SGE Game Engine Documentation

Release 0.24

onpon4

March 10, 2016

CONTENTS

1 SGE Fundamentals **1**
- 1.1 SGE Concepts . . . 1
- 1.2 Global Variables and Constants . . . 3
- 1.3 Information specific to the Pygame SGE . . . 4

2 Tutorial 1: Hello, world! **7**
- 2.1 Setting Up a Project . . . 7
- 2.2 Adding Game Logic . . . 8
- 2.3 Starting the Game . . . 9
- 2.4 The Final Result . . . 10

3 Tutorial 2: Pong **13**
- 3.1 Adding Game Logic . . . 13
- 3.2 Starting the Game . . . 18
- 3.3 The Final Result . . . 19

4 Tutorial 3: Better Pong **23**
- 4.1 Adding Scoring . . . 23
- 4.2 Adding Sounds . . . 27
- 4.3 Adding Joystick Support . . . 28
- 4.4 The Final Result . . . 29

5 sge.input **35**
- 5.1 Input Event Classes . . . 35

6 sge.dsp **41**
- 6.1 sge.dsp Classes . . . 41

7 sge.gfx **63**
- 7.1 sge.gfx Classes . . . 63

8 sge.snd **77**
- 8.1 sge.snd Classes . . . 77
- 8.2 sge.snd Functions . . . 80

9 sge.collision **81**
- 9.1 sge.collision Functions . . . 81

10 sge.joystick **83**
- 10.1 sge.joystick Functions . . . 83

11 sge.keyboard **85**

11.1 sge.keyboard Functions 85

12 sge.mouse **87**
12.1 sge.mouse Functions 87

13 Indices and tables **89**

Python Module Index **91**

Index **93**

CHAPTER

ONE

SGE FUNDAMENTALS

Contents

- SGE Fundamentals
  - SGE Concepts
    - Events
    - Position
    - Z-Axis
    - The Game Loop
  - Global Variables and Constants
  - Information specific to the Pygame SGE
    - License
    - Dependencies
    - Formats Support
    - Missing Features
    - Known Problems
      - Keyboard Lock-up
      - Saving PNG Images

The SGE Game Engine ("SGE", pronounced like "Sage") is a general-purpose 2-D game engine. It takes care of several details for you so you can focus on the game itself. This makes more rapid game development possible, and it also makes the SGE easy to learn.

The SGE is libre software, and the SGE documentation (including all docstrings) is released to the public domain via CC0.

Although it isn't required, you are encouraged to release your games' code under a libre software license, such as the GNU General Public License, the Expat License, or the Apache License. Doing so is easy, does not negatively affect you, and is highly appreciated as a contribution to a free society.

1.1 SGE Concepts

1.1.1 Events

The SGE uses an event-based system. When an event occurs, a certain event method (with a name that begins with `event_`) is called. To define actions triggered by events, simply override the appropriate event method.

At a lower level, it is possible to read "input events" from `sge.game.input_events` and handle them manually. See the documentation for `sge.input` for more information. This is not recommended, however, unless you are running your own loop for some reason (in which case it is necessary to do this in order to get input from the user).

1.1.2 Position

In all cases of positioning for the SGE, it is based on a two-dimensional graph with each unit being a pixel. This graph is not quite like regular graphs. The horizontal direction, normally called `x`, is the same as the x-axis on a regular graph; `0` is the origin, positive numbers are to the right of the origin, and negative numbers are to the left of the origin. However, in the vertical direction, normally called `y`, `0` is the origin, positive numbers are below the origin, and negative numbers are above the origin. While slightly jarring if you are used to normal graphs, this is in fact common in 2-D game development and is also how pixels in most image formats are indexed.

Except where otherwise specified, the origin is always located at the top-leftmost position of an object.

In addition to integers, position variables are allowed by the SGE to be floating-point numbers.

1.1.3 Z-Axis

The SGE uses a Z-axis to determine where objects are placed in the third dimension. Objects with a higher Z value are considered to be closer to the viewer and thus will be placed over objects which have a lower Z value. Note that the Z-axis does not allow 3-D gameplay or effects; it is only used to tell the SGE what to do with objects that overlap. For example, if an object called `spam` has a Z value of `5` while an object called `eggs` has a Z value of `2`, `spam` will obscure part or all of `eggs` when the two objects overlap.

If two objects with the same Z-axis value overlap, the object which was most recently added to the room is placed in front.

1.1.4 The Game Loop

There can occasionally be times where you want to run your own loop, independent of the SGE's main loop. This is not recommended in general, but if you must (to freeze the game, for example), you should know the general game loop structure:

```
while True:
    # Input events
    sge.game.pump_input()
    while sge.game.input_events:
        event = sge.game.input_events.pop(0)

        # Handle event

    # Regulate speed
    time_passed = sge.game.regulate_speed()

    # Logic (e.g. collision detection and step events)

    # Refresh
    sge.game.refresh()
```

`sge.dsp.Game.pump_input()` should be called every frame regardless of whether or not user input is needed. Failing to call it will cause the queue to build up, but more importantly, the OS may decide that the program has locked up if it doesn't get a response for a long time.

`sge.dsp.Game.regulate_speed()` limits the frame rate of the game and tells you how much time has passed since the last frame. It is not technically necessary, but using it is highly recommended; otherwise, the CPU will be working harder than it needs to and if things are moving, their speed will be irregular.

`sge.dsp.Game.refresh()` is necessary for any changes to the screen to be seen by the user. This includes new objects, removed objects, new projections, discontinued projections, etc.

1.2 Global Variables and Constants

sge.**IMPLEMENTATION**
A string indicating the name of the SGE implementation.

sge.**BLEND_NORMAL**
Flag indicating normal blending.

sge.**BLEND_RGBA_ADD**
Flag indicating RGBA Addition blending: the red, green, blue, and alpha color values of the source are added to the respective color values of the destination, to a maximum of 255.

sge.**BLEND_RGBA_SUBTRACT**
Flag indicating RGBA Subtract blending: the red, green, blue, and alpha color values of the source are subtracted from the respective color values of the destination, to a minimum of 0.

sge.**BLEND_RGBA_MULTIPLY**
Flag indicating RGBA Multiply blending: the red, green, blue, and alpha color values of the source and destination are converted to values between 0 and 1 (divided by 255), the resulting destination color values are multiplied by the respective resulting source color values, and these results are converted back into values between 0 and 255 (multiplied by 255).

sge.**BLEND_RGBA_SCREEN**
Flag indicating RGBA Screen blending: the red, green, blue, and alpha color values of the source and destination are inverted (subtracted from 255) and converted to values between 0 and 1 (divided by 255), the resulting destination color values are multiplied by the respective resulting source color values, and these results are converted back into values between 0 and 255 (multiplied by 255) and inverted again (subtracted from 255).

sge.**BLEND_RGBA_MINIMUM**
Flag indicating RGBA Minimum (Darken Only) blending: the smallest respective red, green, blue, and alpha color values out of the source and destination are used.

sge.**BLEND_RGBA_MAXIMUM**
Flag indicating RGBA Maximum (Lighten Only) blending: the largest respective red, green, blue, and alpha color values out of the source and destination are used.

sge.**BLEND_RGB_ADD**
Flag indicating RGB Addition blending: the same thing as RGBA Addition blending (see sge.BLEND_RGBA_ADD) except the destination's alpha values are not changed.

sge.**BLEND_RGB_SUBTRACT**
Flag indicating RGB Subtract blending: the same thing as RGBA Subtract blending (see sge.BLEND_RGBA_SUBTRACT) except the destination's alpha values are not changed.

sge.**BLEND_RGB_MULTIPLY**
Flag indicating RGB Multiply blending: the same thing as RGBA Multiply blending (see sge.BLEND_RGBA_MULTIPLY) except the destination's alpha values are not changed.

sge.**BLEND_RGB_SCREEN**
Flag indicating RGB Screen blending: the same thing as RGBA Screen blending (see sge.BLEND_RGBA_SCREEN) except the destination's alpha values are not changed.

sge.**BLEND_RGB_MINIMUM**
Flag indicating RGB Minimum (Darken Only) blending: the same thing as RGBA Minimum blending (see sge.BLEND_RGBA_MINIMUM) except the destination's alpha values are not changed.

sge.**BLEND_RGB_MAXIMUM**
Flag indicating RGB Maximum (Lighten Only) blending: the same thing as RGBA Maximum blending (see sge.BLEND_RGBA_MAXIMUM) except the destination's alpha values are not changed.

sge.**game**
Stores the current sge.dsp.Game object. If there is no sge.dsp.Game object currently, this variable is set to None.

1.3 Information specific to the Pygame SGE

1.3.1 License

The Pygame SGE is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

The Pygame SGE is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with the Pygame SGE. If not, see <http://www.gnu.org/licenses/>.

1.3.2 Dependencies

- Python 2 (2.7 or later) or 3 (3.1 or later) <http://www.python.org>
- Pygame 1.9.1 or later <http://pygame.org>

1.3.3 Formats Support

sge.gfx.Sprite supports the following image formats:

- PNG
- JPEG
- Non-animated GIF
- BMP
- PCX
- Uncompressed Truevision TGA
- TIFF
- ILBM
- Netpbm
- X Pixmap

If Pygame is built without full image support, sge.gfx.Sprite will only be able to load uncompressed BMP images.

sge.snd.Sound supports the following audio formats:

- Uncompressed WAV
- Ogg Vorbis

sge.snd.Music supports the following audio formats:

- Ogg Vorbis
- MOD
- XM
- MIDI

MP3 is also supported on some systems, but not all, due to software idea patents which restrict use of this format. On some systems, attempting to load an unsupported format can crash the game. Since MP3 support is not available on all systems, it is best to avoid using it; consider using Ogg Vorbis instead.

For starting position in MOD files, the pattern order number is used instead of the number of milliseconds.

The pygame.mixer module, which is used for all audio playback, is optional and depends on SDL_mixer; if pygame.mixer is unavailable, sounds and music will not play.

1.3.4 Missing Features

`sge.gfx.Sprite.draw_line()`, `sge.dsp.Room.project_line()`, and `sge.dsp.Game.project_line()` support anti-aliasing for lines with a thickness of 1 only. `sge.gfx.Sprite.draw_polygon()`, `sge.dsp.Room.project_polygon()`, and `sge.dsp.Game.project_polygon()` support anti-aliasing for outlines of polygons with a thickness of 1 only. `sge.gfx.Sprite.draw_text()`, `sge.dsp.Room.project_text()`, and `sge.dsp.Game.project_text()` support anti-aliasing in all cases. No other drawing or projecting methods support anti-aliasing.

1.3.5 Known Problems

Since the Pygame SGE uses Pygame, it necessarily inherits Pygame's bugs. Below, some notable bugs in Pygame 1.9.1 (the latest Pygame release) are indicated.

