must be whole numbers.

It may make it easier to understand certain layouts if you specify proportion values as fractions (such as 0.25), and make sure they add up to 1.0 in the end.

It is also important to note that the space taken into account for proportioned sprites is *the space left behind by non-proportioned sprites.* So, if you have a layout with a few non-proportioned sprites, and add a sprite with a proportion value of 1 to that layout, it will not fill the entire layout—it will take up the entire space left behind by the other sprites.

This gives you many possibilities for positioning non-proportioned sprites. If you have two normal sprites and put a sprites with a proportion value of 1 between them, for example, you will force both normal sprites to be on opposite ends of the layout. If you want to distribute normal sprites equally across an area, you can space them out with proportion 1 spaces. And so on.

Proportion values are slightly difficult to understand, but once you get them, they will open up many possibilities for layout.

- **proportion_other_way** (*float*) – This argument is much easier to understand than the proportion value.

  It is a percentage value specifying how much space the sprite will take up on the axis perpendicular to the direction the layout flows. So, if the layout flows *vertically,* this specifies how a sprite's *horizontal* size will be affected by it.

  Usually, you will either want this value to be 0, which will leave the sprite's size in that direction untouched, or 1.0, which will make the sprite completely fill the layout in that direction. Anything in between will leave the sprite somewhere in the middle of that. (Or, more accurately, the size of the layout in that direction times this value.)

- **align** (*float*) – If `proportion_other_way` leaves any breathing room for the sprite, this specifies how that sprite should be *positioned* in the layout.

  If this is 0, the sprite will rest along the left or top edge of the layout. If this is 0.5, it will be centered along that direction. If this is 1.0, it will be bottom or right aligned. And other values will be somewhere inbetween.

  You will not need to set this argument in most cases. If you do, 0.5 will likely be the most common value you'll use for it.

clear()
Removes everything in this layout.

Todo: Is this a method of *Sprite*? If it isn't, why not?

horizontal
Whether this layout runs horizontally or not. If this is False, the default, it will run vertically.

reflow()
Repositions and resizes the elements inside to adhere to the layout rules. Must always be *manually called* to update the objects inside, even after adding new objects. (I tried having it automatically update the layouts, and the annoyance of having sprites move around at unpredictable times outweighed the benefits.)

Is a potentially expensive function. If there are proportioned sprites in the layout, it must run two passes through the objects inside. In most cases, is recommended to call this function once—after this layout has

been positioned and sized correctly, and after all of its items have been added.

If any sprites inside this layout have a `reflow` method, it will call it. This is useful for embedding layouts hierarchically, or specifying custom logic for sprites when they are resized.

class `sgl.lib.Layout.`**`Spacer`**(*width=0*, *height=0*)

A dummy object you can add to *`FlowLayout`* instead of a *`Sprite`* when you want to have an invisible spacer sprite to affect the positioning of other sprites.

Provides the absolute minimum amount of data required of an object to interact with the sprite system.

Parameters

- **`width`** (*`int`*) – The width of the spacer, if you do not plan to have it controlled by a proportion value.
- **`height`** (*`int`*) – The height of the spacer, if you do not plan to have it controlled by a proportion value.

4.1.4 sgl.lib.ScriptParser

This module provides the functionality to parse SGLscript. It is mainly used behind the scenes to provide `sgl.lib.Script` with the data it needs to operate, but you may need to interact with this module for real-time script manipulation.

An SGLscript file is formatted like follows:

- First, there is a header section. The format of this section is documented in *`ScriptParser.header`*. (Please read that after reading this, though—the documentation for that function assumes some familiarity with the format of commands.)
- Then, you must specify the beginning of a "label" by having a line that begins with `@`, with the rest of the line consisting of the label name. Any line beginning with a `@` starts a new label.

  Warning: This may be deprecated in favor of just using the normal command syntax to define labels in the future. Even if that happens, though, it will be possible to replicate the current behavior through slightly more configurable macros.

- Every line after that is interpreted as the body of a given label.

In the body, two main rules apply:

- Normal text is interpreted as dialogue that will get output to the screen.
- Anything with in square braces (`[` and `]`) is interpreted as a command.

Commands can consist of three components:

- All commands must have a *command name.* This determines what function is used to handle this command when the script is executed.
- Then, after this, they can have *positional arguments.* These, like positional arguments in Python, or arguments that are specified by order instead of by name.
- Then, after (or instead of) positional arguments, commands can have *keyword arguments.* These are arguments in which their name is specified, and then their value.

There are two syntaxes of commands:

- There are the *old-style commands,* which behave like HTML tags. In this syntax, the command name is specified at the beginning, and is ended by whitespace. After this whitespace, you can have whitespace separated

positional arguments, and then whitespace separated keyword arguments. Keyword arguments are given in a `key=value` form, and any amount of whitespace is accepted before and after the =.

The type of commands look like this: `[show-background "tree.png" transition="fade-in" fade-time=5]`.

> **Warning:** These types of commands will be deprecated as soon as possible. Don't use them. I'm merely documenting them so the source code makes more sense.

- There are the *new-style commands,* or *pretty commands.* With this syntax, command names are first, but are ended by a colon (`:`). Because of this, command names can have spaces in them. Then, positional arguments can be specified the same way as with the old-style. Then, keyword arguments can be specified, but in a `key: value` form. Thus, keyword arguments, like command names, can also have spaces in their names. This creates some ambiguous parsing situations, but these commands are much easier to read.

  These types of commands look like this: `[show background: "tree.png" transition: "fade-in" fade time: 5]`.