Keyboard Lock-up

There is a bug in either Pygame or SDL, most likely SDL, which sometimes causes keyboard input to stop working. In Pygame programs such as this one, this occurs when pygame.display.set_mode is called multiple times, which in the Pygame SGE occurs any time either the size of the window or the video mode (windowed or fullscreen) changes. See this post from the SGE blog for more information:

https://savannah.nongnu.org/forum/forum.php?forum_id=8113

You may also be interested in this report on the Pygame issue tracker:

https://bitbucket.org/pygame/pygame/issue/212/

As mentioned in the post on the SGE blog, this is a particularly serious problem for anyone using the X Window System (e.g. pretty much any GNU/Linux user), or any other window system that gives complete control to fullscreen SDL applications. On these systems, if the game requires keyboard input to either leave fullscreen or exit, the system will become unresponsive to everything that isn't sent directly to the kernel (such as the magic SysRq key in Linux systems).

Luckily, the bug doesn't seem to affect mouse input, so if you allow the player to enter fullscreen mode in-game, it is highly recommended for you to provide some method of either exiting the game or exiting fullscreen with the mouse. This can be a button somewhere on the screen if the game uses the mouse cursor, or it can be a simple mouse button click otherwise.

Saving PNG Images

Some versions of Pygame have problems saving color information of PNG images correctly. As a result, depending on your exact version of Pygame, images saves with `sge.gfx.Sprite.save()` may be discolored. The best workaround is to choose a different format, such as BMP, GIF, or JPEG.

CHAPTER

TWO

TUTORIAL 1: HELLO, WORLD!

Contents

- Tutorial 1: Hello, world!
  - Setting Up a Project
    - Shebang
    - License
    - Imports
  - Adding Game Logic
    - The Game Class
    - The Room Class
  - Starting the Game
  - The Final Result

The easiest way to learn something new is with an example. We will start with a very basic example: the traditional "Hello, world!" program. This example will just project "Hello, world!" onto the screen.

2.1 Setting Up a Project

First, we must create our project directory. I will use "~/hello".

Next, create the game source file inside "~/hello". I am calling it "hello.py".

Open hello.py so you can start editing it.

2.1.1 Shebang

All Python files which are supposed to be executed should start with a shebang, which is a line that tells POSIX systems (such as GNU/Linux systems, BSD, and OS X) how to execute the file. For Python 3, the version of Python we will be using, the shebang is:

```
#!/usr/bin/env python3
```

The shebang should be the very first line of the file. You should also make sure that the file itself uses Unix-style line endings ("\n"); this can be done in most text editors via a drop-down list available when you save, and is done by IDLE automatically. Windows-style line endings ("\r\n") are often interpreted wrongly in POSIX systems, which defeats the purpose of the shebang.

2.1.2 License

The file is copyrighted by default, so if you do not give the file a license, it will be illegal for anyone to copy and share the program. You should always choose a free/libre software license for your programs. In this example, I will use CC0, which is a public domain dedication tool. You can use CC0 if you want, or you can choose another license. You can learn about various free/libre software licenses at http://gnu.org/licenses/.

The license text I am using for CC0 is:

```
# Hello, world!
# Written in 2013 by Julian Marchant <onpon4@riseup.net>
#
# To the extent possible under law, the author(s) have dedicated all
# copyright and related and neighboring rights to this software to the
# public domain worldwide. This software is distributed without any
# warranty.
#
# You should have received a copy of the CC0 Public Domain Dedication
# along with this software. If not, see
# <http://creativecommons.org/publicdomain/zero/1.0/>.
```

Place your license text just under the shebang so that it is prominent.

2.1.3 Imports

Because we are using the SGE, we must import the `sge` library. Add the following line:

```
import sge
```

2.2 Adding Game Logic

2.2.1 The Game Class

In SGE games, everything is controlled by a "game" object. The game object controls everything at the global level, including global events. To define global events, we need to subclass `sge.dsp.Game` and create our own game class. We can just call this class `Game`:

```
class Game(sge.dsp.Game):

    def event_key_press(self, key, char):
        if key == 'escape':
            self.event_close()

    def event_close(self):
        self.end()
```

Because our example is simple, we only need to define two events: the close event, which occurs when the OS tells the game to close (most typically when a close button is clicked on), and the key press event, which occurs when a key is pressed. We want the game to end if either the OS tells it to close or the Esc key is pressed.

As you may have noticed, we define events by defining certain methods; in our case, we defined methods to override the `sge.dsp.Game.event_key_press()` and `sge.dsp.Game.event_close()` methods.

Our definition of `event_close()` is simple enough: we just call `sge.dsp.Game.end()`, which ends the game. Our definition of `event_key_press()` is slightly more complicated; first we have to check what key was pressed,

indicated by the `key` argument. If the key is the Esc key, we call our `event_close()` method. The reason for calling `event_close()` instead of just calling `end()` is simple: in the future, we might want to do more than just call `end()`; perhaps, for example, we decide that we want to add a confirmation dialog before actually quitting. By connecting the key press event to the close event, if we do change what the close event does, that change will also translate to the pressing of the Esc key, avoiding needless duplication of work.

2.2.2 The Room Class

Rooms are distinguished places where things happen; for example, each level in a game would typically be its own room, the title screen might be a room, the credits screen might be a room, and the options menu might be a room. In this example, we are only going to have one room, and this room is going to serve only one function: display "Hello, world!" in the center of the screen. This will be our room class:

```
class Room(sge.dsp.Room):

    def event_step(self, time_passed, delta_mult):
        sge.game.project_text(font, "Hello, world!", sge.game.width / 2,
                              sge.game.height / 2,
                              color=sge.gfx.Color("black"), halign="center",
                              valign="middle")
```

You can see that the room class is defined very similarly to the game class. We subclass `sge.dsp.Room` and add a method to override `sge.dsp.Room.event_step()`, which defines the step event of our room class. The step event happens over and over again, once every "frame". You can think of frames as being like the frames in a video; each frame makes small changes to the image on the screen and then gives you the new image in a fraction of a second, providing an illusion of movement.

To display "Hello, world!" onto the screen, we use `sge.dsp.Game.project_text()`, which instantly displays any text we want onto the screen. `sge.game` is a variable that always points to the `sge.dsp.Game` object currently in use.

The first argument of this method is the font to use; we don't have a font yet, but we are going to define one later and assign it to `font`. Next is the text to display, which for us is `"Hello, world!"`.

The next arguments are the horizontal and vertical location of the text on the screen; we set these to half of the game's width and height, respectively, to place the text in the center.

Now that all required arguments are defined, we are going to define the color of the text as a keyword argument, setting it explicitly to black.

Finally, we define `halign` and `valign` as keyword arguments; these arguments specify the horizontal and vertical alignment of the text, respectively.

You might be wondering: why do we keep doing this every frame? Can't we just do it once, since we're not changing the image? In fact, we can't. `sge.dsp.Game.project_text()` shows our text, but it only does so for one frame. You can think of it as working like a movie projector: if you keep the projector on, you will continue to see the image, but as soon as the projector stops projecting the image, you can no longer see the image from the projector. `sge.dsp.Game.project_text()` and other similar projection methods work the same way.

2.3 Starting the Game

If you try to run hello.py now, you will notice that nothing happens. This is because, while we defined the game logic, we didn't actually execute it.

Additionally, we are still missing a resource: the font object we want to use to project text onto the screen. We need to load this resource.

We are going to fix both of these problems by adding some code after our class definitions:

```
# Create Game object
Game()

# Create backgrounds
background = sge.gfx.Background([], sge.gfx.Color("white"))

# Load fonts
font = sge.gfx.Font()

# Create rooms
sge.game.start_room = Room(background=background)

if __name__ == '__main__':
    sge.game.start()
```

First, we create a `sge.dsp.Game` object; we don't need to store it in anything since it is automatically stored in `sge.game`.

Second, we create a `sge.gfx.Background` object to specify what the background looks like. We make our background all white, with no layers. (Layers are used to give backgrounds more than a solid color, which we don't need.)

Third, we create our font. We don't really care what this font looks like, so we allow the SGE to pick a font. If you do care what font is used, you can pass the name of a font onto the `name` keyword argument.

Fourth, we create a room. The only argument we pass is the background argument; we set this to the background we created earlier. Since it is the room that we are going to start the game with, we need to assign this room to the special attribute, `sge.game.start_room`, which indicates the room that the game starts with.

Finally, with everything in place, we call the `sge.dsp.Game.start()` method of our game object. This executes all the game logic we defined earlier. However, we only do this if the special Python variable, `__name__`, is set to `"__main__"`, which means that the current module is the main module, i.e. was executed rather than imported. It is a good practice to include this distinction between being executed and being imported in all of your Python scripts.

2.4 The Final Result

That's it! If you execute the script now, you will see a white screen with black text in the center reading "Hello, world!" Pressing the Esc key or clicking on the close button in the window will close the program. Congratulations on writing your first SGE program!

This is the completed Hello World program:

```
#!/usr/bin/env python3

# Hello, world!
# Written in 2013 by Julian Marchant <onpon4@riseup.net>
#
# To the extent possible under law, the author(s) have dedicated all
# copyright and related and neighboring rights to this software to the
# public domain worldwide. This software is distributed without any
# warranty.
#
# You should have received a copy of the CC0 Public Domain Dedication
# along with this software. If not, see
# <http://creativecommons.org/publicdomain/zero/1.0/>.
```

```
import sge


class Game(sge.dsp.Game):

    def event_key_press(self, key, char):
        if key == 'escape':
            self.event_close()

    def event_close(self):
        self.end()


class Room(sge.dsp.Room):

    def event_step(self, time_passed, delta_mult):
        sge.game.project_text(font, "Hello, world!", sge.game.width / 2,
                              sge.game.height / 2,
                              color=sge.gfx.Color("black"), halign="center",
                              valign="middle")


# Create Game object
Game()

# Create backgrounds
background = sge.gfx.Background([], sge.gfx.Color("white"))

# Load fonts
font = sge.gfx.Font()

# Create rooms
sge.game.start_room = Room(background=background)

if __name__ == '__main__':
    sge.game.start()
```

CHAPTER

THREE

TUTORIAL 2: PONG

Contents

- Tutorial 2: Pong
  - Adding Game Logic
    - The Game Class
    - The Object Classes
      - Player
      - Ball
  - Starting the Game
    - Loading Sprites
    - Loading Backgrounds
    - Creating Objects
    - Creating Rooms
    - Making the Mouse Invisible
    - Starting the Game
  - The Final Result

Now that you've seen the basics of the SGE, it's time to create an actual game. Although Pong might seem extremely simple, it will give you a great foundation for developing more complex games in the future.

Start out by setting up the project like we did in the Hello World tutorial.

3.1 Adding Game Logic

3.1.1 The Game Class

For our `sge.dsp.Game` class, we want to of course provide a way to exit the game, and in this case, we are also going to provide a way to pause the game. Just for the heck of it, let's also allow the player to take a screenshot by pressing F8 and toggle fullscreen by pressing F11.

Let's take it one event at a time. Our close event is simple enough:

```
def event_close(self):
    self.end()
```

Our key press event is slightly more involved. To take a screenshot, we simply use a combination of `sge.gfx.Sprite.from_screenshot()` and `sge.gfx.Sprite.save()`. To toggle fullscreen, we simply change the value of `sge.dsp.Game.fullscreen`. To pause the game, we use `sge.dsp.Game.pause()`. We end up with this:

```
def event_key_press(self, key, char):
    if key == 'f8':
        sge.gfx.Sprite.from_screenshot().save('screenshot.jpg')
    elif key == 'f11':
        self.fullscreen = not self.fullscreen
    elif key == 'escape':
        self.event_close()
    elif key in ('p', 'enter'):
        self.pause()
```

This is incomplete, though. When `sge.dsp.Game.pause()` is called, the game enters a special loop where normal events are ignored. In their place, we need to use "paused" events to give the player a chance to unpause. We also should allow the player to quit the game while it is paused. To achieve these goals, we add the special events, `sge.dsp.Game.event_paused_key_pressed()` and `sge.dsp.Game.event_paused_close()`:

```
def event_paused_key_press(self, key, char):
    if key == 'escape':
        # This allows the player to still exit while the game is
        # paused, rather than having to unpause first.
        self.event_close()
    else:
        self.unpause()

def event_paused_close(self):
    # This allows the player to still exit while the game is paused,
    # rather than having to unpause first.
    self.event_close()
```

In this case, we are defining the paused key press event to unpause the game when any key except for the Esc key is pressed.

3.1.2 The Object Classes

`sge.dsp.Object` objects are things in a game that we want to be displayed in a room. These objects tend to represent players, enemies, tiles, decorations, and pretty much anything else you can think of.

For Pong, we need three objects: the two players, and the ball. We will define two sub-classes of `sge.dsp.Object` for this purpose: `Player` and `Ball`.

Player

`Player` is used for the paddles. These are what the players control.

For `Player`, the difference between different objects is which player controls it. Every other difference (the position, the controls, and the direction it hits the ball) can be easily derived from that. We are therefore going to define `Player.__init__()` to reflect this.

`Player.__init__()` will take a single argument, `player`. This argument will indicate which player the object is for: `1` for player 1, or `2` for player 2. We will set a few attributes based on this:

- `up_key` will indicate the key that moves the paddle up. We will set it to `"w"` for player 1, or `"up"` for player 2.
- `down_key` will indicate the key that moves the paddle down. We will set it to `"s"` for player 1, or `"down"` for player 2.

- `x` is an attribute inherited from `sge.dsp.Object` which indicates the horizontal position of the object. We will set this based on a constant we will define (technically just a variable, since Python doesn't support constants) called `PADDLE_XOFFSET`: `PADDLE_XOFFSET` for player 1, or `sge.game.width - PADDLE_XOFFSET` for player 2. We will define `PADDLE_XOFFSET` near the top of our code file, beneath imports, as `32`.
- `hit_direction` will indicate the direction the paddle hits the ball. We will set it to `1` for player 1, and `-1` for player 2.

Additionally, certain attributes inherited from `sge.dsp.Object` will be the same for both `Player` objects. `y` will always be `sge.game.height / 2` (vertically centered). `sprite` will always be `paddle_sprite` (a sprite we will create later). `checks_collisions` will always be `False`, since player objects don't need to check for collisions with each other; we can therefore leave all collision checking to the ball object.

All attributes inherited from `sge.dsp.Object` will be defined by passing their values to `sge.dsp.Object.__init__()`, which we will call with `super().__init__(*args, **kwargs)`. This makes our `Player.__init__()` defintion an extension, rather than an override, of `sge.dsp.Object.__init__()`, which is important; overriding this method would be likely to break something.

Our definition of `Player.__init__`() ends up looking something like this:

```
def __init__(self, player):
    if player == 1:
        self.joystick = 0
        self.up_key = "w"
        self.down_key = "s"
        x = PADDLE_XOFFSET
        self.hit_direction = 1
    else:
        self.joystick = 1
        self.up_key = "up"
        self.down_key = "down"
        x = sge.game.width - PADDLE_XOFFSET
        self.hit_direction = -1

    y = sge.game.height / 2
    super().__init__(x, y, sprite=paddle_sprite, checks_collisions=False)
```

We need to allow the players to move the paddles. We could do this by using key press events, but since we would like the players to be able to continuously move the paddles by holding down the key, the proper way to do this is to check for the state of the keys every frame and move accordingly.

`sge.keyboard.get_pressed()` returns the state of a key on the keyboard. We will check this in the step event to decide how the paddle should move on any given frame. The step event, defined by `sge.dsp.Object.event_step()`, is an event which always executes every frame.

What we will do is subtract the state of `up_key` from the state of `down_key`. This will give us `-1` if only `up_key` is pressed, `1` if only `down_key` is pressed, and `0` if neither or both keys are pressed. We can multiply this result by a constant, which we will call `PADDLE_SPEED`, to get the amount that the paddle should move this frame, and assign this value to the player's `sge.dsp.Object.yvelocity`, an attribute which indicates the number of pixels an object will move vertically each frame. We will define `PADDLE_SPEED` as `4`.

This isn't quite enough, though. With just this, the paddle can be moved off-screen! To prevent this from happening, we will check the player object's `bbox_top` and `bbox_bottom` values; these indicate the current location of the object's bounding box. If `bbox_top` is less than `0`, we will set it to `0`. If `bbox_bottom` is greater than `sge.game.current_room.height`, we will set it to `sge.game.current_room.height`. `sge.game.current_room`, as its name implies, indicates the currently running `sge.game.Room` object.

Our step event ends up looking something like this:

```
def event_step(self, time_passed, delta_mult):
    # Movement
    key_motion = (sge.keyboard.get_pressed(self.down_key) -
                  sge.keyboard.get_pressed(self.up_key))

    self.yvelocity = key_motion * PADDLE_SPEED

    # Keep the paddle inside the window
    if self.bbox_top < 0:
        self.bbox_top = 0
    elif self.bbox_bottom > sge.game.current_room.height:
        self.bbox_bottom = sge.game.current_room.height
```

Ball

`Ball` is the ball. It is bounced back and forth by the players. If it touches the top or bottom edge of the screen, it bounces off. If it passes one of the players, the other player gets a point and the ball is returned to the playing field.

Any `Ball` object is always going to have the same initial attributes as any other `Ball` object, so much like what we did with `Player`, we are going to define a custom `Ball.__init__()`.

In this case, it's much simpler: `x` and `y` are going to start at the center of the screen, and `sprite` is going to be `ball_sprite`. These are attributes inherited from `sge.dsp.Object`, so we indicate them in a call to `super().__init__`. `Ball.__init__()` ends up as:

```
def __init__(self):
    x = sge.game.width / 2
    y = sge.game.height / 2
    super().__init__(x, y, sprite=ball_sprite)
```

Since we want to serve the ball both at the start of the game and every time the ball passes a player, we should define a `Ball.serve()` method. This method needs to do two things: first, it needs to return the ball to its original position in the center. Second, it needs to set the speed so that it moves either straight to the left or straight to the right. If a direction isn't specified, it needs to choose a direction at random.

For the first task, we can use `sge.dsp.Object.xstart` and `sge.dsp.Object.ystart`. These attributes indicate the original position of an object when it was first created, which in the case of `Ball` objects is in the center of the screen.

For the second task, we have an argument called `direction`. If it is `None`, it randomly becomes either `1` or `-1`. The value is then multiplied by a constant called `BALL_START_SPEED`, which we will set to `2`, and this becomes the ball's `sge.dsp.Object.xvelocity` value. The ball's `sge.dsp.Object.yvelocity` value is then set to `0`.

The result looks like this:

```
def serve(self, direction=None):
    if direction is None:
        direction = random.choice([-1, 1])

    self.x = self.xstart
    self.y = self.ystart

    # Next round
    self.xvelocity = BALL_START_SPEED * direction
    self.yvelocity = 0
```

Note: Since we are now using the `random` module, we need to also import it at the top of our code file.

When the ball is created, we want to serve it immediately. we will put this in the create event, which is defined by `sge.dsp.Object.event_create()`. The create event happens whenever the object is created in the room. This is the create event of `Ball`:

```
def event_create(self):
    self.serve()
```

For `Ball`'s step event, we need to do two things: cause the ball to bounce off of the top and bottom edges of the screen, and serve the ball when it passes the left or right edge of the screen.

For the first task, we do the same thing we did with `Player`, but we also set whether `yvelocity` is positive or negative; we make it negative when the ball touches the bottom, and positive when the ball touches the top.

For the second task, we do a similar check, but we phrase the check such that the ball needs to be completely outside of the room, rather than just touching the edge. We do this by checking `bbox_right` against the left edge, and `bbox_left` against the right edge. When the ball is outside the screen, we serve it in the direction of the player it passed (so that the player who lost the round gets initial control of the ball).

Our step event for `Ball` ends up looking something like this:

```
def event_step(self, time_passed, delta_mult):
    # Scoring
    if self.bbox_right < 0:
        self.serve(-1)
    elif self.bbox_left > sge.game.current_room.width:
        self.serve(1)

    # Bouncing off of the edges
    if self.bbox_bottom > sge.game.current_room.height:
        self.bbox_bottom = sge.game.current_room.height
        self.yvelocity = -abs(self.yvelocity)
    elif self.bbox_top < 0:
        self.bbox_top = 0
        self.yvelocity = abs(self.yvelocity)
```

Now, we need to allow the players to repel the ball. We will do this with a collision event. Collision events, controlled by `sge.dsp.Object.event_collision()`, occur when two objects touch each other.

We first need to verify what type of object we're colliding with. The most straightforward way is to use `isinstance()` to check whether or not the object being collided with, which is passed on to the `other` argument, is an instance of `Player`. We write the collision code for these two objects under this check.

The most straightforward way to do this is with directional collision detection, but we are going to instead use `Player.hit_direction` to determine what to do. If the `other.hit_direction` is `1`, we bounce the ball to the right. Otherwise, we bounce the ball to the left.

We need to make the ball accelerate each time the ball hits a paddle, so that the round goes faster over time. We will store the amount of acceleration in a constant called `BALL_ACCELERATION`, which we will define as `0.2`. We will then set `self.xvelocity` to `(abs(self.xvelocity) + BALL_ACCELERATION) * other.hit_direction`.

We also need to make the ball's vertical movement change based on where it hits the paddle. To do this, we will subtract `other.y` from `self.y` and multiply that by a constant called `PADDLE_VERTICAL_FORCE`, which we will define as `1 / 12`; this value will be added to `self.yvelocity`.

There is one problem left, though it is not particularly obvious. The way we have it set up at this point, the ball will eventually move so fast that it will fail to collide with the paddles. This is due to how movement works; it's not actual movement, but rather a slight change of position done every frame. If that change of position is too much, the ball can pass right over a paddle.

To prevent this, we need to set a limit for how fast the ball can move horizontally. Instead of just multiplying `(abs(self.xvelocity) + BALL_ACCELERATION)` by `other.hit_direction`, we multiply the smallest out of that, and a new constant called `BALL_MAX_SPEED`, by `other.hit_direction`. We will define `BALL_MAX_SPEED` as `15`.