  With the sense of commands, command and keyword argument names are case insensitive, and can handle arbitrary whitespace in the middle of them. As long as you don't misspell them, you can mangle their formatting quite a bit and they will parse correctly.

  Todo:

  - Okay, we might have to see about that "arbitrary" whitespace... :|

  > **Warning:** These types of commands will be what SGLscript uses in the future. Positional arguments may be deprecated in the future, so please stick to using keyword arguments only if you want to play it safe. This will also make your scripts easier to read.

Both styles use identical syntax for values for arguments:

- If a value begins with a `"` or `'`, it is a string. These behave the same as Python strings—you must escape the other type of quote with backslashes.
- If a value begins with a digit or `-`, it is a number. Mathematical expressions are not currently allowed—you must just input numbers.
- Anything else is interpreted as a *keyword*—or a single word string. This string must have no spaces is in it, and it will be converted to lowercase during parsing. This is a shortcut for specifying short string arguments without having to type quotes.

  For example, the earlier example could be inputted as `[show background: tree.png transition: fade-in fade time: 5]` with no information being lost.

  > **Warning:** Quoteless string values may be deprecated in the future. They can create some annoyingly ambiguous parsing situations with pretty commands, like `[show thing: cool transition: "dissolve"]`. Is "cool" a positional argument or is the name of the keyword argument "cool transition"?

I cannot guarantee the behavior of anything left undescribed here. You have been warned.

In addition, the parser supports some automatic manipulation of commands during parsing. This is documented more thoroughly in *ParserSettings*.

class `sgl.lib.ScriptParser.`**`Command`**(*name*, *pos_arg=[]*, *key_arg={}*)

Represents a single command in a script. Converting this to a string should yield a parsable command.

Todo:

- Make that yield a parsable *pretty* command.

class sgl.lib.ScriptParser.**Label**(*name*, *body*)
A class to represent labels in the script.

class sgl.lib.ScriptParser.**Parser**(*text=''*)
A base class for handling recursive descent parsers (possibly badly).

You should definitely not use this class in your own programs, but I need to document it for myself, so there.

Parameters **text** (*string*) – If this argument is specified, it will call *load_text* with this text during initialization.

at_content_beginning()
Returns if we're at the beginning of the content of the line. (As in, past the whitespace.)

at_line_beginning()
Returns if we're at the beginning of the given line.

at_literal()
Returns whether we currently at the beginning of a string or number literal.

char
Returns the current character.

eat(*char*)
Eats a character, and if it's not what you specify, raises an error with *error*.

Parameters **char** (*string*) – The character the current character must be equal to.

error(*text*)
Raises an error and prints information about the state of the parser.

Parameters **text** (*string*) – A short description of the error.

Raises *ParserError* – Will always raise this.

Todo:

- Maybe I should be using normal exceptions like a normal person? Not really sure how to easily do that while keeping track of the line and column position, though.

line_empty()
Peeks ahead to see if the current line has any non-whitespace content or not.

literal()
Reads a string or number literal, and returns the parsed value.

load_text(*text*)
Loads text into the parser, and resets all the position counters.

Parameters **text** (*string*) – A string containing the text to load. Should be able to deal fine with Unicode. Should.

newline()
Expects a newline at char.

next()
Moves the position onwards, keeping track of the current line and column.

next_char
Read-only property returning the character after the current one.

position_valid()
Returns if we're in the document.

prev_char
Read-only property returning the character before the current one.

symbol()
Reads a symbol name.

Todo: Has a bunch of hardcoded values for SGLscript. Factor out.

whitespace(*allow_newline=True*)
Keeps advancing the cursor until reaching non-whitespace characters.

Parameters **allow_newline** (*bool*) – Whether to eat newlines as well.

exception sgl.lib.ScriptParser.**ParserError**(*text*, *line*, *char*)
This exception handles errors that happened while parsing a script.

class sgl.lib.ScriptParser.**ParserSettings**
An object for holding settings on how the SGLscript parser behaves that the user is meant to be able to change.

class sgl.lib.ScriptParser.**ScriptParser**(*text=''*)
The object you'll be interacting with the most if you use this module. This is what actually lets you take a string, parse it, and get back the proper objects.

Todo:

- Trash a good portion of BodyParser.
- Un-hardcode the concept of labels. Have labels just be commands whose position is tracked so they can be quickly accessed.

Parameters **text** (*string*) – If this argument is specified, it will call *load_text* with this text during initialization.

header()
Parses special commands before the first label. You should not be calling this yourself, but the header is strange enough to warrant additional documentation.

Currently, you can only use two commands here:

- `define macro`: This takes two positional arguments. The first one is the original text, the second one is what to replace that text with.
- `use pretty commands`: This determines whether to parse all commands as pretty commands or not. It only takes one positional argument, which must be the string or keyword "yes" to activate this.

If you attempt to use any other commands, or normal body text here, it will raise a HeaderError. Comments and all permutations of whitespace are fine.

You can potentially raise a MacroError by defining macros that begin with characters that conflict with SGLscript (currently `\`, `[`, and `@`). Don't do this.

Both of those types of exceptions only have one parameter, `text`, so I'm not documenting them in detail.

parse()
Parses a script loaded by *load_text* and returns a dictionary of labels.

Returns A dictionary in which the keys are the name of a given level, and the value is the corresponding *Label* object.

Return type dictionary

Todo:

- *Label* objects also contains the name of the label. There has to be a less dumb way to do this.

reset_settings()
Resets all parser settings to their default values.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

S

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W