Our collision event ends up looking something like this:

```
def event_collision(self, other, xdirection, ydirection):
    if isinstance(other, Player):
        if other.hit_direction == 1:
            self.bbox_left = other.bbox_right + 1
        else:
            self.bbox_right = other.bbox_left - 1

        self.xvelocity = min(abs(self.xvelocity) + BALL_ACCELERATION,
                             BALL_MAX_SPEED) * other.hit_direction
        self.yvelocity += (self.y - other.y) * PADDLE_VERTICAL_FORCE
```

3.2 Starting the Game

It's time to get our game started.

We are going to pass some arguments to the creation of our `Game` object: we are going to define `width` as `640`, `height` as `480`, `fps` as `120`, and `window_text` as `"Pong"`. Specify them as keyword arguments.

3.2.1 Loading Sprites

We need two sprites: a paddle sprite and a ball sprite. We also need a black background with a line down the middle. We could draw these in an image editor and load them, but since they are so simple, we are going to generate them dynamically instead.

Sprites are stored as `sge.gfx.Sprite` objects, so we are going to create two of them:

```
paddle_sprite = sge.gfx.Sprite(width=8, height=48, origin_x=4, origin_y=24)
ball_sprite = sge.gfx.Sprite(width=8, height=8, origin_x=4, origin_y=4)
```

`sge.gfx.Sprite.origin_x` and `sge.gfx.Sprite.origin_y` indicate the origin of the sprite. In this case, we are setting the origins to the center of the sprites. This is necessary for our method of determining how the paddles affect vertical speed to work, and it also makes symmetry easier.

Currently, both of these sprites are blank. We need to draw the images on them. In this case, we will just draw white rectangles that fill the entirety of the sprites, which can be done with `sge.gfx.Sprite.draw_rectangle()`:

```
paddle_sprite.draw_rectangle(0, 0, paddle_sprite.width,
                             paddle_sprite.height, fill=sge.gfx.Color("white"))
ball_sprite.draw_rectangle(0, 0, ball_sprite.width, ball_sprite.height,
                           fill=sge.gfx.Color("white"))
```

3.2.2 Loading Backgrounds

Now we need a background. Our sprites are white, so we need a black background. We could of course leave it just at that, but that would be boring, so we are also going to also have a white line in the middle. We can do this easily by using the paddle sprite as a background layer. Background layers are special objects that indicate sprites that are used in a background. We create the layer, put it in a list, and pass that list onto `sge.gfx.Background.__init__()`'s `layers` argument:

```
layers = [sge.gfx.BackgroundLayer(paddle_sprite, sge.game.width / 2, 0, -10000,
                                  repeat_up=True, repeat_down=True)]
background = sge.gfx.Background(layers, sge.gfx.Color("black"))
```

The fourth argument of `sge.BackgroudLayer.__init__()` is the layer's Z-axis value. The Z-axis is used to determine what objects are in front of what other objects; objects with a higher Z-axis value are closer to the viewer. The default Z-axis value is `0`. Since we want all objects to be in front of the layer, we set its Z-axis value to a very low negative value.

3.2.3 Creating Objects

Don't forget to create our objects! In `player1`, store a `Player` object with the `player` argument specified as `1`. In `player2`, store a `Player` object with the `player` argument specified as `2`. Finally, create a `Ball` object and store it in `ball`. Put all of these objects in a list and assign this list to a variable called `objects`.

3.2.4 Creating Rooms

Create a `Room` object. Specify the first argument as `objects`, and specify the keyword argument `background` as `background`. Don't forget to assign it to `sge.game.start_room`!

3.2.5 Making the Mouse Invisible

Since we don't need to see the mouse cursor, we will hide it. To do this, set `sge.game.mouse.visible` to `False`.

3.2.6 Starting the Game

Add a call to `sge.game.start()` at the end, under a check for the value of `__name__`.

3.3 The Final Result

You should now have a script that looks something like this:

```
#!/usr/bin/env python3

# Pong Example
# Written in 2013, 2014 by Julian Marchant <onpon4@riseup.net>
#
# To the extent possible under law, the author(s) have dedicated all
# copyright and related and neighboring rights to this software to the
# public domain worldwide. This software is distributed without any
# warranty.
#
# You should have received a copy of the CC0 Public Domain Dedication
# along with this software. If not, see
# <http://creativecommons.org/publicdomain/zero/1.0/>.

import random

import sge
```

```
PADDLE_XOFFSET = 32
PADDLE_SPEED = 4
PADDLE_VERTICAL_FORCE = 1 / 12
BALL_START_SPEED = 2
BALL_ACCELERATION = 0.2
BALL_MAX_SPEED = 15


class Game(sge.dsp.Game):

    def event_key_press(self, key, char):
        global game_in_progress

        if key == 'f8':
            sge.gfx.Sprite.from_screenshot().save('screenshot.jpg')
        elif key == 'f11':
            self.fullscreen = not self.fullscreen
        elif key == 'escape':
            self.event_close()
        elif key in ('p', 'enter'):
            self.pause()

    def event_close(self):
        self.end()

    def event_paused_key_press(self, key, char):
        if key == 'escape':
            # This allows the player to still exit while the game is
            # paused, rather than having to unpause first.
            self.event_close()
        else:
            self.unpause()

    def event_paused_close(self):
        # This allows the player to still exit while the game is paused,
        # rather than having to unpause first.
        self.event_close()


class Player(sge.dsp.Object):

    def __init__(self, player):
        if player == 1:
            self.up_key = "w"
            self.down_key = "s"
            x = PADDLE_XOFFSET
            self.hit_direction = 1
        else:
            self.up_key = "up"
            self.down_key = "down"
            x = sge.game.width - PADDLE_XOFFSET
            self.hit_direction = -1

        y = sge.game.height / 2
        super().__init__(x, y, sprite=paddle_sprite, checks_collisions=False)

    def event_step(self, time_passed, delta_mult):
        # Movement
```

```
        key_motion = (sge.keyboard.get_pressed(self.down_key) -
                      sge.keyboard.get_pressed(self.up_key))

        self.yvelocity = key_motion * PADDLE_SPEED

        # Keep the paddle inside the window
        if self.bbox_top < 0:
            self.bbox_top = 0
        elif self.bbox_bottom > sge.game.current_room.height:
            self.bbox_bottom = sge.game.current_room.height


class Ball(sge.dsp.Object):

    def __init__(self):
        x = sge.game.width / 2
        y = sge.game.height / 2
        super().__init__(x, y, sprite=ball_sprite)

    def event_create(self):
        self.serve()

    def event_step(self, time_passed, delta_mult):
        # Scoring
        if self.bbox_right < 0:
            self.serve(-1)
        elif self.bbox_left > sge.game.current_room.width:
            self.serve(1)

        # Bouncing off of the edges
        if self.bbox_bottom > sge.game.current_room.height:
            self.bbox_bottom = sge.game.current_room.height
            self.yvelocity = -abs(self.yvelocity)
        elif self.bbox_top < 0:
            self.bbox_top = 0
            self.yvelocity = abs(self.yvelocity)

    def event_collision(self, other, xdirection, ydirection):
        if isinstance(other, Player):
            if other.hit_direction == 1:
                self.bbox_left = other.bbox_right + 1
            else:
                self.bbox_right = other.bbox_left - 1

            self.xvelocity = min(abs(self.xvelocity) + BALL_ACCELERATION,
                                 BALL_MAX_SPEED) * other.hit_direction
            self.yvelocity += (self.y - other.y) * PADDLE_VERTICAL_FORCE

    def serve(self, direction=None):
        if direction is None:
            direction = random.choice([-1, 1])

        self.x = self.xstart
        self.y = self.ystart

        # Next round
        self.xvelocity = BALL_START_SPEED * direction
        self.yvelocity = 0
```

```
# Create Game object
Game(width=640, height=480, fps=120, window_text="Pong")

# Load sprites
paddle_sprite = sge.gfx.Sprite(width=8, height=48, origin_x=4, origin_y=24)
ball_sprite = sge.gfx.Sprite(width=8, height=8, origin_x=4, origin_y=4)
paddle_sprite.draw_rectangle(0, 0, paddle_sprite.width, paddle_sprite.height,
                             fill=sge.gfx.Color("white"))
ball_sprite.draw_rectangle(0, 0, ball_sprite.width, ball_sprite.height,
                           fill=sge.gfx.Color("white"))

# Load backgrounds
layers = [sge.gfx.BackgroundLayer(paddle_sprite, sge.game.width / 2, 0, -10000,
                                  repeat_up=True, repeat_down=True)]
background = sge.gfx.Background(layers, sge.gfx.Color("black"))

# Create objects
player1 = Player(1)
player2 = Player(2)
ball = Ball()
objects = [player1, player2, ball]

# Create rooms
sge.game.start_room = sge.dsp.Room(objects, background=background)

sge.game.mouse.visible = False


if __name__ == '__main__':
    sge.game.start()
```

This is a basically complete Pong game, but it lacks some features. First, this game doesn't keep track of the score. It is left up to the players to keep track of who is winning. Second, there is no sound. We should fix both of these problems.

Additionally, it would be nice if our game could support joystick input.

In the next tutorial, we will improve on these points to make a Pong game more on par with Atari's original Pong.

CHAPTER

FOUR

TUTORIAL 3: BETTER PONG

Contents

- Tutorial 3: Better Pong
  - Adding Scoring
    - Making Enter Restart the Game
    - Giving Points to the Players
    - Displaying the Scores
      - New Resources
      - Drawing the HUD
      - Displaying the HUD
    - Giving Victory
  - Adding Sounds
    - Getting the Sounds
    - Loading the Sounds
    - Playing the Sounds
  - Adding Joystick Support
    - Axis Movement
    - Trackball Movement
    - Applying the Joystick Controls
  - The Final Result

In the last tutorial, we made a simple Pong game that was kind of boring. We're going to make it better by adding scores, sounds, and joystick support.

4.1 Adding Scoring

Adding a score system will make our Pong game feel more like a game and less like a toy. Every time a player wins a round, they will get one point. When a player gets ten points, they will win the game, and a new game can be started by pressing the Enter key.

4.1.1 Making Enter Restart the Game

We are going to need a new global variable: `game_in_progress`. This variable will indicate whether or not a game is currently going and will be used to determine whether to start a new game or pause when the Enter key is pressed. Set it to `True` by default.

To make pressing Enter start a new game, we will check `game_in_progress`. If a game is in progress, we will pause the game, as we had it do previously. Otherwise, we will set `game_in_progress` to `True` and restart the

room.

If you look through the documentation for sge.dsp.Room, you may notice that no "restart" method exists. In fact, this is a design choice; earlier versions of the SGE did have a method to restart rooms, but it was removed because this feature is overly difficult to maintain properly. But how do we restart the room, then? Well, we technically don't. Instead, we create a new room which is exactly like the one we wanted to restart, and immediately start it. We will put the creation of the room into a new function, create_room(). Our definition of Game.event_key_press() becomes:

```
def event_key_press(self, key, char):
    global game_in_progress

    if key == 'f8':
        sge.gfx.Sprite.from_screenshot().save('screenshot.jpg')
    elif key == 'f11':
        self.fullscreen = not self.fullscreen
    elif key == 'escape':
        self.event_close()
    elif key in ('p', 'enter'):
        if game_in_progress:
            self.pause()
        else:
            game_in_progress = True
            create_room().start()
```

Now, we need to define create_room(). This is very simple; we just copy and paste the code we used at the bottom to create the room into it, but specify that player and player2 are global. Our function is as follows:

```
def create_room():
    global player1
    global player2
    player1 = Player(1)
    player2 = Player(2)
    ball = Ball()
    return sge.dsp.Room([player1, player2, ball], background=background)
```

Of course, this makes the identical code at the bottom redundant, so we will replace it with a call to create_room().

4.1.2 Giving Points to the Players

We now need to add score attributes to the Player objects. We will initialize the new attribute, score, in Player.event_create() as 0.

Now, in Ball.event_step(), add lines to increase player1.score and player2.score whenever the respective player wins a round.

4.1.3 Displaying the Scores

The players have points, but can't see the score! We need to add a HUD (heads-up display) to show the score to the players.

There are a couple of ways we can do this. Most obviously, we can use sge.dsp.Game.project_text() or sge.dsp.Room.project_text(). However, there is a much better way: have a dynamically generated sprite that represents the look of the HUD at any given time, and displaying that sprite.

New Resources

We need to add a new global variable called `hud_sprite`. Assign a new sprite to this variable with a `width` of `320`, a `height` of `120`, an `origin_x` of `160`, and an `origin_y` of `0`.

To draw text, we need a font. Create a new `sge.gfx.Font` object and assign it to `hud_font`. For now, we will use a system font. I am choosing `"Droid Sans Mono"`, but you can choose whatever font you prefer. Pass your choice as the first argument to `sge.gfx.Font.__init__()`. Set the `size` keyword argument to `48`.

Note: We are using system fonts for simplicity, but it is generally a bad idea to rely on them. There is no standard for what fonts are available on the system, and the set of fonts available on the system varies widely. In real projects, it is better to distribute a font file with the game and use that.

Drawing the HUD

There are a few times we need to redraw the HUD: when the game starts, when player 1 scores, and when player 2 scores. Therefore, we will put the redrawing code into a function, `refresh_hud()`. This function needs to clear the HUD sprite, draw Player 1's score, and then draw Player 2's score.

Another constant is needed: `TEXT_OFFSET`, which we will define as `16`.

We clear the HUD sprite with `sge.gfx.Sprite.draw_clear()`.

To draw the text, we use `sge.gfx.Sprite.draw_text()`. Both calls have a few arguments in common: `font` is set to `hud_font`, `y` is set to `TEXT_OFFSET`, `color` is set to white, and `valign` is set to `"top"`.

The rest of the arguments are different between the two. `text` is set to the respective player's score, converted to a string. `x` is set to `hud_sprite.width / 2 - TEXT_OFFSET` for player 1's score, and `hud_sprite.width / 2 + TEXT_OFFSET` for player 2's score. `halign` is set to `"right"` for player 1's score, and `"left"` for player 2's score.

`refresh_hud()` ends up something like this:

```
def refresh_hud():
    # This fixes the HUD sprite so that it displays the correct score.
    hud_sprite.draw_clear()
    x = hud_sprite.width / 2
    hud_sprite.draw_text(hud_font, str(player1.score), x - TEXT_OFFSET,
                         TEXT_OFFSET, color=sge.gfx.Color("white"),
                         halign="right", valign="top")
    hud_sprite.draw_text(hud_font, str(player2.score), x + TEXT_OFFSET,
                         TEXT_OFFSET, color=sge.gfx.Color("white"),
                         halign="left", valign="top")
```

Add calls to `refresh_hud()` in the three places where a `Player.score` value changes, right after the change. These places are in `Player.event_create()` and `Ball.event_step()`.

we have one more problem. `refresh_hud()` requires `player1` and `player2` to each have an attribute called `score`, but the first time it is called, one of the player objects has not had a chance to initialize this attribute. To work around this, add a class attribute to `Player` called `score`, and set it to `0`. This will cause `player1.score` and `player2.score` to be `0` in the event that the respective object's `score` has not been initialized yet.

Displaying the HUD

At this point, we have our HUD, but it isn't displayed. We will fix this simply by adding a step event to `Game` which projects the HUD sprite onto the screen:

```
def event_step(self, time_passed, delta_mult):
    self.project_sprite(hud_sprite, 0, self.width / 2, 0)
```

Unlike `sge.dsp.Room` projections, `sge.dsp.Game` projections are relative to the screen. Additionally, these projections are always on top of everything else on the screen. This is usually how we want a HUD to be displayed, which is why we are using a `sge.dsp.Game` projection instead of a `sge.dsp.Room` projection or `sge.dsp.Object` object.

Note: You may notice that, when you pause the game, the HUD disappears. This is *not* a bug! This happens because the step event doesn't occur while the game is paused. If you want the HUD to show up while the game is paused, project it in the paused step event, defined by `sge.dsp.Game.event_paused_step()`, as well.

4.1.4 Giving Victory

At this point, we have scores, but no one ever officially wins. We need to end the game when someone gets 10 points. We will go a little further and replace the scores with text that says "WIN" and "LOSE" for the winner and loser, respectively.

Define a new constant called `POINTS_TO_WIN` as `10`.

In our case, the most convenient place to check for victory is within `Ball.serve()`. Specifically, put the code that sets the speed of the ball under a conditional that checks whether the `score` values of both players are less than `POINTS_TO_WIN`. Add an `else` block below that. This is where a player has won the game.

Since the game is over, stop the movement of the ball by setting `xvelocity` and `yvelocity` to `0`. We don't want any more scoring to happen.

Now, draw the new text onto the HUD. We do this using the same call to `sge.gfx.Sprite.draw_text()` we used in `refresh_hud()`, except instead of drawing the scores converted to strings, we draw `"WIN"` or `"LOSE"` depending on whether or not the respective player's score is greater than the other player's score.

Finally, set `game_in_progress` to `False`. Don't forget to declare it with `global` first.

The new `Ball.serve()` looks something like this:

```
def serve(self, direction=None):
    global game_in_progress

    if direction is None:
        direction = random.choice([-1, 1])

    self.x = self.xstart
    self.y = self.ystart

    if (player1.score < POINTS_TO_WIN and
            player2.score < POINTS_TO_WIN):
        # Next round
        self.xvelocity = BALL_START_SPEED * direction
        self.yvelocity = 0
    else:
        # Game Over!
        self.xvelocity = 0
        self.yvelocity = 0
        hud_sprite.draw_clear()
        x = hud_sprite.width / 2
        p1text = "WIN" if player1.score > player2.score else "LOSE"
        p2text = "WIN" if player2.score > player1.score else "LOSE"
```

```
        hud_sprite.draw_text(hud_font, p1text, x - TEXT_OFFSET,
                             TEXT_OFFSET, color=sge.gfx.Color("white"),
                             halign="right", valign="top")
        hud_sprite.draw_text(hud_font, p2text, x + TEXT_OFFSET,
                             TEXT_OFFSET, color=sge.gfx.Color("white"),
                             halign="left", valign="top")
        game_in_progress = False
```

4.2 Adding Sounds

We have a complete Pong game now, but it's still a little quiet. Let's make it more lively by adding some sounds.

4.2.1 Getting the Sounds

I would normally go to a database like OpenGameArt for sound effects, but in this case, we are instead going to use a nice free/libre program called Sfxr. This program makes it easy to generate retro-sounding sound effects, so it's perfect for Pong sounds. Generate three sounds: one for the ball bouncing off a paddle ("bounce.wav"), one for the ball bouncing off a wall ("bounce_wall.wav"), and one for the ball passing by a player ("score.wav"). Alternatively, you can copy the sounds I generated from examples/data. Create a folder in your project directory with the name "data" and put your sounds in this folder.

Note: Some file systems, like FAT32 and NTFS, are case-insensitive and will allow you to treat "bounce.wav" and "Bounce.wav" as if they are the same file name, but some, such as pretty much every Linux file system, are case-sensitive, meaning that "bounce.wav" and "Bounce.wav" are two completely different names; requesting one will never give you the other. If you have a case-insensitive file system, be careful to not get the case wrong, or some people who play the game will face a crash that will be completely invisible to you!

4.2.2 Loading the Sounds

Sounds in the SGE are stored in `sge.snd.Sound` objects. As the only argument, indicate the full path to the file. There are two ways to indicate the path: using the current working directory as a base, and using the directory of pong.py as a base. Both of methods require the `os` module, so be sure to add this to your list of imports.

The easiest way to get the path of the file is to use the current working directory as a base, on the assumption that the current working directory is also the directory that the "data" folder is located in. This method is very simple; assuming we want the file called "spam.wav", we would use this code:

```
os.path.join("data", "spam.wav")
```

However, it is not always the case that the current working directory is the appropriate location to search for the "data" folder. It could be that the current working directory is the user's home directory, for instance. To prevent the game from crashing in this case, define a constant called `DATA`, indicating the "data" directory relative to the location of pong.py:

```
DATA = os.path.join(os.path.dirname(__file__), "data")
```

`__file__` is a special variable indicating the full path to the current file, i.e. pong.py in this case. By getting the directory name of the current file, we can be certain of where to look for the "data" folder. `DATA` now indicates the appropriate path to the "data" folder, so from now on, if we want a file called "spam.wav" located in this directory, we use this code:

```
os.path.join(DATA, "spam.wav")
```

Assign the appropriate `sge.snd.Sound` objects to `bounce_sound`, `bounce_wall_sound`, and `score_sound`.

4.2.3 Playing the Sounds

Sounds are played with `sge.snd.Sound.play()`. Call this method in the appropriate places: when a player scores, when the ball bounces off an edge of the screen, and when the ball hits a paddle. There are five places in total.

With that, our Pong game now has sound effects.

4.3 Adding Joystick Support

Joystick support is a nice thing to have in a game, so we are going to add it. We are going to support analog sticks and trackballs. Mouse control would actually be even better, but this would put one of the players at an unfair advantage.

First, we will add an attribute to `Player` indicating what joystick to use, called `joystick`. Set it to `0` (which is the first joystick) for player 1, and `1` (which is the second joystick) for player 2.

4.3.1 Axis Movement

Adding movement based on a joystick axis is easy. For this, we use `sge.joystick.get_axis()` in the step event of `Player`. Pass `self.joystick` as the first argument, and `1` (which is the Y-axis) as the second argument. Assign it to a variable called `axis_motion`. Later, we will be modifying the code that sets `yvelocity` so that it is chosen based on axis position, trackball movement, or key presses, whichever one would cause it to move fastest.

4.3.2 Trackball Movement

Since trackball motion is relative, it is a little trickier. We need to store the amount of movement it makes each frame. We will use an attribute called `trackball_motion` for that; initialize it as `0` in the create event.

We now need to define the trackball move event, which is defined by `sge.dsp.Object.event_joystick_trackball_move()`. Within this event, if the `joystick` argument is the same as `self.joystick`, add `y` to `self.trackball_motion`. We are adding to it, rather than replacing it, because the trackball might move multiple times in the same frame.

4.3.3 Applying the Joystick Controls

Currently, we have this line:

```
self.yvelocity = key_motion * PADDLE_SPEED
```

This line uses the state of the keys to determine how to move the paddle. We need to change this so that the joystick controls we defined can be used as well. It will be replaced with the following:

- If the absolute value of `axis_motion` is greater than the absolute value of both `key_motion` and `trackball_motion`, set `yvelocity` to `axis_motion * PADDLE_SPEED`.
- Otherwise, if `trackball_motion` is greater than `key_motion`, set `yvelocity` to `self.trackball_motion * PADDLE_SPEED`

- Otherwise, use the line we have been using up until this point.

After this, we must set `trackball_motion` to `0`.

4.4 The Final Result

Our final Pong game now has scores, sounds, and even joystick support:

```
#!/usr/bin/env python3

# Pong Example
# Written in 2013, 2014, 2015 by Julian Marchant <onpon4@riseup.net>
#
# To the extent possible under law, the author(s) have dedicated all
# copyright and related and neighboring rights to this software to the
# public domain worldwide. This software is distributed without any
# warranty.
#
# You should have received a copy of the CC0 Public Domain Dedication
# along with this software. If not, see
# <http://creativecommons.org/publicdomain/zero/1.0/>.

import random

import sge

DATA = os.path.join(os.path.dirname(__file__), "data")
PADDLE_XOFFSET = 32
PADDLE_SPEED = 4
PADDLE_VERTICAL_FORCE = 1 / 12
BALL_START_SPEED = 2
BALL_ACCELERATION = 0.2
BALL_MAX_SPEED = 15
POINTS_TO_WIN = 10
TEXT_OFFSET = 16

game_in_progress = True


class Game(sge.dsp.Game):

    def event_step(self, time_passed, delta_mult):
        self.project_sprite(hud_sprite, 0, self.width / 2, 0)

    def event_key_press(self, key, char):
        global game_in_progress

        if key == 'f8':
            sge.gfx.Sprite.from_screenshot().save('screenshot.jpg')
        elif key == 'f11':
            self.fullscreen = not self.fullscreen
        elif key == 'escape':
            self.event_close()
        elif key in ('p', 'enter'):
            if game_in_progress:
                self.pause()
            else:
```

```
                game_in_progress = True
                self.current_room.start()

    def event_close(self):
        self.end()

    def event_paused_key_press(self, key, char):
        if key == 'escape':
            # This allows the player to still exit while the game is
            # paused, rather than having to unpause first.
            self.event_close()
        else:
            self.unpause()

    def event_paused_close(self):
        # This allows the player to still exit while the game is paused,
        # rather than having to unpause first.
        self.event_close()


class Player(sge.dsp.Object):

    score = 0

    def __init__(self, player):
        if player == 1:
            self.joystick = 0
            self.up_key = "w"
            self.down_key = "s"
            x = PADDLE_XOFFSET
            self.hit_direction = 1
        else:
            self.joystick = 1
            self.up_key = "up"
            self.down_key = "down"
            x = sge.game.width - PADDLE_XOFFSET
            self.hit_direction = -1

        y = sge.game.height / 2
        super().__init__(x, y, sprite=paddle_sprite, checks_collisions=False)

    def event_create(self):
        self.score = 0
        refresh_hud()
        self.trackball_motion = 0

    def event_step(self, time_passed, delta_mult):
        # Movement
        key_motion = (sge.keyboard.get_pressed(self.down_key) -
                      sge.keyboard.get_pressed(self.up_key))
        axis_motion = sge.joystick.get_axis(self.joystick, 1)

        if (abs(axis_motion) > abs(key_motion) and
                abs(axis_motion) > abs(self.trackball_motion)):
            self.yvelocity = axis_motion * PADDLE_SPEED
        elif abs(self.trackball_motion) > abs(key_motion):
            self.yvelocity = self.trackball_motion * PADDLE_SPEED
        else:
```

```
            self.yvelocity = key_motion * PADDLE_SPEED

        self.trackball_motion = 0

        # Keep the paddle inside the window
        if self.bbox_top < 0:
            self.bbox_top = 0
        elif self.bbox_bottom > sge.game.current_room.height:
            self.bbox_bottom = sge.game.current_room.height

    def event_joystick_trackball_move(self, joystick, ball, x, y):
        if joystick == self.joystick:
            self.trackball_motion += y


class Ball(sge.dsp.Object):

    def __init__(self):
        x = sge.game.width / 2
        y = sge.game.height / 2
        super().__init__(x, y, sprite=ball_sprite)

    def event_create(self):
        self.serve()

    def event_step(self, time_passed, delta_mult):
        # Scoring
        if self.bbox_right < 0:
            player2.score += 1
            refresh_hud()
            score_sound.play()
            self.serve(-1)
        elif self.bbox_left > sge.game.current_room.width:
            player1.score += 1
            refresh_hud()
            score_sound.play()
            self.serve(1)

        # Bouncing off of the edges
        if self.bbox_bottom > sge.game.current_room.height:
            self.bbox_bottom = sge.game.current_room.height
            self.yvelocity = -abs(self.yvelocity)
            bounce_wall_sound.play()
        elif self.bbox_top < 0:
            self.bbox_top = 0
            self.yvelocity = abs(self.yvelocity)
            bounce_wall_sound.play()

    def event_collision(self, other, xdirection, ydirection):
        if isinstance(other, Player):
            if other.hit_direction == 1:
                self.bbox_left = other.bbox_right + 1
            else:
                self.bbox_right = other.bbox_left - 1

            self.xvelocity = min(abs(self.xvelocity) + BALL_ACCELERATION,
                                 BALL_MAX_SPEED) * other.hit_direction
            self.yvelocity += (self.y - other.y) * PADDLE_VERTICAL_FORCE
```

```
            bounce_sound.play()

    def serve(self, direction=None):
        global game_in_progress

        if direction is None:
            direction = random.choice([-1, 1])

        self.x = self.xstart
        self.y = self.ystart

        if (player1.score < POINTS_TO_WIN and
                player2.score < POINTS_TO_WIN):
            # Next round
            self.xvelocity = BALL_START_SPEED * direction
            self.yvelocity = 0
        else:
            # Game Over!
            self.xvelocity = 0
            self.yvelocity = 0
            hud_sprite.draw_clear()
            x = hud_sprite.width / 2
            p1text = "WIN" if player1.score > player2.score else "LOSE"
            p2text = "WIN" if player2.score > player1.score else "LOSE"
            hud_sprite.draw_text(hud_font, p1text, x - TEXT_OFFSET,
                                 TEXT_OFFSET, color=sge.gfx.Color("white"),
                                 halign="right", valign="top")
            hud_sprite.draw_text(hud_font, p2text, x + TEXT_OFFSET,
                                 TEXT_OFFSET, color=sge.gfx.Color("white"),
                                 halign="left", valign="top")
            game_in_progress = False


def create_room():
    global player1
    global player2
    player1 = Player(1)
    player2 = Player(2)
    ball = Ball()
    return sge.dsp.Room([player1, player2, ball], background=background)


def refresh_hud():
    # This fixes the HUD sprite so that it displays the correct score.
    hud_sprite.draw_clear()
    x = hud_sprite.width / 2
    hud_sprite.draw_text(hud_font, str(player1.score), x - TEXT_OFFSET,
                         TEXT_OFFSET, color=sge.gfx.Color("white"),
                         halign="right", valign="top")
    hud_sprite.draw_text(hud_font, str(player2.score), x + TEXT_OFFSET,
                         TEXT_OFFSET, color=sge.gfx.Color("white"),
                         halign="left", valign="top")


# Create Game object
Game(width=640, height=480, fps=120, window_text="Pong")

# Load sprites
```

```
paddle_sprite = sge.gfx.Sprite(width=8, height=48, origin_x=4, origin_y=24)
ball_sprite = sge.gfx.Sprite(width=8, height=8, origin_x=4, origin_y=4)
paddle_sprite.draw_rectangle(0, 0, paddle_sprite.width, paddle_sprite.height,
                             fill=sge.gfx.Color("white"))
ball_sprite.draw_rectangle(0, 0, ball_sprite.width, ball_sprite.height,
                           fill=sge.gfx.Color("white"))
hud_sprite = sge.gfx.Sprite(width=320, height=120, origin_x=160, origin_y=0)

# Load backgrounds
layers = [sge.gfx.BackgroundLayer(paddle_sprite, sge.game.width / 2, 0, -10000,
                                  repeat_up=True, repeat_down=True)]
background = sge.gfx.Background(layers, sge.gfx.Color("black"))

# Load fonts
hud_font = sge.gfx.Font("Droid Sans Mono", size=48)

# Load sounds
bounce_sound = sge.snd.Sound(os.path.join(DATA, 'bounce.wav'))
bounce_wall_sound = sge.snd.Sound(os.path.join(DATA, 'bounce_wall.wav'))
score_sound = sge.snd.Sound(os.path.join(DATA, 'score.wav'))

# Create rooms
sge.game.start_room = create_room()

sge.game.mouse.visible = False


if __name__ == '__main__':
    sge.game.start()
```

CHAPTER

FIVE

SGE.INPUT

Contents

- sge.input
  - Input Event Classes

This module provides input event classes. Input event objects are used to consolidate all necessary information about input events in a clean way.

You normally don't need to use input event objects directly. Input events are handled automatically in each frame of the SGE's main loop. You only need to use input event objects directly if you take control away from the SGE's main loop, e.g. to create your own loop.

5.1 Input Event Classes

class `sge.input.`**`KeyPress`**(*key*, *char*)

This input event represents a key on the keyboard being pressed.

`key`

The identifier string of the key that was pressed. See the table below.

`char`

The Unicode character associated with the key press, or an empty Unicode string if no Unicode character is associated with the key press. See the table below.

Key Name	Identifier String	Unicode Character
0	`"0"`	`"0"`
1	`"1"`	`"1"`
2	`"2"`	`"2"`
3	`"3"`	`"3"`
4	`"4"`	`"4"`
5	`"5"`	`"5"`
6	`"6"`	`"6"`
7	`"7"`	`"7"`
8	`"8"`	`"8"`
9	`"9"`	`"9"`
A	`"a"`	`"a"`
B	`"b"`	`"b"`
C	`"c"`	`"c"`

Continued on next page

Table 5.1 – continued from previous page

Key Name	Identifier String	Unicode Character
D	"d"	"d"
E	"e"	"e"
F	"f"	"f"
G	"g"	"g"
H	"h"	"h"
I	"i"	"i"
J	"j"	"j"
K	"k"	"k"
L	"l"	"l"
M	"m"	"m"
N	"n"	"n"
O	"o"	"o"
P	"p"	"p"
Q	"q"	"q"
R	"r"	"r"
S	"s"	"s"
T	"t"	"t"
U	"u"	"u"
V	"v"	"v"
W	"w"	"w"
X	"x"	"x"
Y	"y"	"y"
Z	"z"	"z"
Period	"period"	"."
Comma	"comma"	","
Less Than	"less_than"	"<"
Greater Than	"greater_than"	">"
Forward Slash	"slash"	"/"
Question Mark	"question"	"?"
Apostrophe	"apostrophe"	"'"
Quotation Mark	"quote"	'"'
Colon	"colon"	":"
Semicolon	"semicolon"	";"
Exclamation Point	"exclamation"	"!"
At	"at"	"@"
Hash	"hash"	"#"
Dollar Sign	"dollar"	"$"
Carat	"carat"	"^"
Ampersand	"ampersand"	"&"
Asterisk	"asterisk"	"*"
Left Parenthesis	"parenthesis_left"	"("
Right Parenthesis	"parenthesis_right"	")"
Hyphen	"hyphen"	"-"
Underscore	"underscore"	"_"
Plus Sign	"plus"	"+"
Equals Sign	"equals"	"="
Left Bracket	"bracket_left"	"["
Right Bracket	"bracket_right"	"]"
Backslash	"backslash"	"\\"

Continued on next page

Table 5.1 – continued from previous page

Key Name	Identifier String	Unicode Character
Backtick	"backtick"	"`"
Euro	"euro"	"\u20ac"
Keypad 0	"kp_0"	"0"
Keypad 1	"kp_1"	"1"
Keypad 2	"kp_2"	"2"
Keypad 3	"kp_3"	"3"
Keypad 4	"kp_4"	"4"
Keypad 5	"kp_5"	"5"
Keypad 6	"kp_6"	"6"
Keypad 7	"kp_7"	"7"
Keypad 8	"kp_8"	"8"
Keypad 9	"kp_9"	"9"
Keypad Decimal Point	"kp_point"	"."
Keypad Plus	"kp_plus"	"+"
Keypad Minus	"kp_minus"	"-"
Keypad Multiply	"kp_multiply"	"*"
Keypad Divide	"kp_divide"	"/"
Keypad Equals	"kp_equals"	"="
Keypad Enter	"kp_enter"	"\n"
Left Arrow	"left"	""
Right Arrow	"right"	""
Up Arrow	"up"	""
Down Arrow	"down"	""
Home	"home"	""
End	"end"	""
Page Up	"pageup"	""
Page Down	"pagedown"	""
Tab	"tab"	"\t"
Space Bar	"space"	" "
Enter/Return	"enter"	"\n"
Backspace	"backspace"	"\b"
Delete	"delete"	""
Clear	"clear"	""
Left Shift	"shift_left"	""
Right Shift	"shift_right"	""
Left Ctrl	"ctrl_left"	""
Right Ctrl	"ctrl_right"	""
Left Alt	"alt_left"	""
Right Alt	"alt_right"	""
Left Super	"super_left"	""
Right Super	"super_right"	""
Mode	"mode"	""
Menu	"menu"	""
Caps Lock	"caps_lock"	""
Esc	"escape"	""
Num Lock	"num_lock"	""
Scroll Lock	"scroll_lock"	""
Break	"break"	""
Insert	"insert"	""

Continued on next page

Table 5.1 – continued from previous page

Key Name	Identifier String	Unicode Character
Pause	`"pause"`	`""`
Power	`"power"`	`""`
Print Screen	`"print_screen"`	`""`
SysRq	`"sysrq"`	`""`
F1	`"f1"`	`""`
F2	`"f2"`	`""`
F3	`"f3"`	`""`
F4	`"f4"`	`""`
F5	`"f5"`	`""`
F6	`"f6"`	`""`
F7	`"f7"`	`""`
F8	`"f8"`	`""`
F9	`"f9"`	`""`
F10	`"f10"`	`""`
F11	`"f11"`	`""`
F12	`"f12"`	`""`

class `sge.input.`**`KeyRelease`**(*key*)
This input event represents a key on the keyboard being released.

`key`
The identifier string of the key that was released. See the table in the documentation for `sge.input.KeyPress`.

class `sge.input.`**`MouseMove`**(*x*, *y*)
This input event represents the mouse being moved.

`x`
The horizontal relative movement of the mouse.

`y`
The vertical relative movement of the mouse.

class `sge.input.`**`MouseButtonPress`**(*button*)
This input event represents a mouse button being pressed.

`button`
The identifier string of the mouse button that was pressed. See the table below.

Mouse Button Name	Identifier String
Left mouse button	`"left"`
Right mouse button	`"right"`
Middle mouse button	`"middle"`
Mouse wheel up	`"wheel_up"`
Mouse wheel down	`"wheel_down"`
Mouse wheel tilt left	`"wheel_left"`
Mouse wheel tilt right	`"wheel_right"`

class `sge.input.`**`MouseButtonRelease`**(*button*)
This input event represents a mouse button being released.

`button`
The identifier string of the mouse button that was released. See the table in the documentation for `sge.input.MouseButtonPress`.

class sge.input.**JoystickAxisMove**(*js_name*, *js_id*, *axis*, *value*)
This input event represents a joystick axis moving.

js_name
The name of the joystick.

js_id
The number of the joystick, where 0 is the first joystick.

axis
The number of the axis that moved, where 0 is the first axis on the joystick.

value
The tilt of the axis as a float from −1 to 1, where 0 is centered, −1 is all the way to the left or up, and 1 is all the way to the right or down.

class sge.input.**JoystickHatMove**(*js_name*, *js_id*, *hat*, *x*, *y*)
This input event represents a joystick hat moving.

js_name
The name of the joystick.

js_id
The number of the joystick, where 0 is the first joystick.

hat
The number of the hat that moved, where 0 is the first axis on the joystick.

x
The horizontal position of the hat, where 0 is centered, −1 is left, and 1 is right.

y
The vertical position of the hat, where 0 is centered, −1 is up, and 1 is down.

class sge.input.**JoystickTrackballMove**(*js_name*, *js_id*, *ball*, *x*, *y*)
This input event represents a joystick trackball moving.

js_name
The name of the joystick.

js_id
The number of the joystick, where 0 is the first joystick.

ball
The number of the trackball that moved, where 0 is the first trackball on the joystick.

x
The horizontal relative movement of the trackball.

y
The vertical relative movement of the trackball.

class sge.input.**JoystickButtonPress**(*js_name*, *js_id*, *button*)
This input event represents a joystick button being pressed.

js_name
The name of the joystick.

js_id
The number of the joystick, where 0 is the first joystick.

button
The number of the button that was pressed, where 0 is the first button on the joystick.

class sge.input.**JoystickButtonRelease**(*js_name*, *js_id*, *button*)
This input event represents a joystick button being released.

js_name
The name of the joystick.

js_id
The number of the joystick, where 0 is the first joystick.

button
The number of the button that was released, where 0 is the first button on the joystick.

class sge.input.**KeyboardFocusGain**
This input event represents the game window gaining keyboard focus. Keyboard focus is normally needed for keyboard input to be received.

Note: On some window systems, such as the one used by Windows, no distinction is made between keyboard and mouse focus, but on some other window systems, such as the X Window System, a distinction is made: one window can have keyboard focus while another has mouse focus. Be careful to observe the difference; failing to do so may result in annoying bugs, and you won't notice these bugs if you are testing on a window manager that doesn't recognize the difference.

---

class sge.input.**KeyboardFocusLose**
This input event represents the game window losing keyboard focus. Keyboard focus is normally needed for keyboard input to be received.

Note: See the note in the documentation for sge.input.KeyboardFocusGain.

---

class sge.input.**MouseFocusGain**
This input event represents the game window gaining mouse focus. Mouse focus is normally needed for mouse input to be received.

Note: See the note in the documentation for sge.input.KeyboardFocusGain.

---

class sge.input.**MouseFocusLose**
This input event represents the game window losing mouse focus. Mouse focus is normally needed for mouse input to be received.

Note: See the note in the documentation for sge.input.KeyboardFocusGain.

---

class sge.input.**QuitRequest**
This input event represents the OS requesting for the program to close (e.g. when the user presses a "close" button on the window border).

CHAPTER

SIX

SGE.DSP

Contents

- sge.dsp
  - sge.dsp Classes
    - sge.dsp.Game
      - sge.dsp.Game Methods
      - sge.dsp.Game Event Methods
    - sge.dsp.Room
      - sge.dsp.Room Methods
      - sge.dsp.Room Event Methods
    - sge.dsp.View
      - sge.dsp.View Methods
    - sge.dsp.Object
      - sge.dsp.Object Methods
      - sge.dsp.Object Event Methods

This module provides classes related to the graphical display.

6.1 sge.dsp Classes

6.1.1 sge.dsp.Game

class `sge.dsp.`**`Game`**(*width=640*, *height=480*, *fullscreen=False*, *scale=None*, *scale_proportional=True*, *scale_smooth=False*, *fps=60*, *delta=False*, *delta_min=15*, *delta_max=None*, *grab_input=False*, *window_text=None*, *window_icon=None*, *collision_events_enabled=True*)

This class handles most parts of the game which operate on a global scale, such as global game events. Before anything else is done with the SGE, an object either of this class or of a class derived from it must be created.

When an object of this class is created, it is automatically assigned to `sge.game`.

Note: This class is designed to be used as a singleton. Do not create multiple `sge.dsp.Game` objects. Doing so is unsupported and may cause errors.

`width`
The width of the game's display.

`height`
The height of the game's display.

fullscreen
Whether or not the game should be in fullscreen mode.

scale
A number indicating a fixed scale factor (e.g. `1` for no scaling, `2` for doubled size). If set to `None` or `0`, scaling is automatic (causes the game to fit the window or screen).

If a fixed scale factor is defined and the game is in fullscreen mode, the scale factor multiplied by `width` and `height` is used to suggest what resolution to use.

scale_proportional
If set to `True`, scaling is always proportional. If set to `False`, the image will be distorted to completely fill the game window or screen. This has no effect unless `scale` is `None` or `0`.

scale_smooth
Whether or not a smooth scaling algorithm (as opposed to a simple scaling algorithm such as nearest-neighbor) should be used.

fps
The rate the game should run in frames per second.

Note: This is only the maximum; if the computer is not fast enough, the game may run more slowly.

---

delta
Whether or not delta timing should be used. Delta timing affects object speeds, animation rates, and alarms.

delta_min
Delta timing can cause the game to be choppy. This attribute limits this by pretending that the frame rate is never lower than this amount, resulting in the game slowing down like normal if it is.

delta_max
Indicates a higher frame rate cap than `fps` to allow the game to reach by using delta timing to slow object speeds, animation rates, and alarms down. If set to `None`, this feature is disabled and the game will not be permitted to run faster than `fps`.

This attribute has no effect unless `delta` is `True`.

grab_input
Whether or not all input should be forcibly grabbed by the game. If this is `True` and `sge.mouse.visible` is `False`, the mouse will be in relative mode. Otherwise, the mouse will be in absolute mode.

window_text
The text for the OS to display as the window title, e.g. in the frame of the window. If set to `None`, the SGE chooses the text.

window_icon
The path to the image file to use as the window icon. If set to `None`, the SGE chooses the icon. If the file specified does not exist, `IOError` is raised.

collision_events_enabled
Whether or not collision events should be executed. Setting this to `False` will improve performence if collision events are not needed.

alarms
A dictionary containing the global alarms of the game. Each value decreases by 1 each frame (adjusted for delta timing if it is enabled). When a value is at or below 0, `sge.dsp.Game.event_alarm()` is executed with `alarm_id` set to the respective key, and the item is deleted from this dictionary.

input_events
A list containing all input event objects which have not yet been handled, in the order in which they occurred.

Note: If you handle input events manually, be sure to delete them from this list, preferably by getting them with `list.pop()`. Otherwise, the event will be handled more than once, which is usually not what you want.

---

start_room
The room which becomes active when the game first starts and when it restarts. Must be set exactly once, before the game first starts, and should not be set again afterwards.

current_room
The room which is currently active. (Read-only)

mouse
A `sge.dsp.Object` object which represents the mouse cursor. Its bounding box is a one-pixel square. It is automatically added to every room's default list of objects.

Some of this object's attributes control properties of the mouse. See the documentation for `sge.mouse` for more information.

(Read-only)

sge.dsp.Game Methods

`Game.__init__`(*width=640*, *height=480*, *fullscreen=False*, *scale=None*, *scale_proportional=True*, *scale_smooth=False*, *fps=60*, *delta=False*, *delta_min=15*, *delta_max=None*, *grab_input=False*, *window_text=None*, *window_icon=None*, *collision_events_enabled=True*)
Arguments set the respective initial attributes of the game. See the documentation for `sge.dsp.Game` for more information.

The created `sge.dsp.Game` object is automatically assigned to `sge.game`.

`Game.start`()
Start the game. Should only be called once; the effect of any further calls is undefined.

`Game.end`()
Properly end the game.

`Game.pause`(*sprite=None*)
Pause the game.

Arguments:

- `sprite` – The sprite to show in the center of the screen while the game is paused. If set to `None`, the SGE chooses the image.

Normal events are not executed while the game is paused. Instead, events with the same name, but prefixed with `event_paused_` instead of `event_` are executed. Note that not all events have these alternative "paused" events associated with them.

`Game.unpause`()
Unpause the game.

`Game.pump_input`()
Cause the SGE to recieve input from the OS.

This method needs to be called periodically for the SGE to recieve events from the OS, such as key presses and mouse movement, as well as to assure the OS that the program is not locked up.

Upon calling this, each event is translated into the appropriate class in `sge.input` and the resulting object is appended to `input_events`.

You normally don't need to use this function directly. It is called automatically in each frame of the SGE's main loop. You only need to use this function directly if you take control away from the SGE's main loop, e.g. to create your own loop.

`Game.`**`regulate_speed`**(*fps=None*)
Regulate the SGE's running speed and return the time passed.

Arguments:

- `fps` – The target frame rate in frames per second. Set to `None` to target the current value of `fps`.

When this method is called, the program will sleep long enough so that the game runs at `fps` frames per second, then return the number of milliseconds that passed between the previous call and the current call of this method.

You normally don't need to use this function directly. It is called automatically in each frame of the SGE's main loop. You only need to use this function directly if you want to create your own loop.

`Game.`**`refresh`**()
Refresh the screen.

This method needs to be called for changes to the screen to be seen by the user. It should be called every frame.

You normally don't need to use this function directly. It is called automatically in each frame of the SGE's main loop. You only need to use this function directly if you take control away from the SGE's main loop, e.g. to create your own loop.

`Game.`**`project_dot`**(*x*, *y*, *color*, *z=0*)
Project a single-pixel dot onto the game window.

Arguments:

- `x` – The horizontal location relative to the window to project the dot.
- `y` – The vertical location relative to the window to project the dot.
- `z` – The Z-axis position of the projection in relation to other window projections.

Window projections are projections made directly onto the game window, independent of the room or any views.

Note: The Z-axis position of a window projection does not correlate with the Z-axis position of anything positioned within the room, such as room projections and `sge.dsp.Object` objects. Window projections are always positioned in front of such things.

See the documentation for `sge.gfx.Sprite.draw_dot()` for more information.

`Game.`**`project_line`**(*x1*, *y1*, *x2*, *y2*, *color*, *z=0*, *thickness=1*, *anti_alias=False*)
Project a line segment onto the game window.

Arguments:

- `x1` – The horizontal location relative to the window of the first endpoint of the projected line segment.
- `y1` – The vertical location relative to the window of the first endpoint of the projected line segment.
- `x2` – The horizontal location relative to the window of the second endpoint of the projected line segment.
- `y2` – The vertical location relative to the window of the second endpoint of the projected line segment.
- `z` – The Z-axis position of the projection in relation to other window projections.

See the documentation for `sge.gfx.Sprite.draw_line()` and `sge.dsp.Game.project_dot()` for more information.

`Game`.**`project_rectangle`**(*x*, *y*, *width*, *height*, *z=0*, *fill=None*, *outline=None*, *outline_thickness=1*)
Project a rectangle onto the game window.

Arguments:

- `x` – The horizontal location relative to the window to project the rectangle.
- `y` – The vertical location relative to the window to project the rectangle.
- `z` – The Z-axis position of the projection in relation to other window projections.

See the documentation for `sge.gfx.Sprite.draw_rectangle()` and `sge.dsp.Game.project_dot()` for more information.

`Game`.**`project_ellipse`**(*x*, *y*, *width*, *height*, *z=0*, *fill=None*, *outline=None*, *outline_thickness=1*, *anti_alias=False*)
Project an ellipse onto the game window.

Arguments:

- `x` – The horizontal location relative to the window to position the imaginary rectangle containing the ellipse.
- `y` – The vertical location relative to the window to position the imaginary rectangle containing the ellipse.
- `z` – The Z-axis position of the projection in relation to other window projections.
- `width` – The width of the ellipse.
- `height` – The height of the ellipse.
- `outline_thickness` – The thickness of the outline of the ellipse.
- `anti_alias` – Whether or not anti-aliasing should be used.

See the documentation for `sge.gfx.Sprite.draw_ellipse()` and `sge.dsp.Game.project_dot()` for more information.

`Game`.**`project_circle`**(*x*, *y*, *radius*, *z=0*, *fill=None*, *outline=None*, *outline_thickness=1*, *anti_alias=False*)
Project a circle onto the game window.

Arguments:

- `x` – The horizontal location relative to the window to position the center of the circle.
- `y` – The vertical location relative to the window to position the center of the circle.
- `z` – The Z-axis position of the projection in relation to other window projections.

See the documentation for `sge.gfx.Sprite.draw_circle()` and `sge.dsp.Game.project_dot()` for more information.

`Game`.**`project_polygon`**(*points*, *z=0*, *fill=None*, *outline=None*, *outline_thickness=1*, *anti_alias=False*)
Draw a polygon on the sprite.

Arguments:

- `points` – A list of points relative to the room to position each of the polygon's angles. Each point should be a tuple in the form `(x, y)`, where x is the horizontal location and y is the vertical location.
- `z` – The Z-axis position of the projection in relation to other window projections.

See the documentation for `sge.gfx.Sprite.draw_polygon()` and `sge.dsp.Game.project_dot()` for more information.

Game.**project_sprite**(*sprite*, *image*, *x*, *y*, *z=0*, *blend_mode=None*)
Project a sprite onto the game window.

Arguments:

- x – The horizontal location relative to the window to project sprite.
- y – The vertical location relative to the window to project sprite.
- z – The Z-axis position of the projection in relation to other window projections.

See the documentation for sge.gfx.Sprite.draw_sprite() and sge.dsp.Game.project_dot() for more information.

Game.**project_text**(*font*, *text*, *x*, *y*, *z=0*, *width=None*, *height=None*, *color=sge.gfx.Color("white")*, *halign='left'*, *valign='top'*, *anti_alias=True*)
Project text onto the game window.

Arguments:

- x – The horizontal location relative to the window to project the text.
- y – The vertical location relative to the window to project the text.
- z – The Z-axis position of the projection in relation to other window projections.

See the documentation for sge.gfx.Sprite.draw_text() and sge.dsp.Game.project_dot() for more information.

sge.dsp.Game Event Methods

Game.**event_step**(*time_passed*, *delta_mult*)
Called once each frame.

Arguments:

- time_passed – The number of milliseconds that have passed during the last frame.
- delta_mult – What speed and movement should be multiplied by this frame due to delta timing. If delta is False, this is always 1.

Game.**event_alarm**(*alarm_id*)
Called when the value of an alarm reaches 0.

See the documentation for sge.dsp.Game.alarms for more information.

Game.**event_key_press**(*key*, *char*)
See the documentation for sge.input.KeyPress for more information.

Game.**event_key_release**(*key*)
See the documentation for sge.input.KeyRelease for more information.

Game.**event_mouse_move**(*x*, *y*)
See the documentation for sge.input.MouseMove for more information.

Game.**event_mouse_button_press**(*button*)
See the documentation for sge.input.MouseButtonPress for more information.

Game.**event_mouse_button_release**(*button*)
See the documentation for sge.input.MouseButtonRelease for more information.

Game.**event_joystick_axis_move**(*js_name*, *js_id*, *axis*, *value*)
See the documentation for sge.input.JoystickAxisMove for more information.

Game.**event_joystick_hat_move**(*js_name*, *js_id*, *hat*, *x*, *y*)
See the documentation for sge.input.JoystickHatMove for more information.

Game.**event_joystick_trackball_move**(*js_name*, *js_id*, *ball*, *x*, *y*)
See the documentation for sge.input.JoystickTrackballMove for more information.

Game.**event_joystick_button_press**(*js_name*, *js_id*, *button*)
See the documentation for sge.input.JoystickButtonPress for more information.

Game.**event_joystick_button_release**(*js_name*, *js_id*, *button*)
See the documentation for sge.input.JoystickButtonRelease for more information.

Game.**event_gain_keyboard_focus**()
See the documentation for sge.input.KeyboardFocusGain for more information.

Game.**event_lose_keyboard_focus**()
See the documentation for sge.input.KeyboardFocusLose for more information.

Game.**event_gain_mouse_focus**()
See the documentation for sge.input.MouseFocusGain for more information.

Game.**event_lose_mouse_focus**()
See the documentation for sge.input.MouseFocusLose for more information.

Game.**event_close**()
See the documentation for sge.input.QuitRequest for more information.

This is always called after any sge.dsp.Room.event_close() occurring at the same time.

Game.**event_mouse_collision**(*other*, *xdirection*, *ydirection*)
Proxy for sge.game.mouse.event_collision(). See the documentation for sge.dsp.Object.event_collision() for more information.

Game.**event_paused_step**(*time_passed*, *delta_mult*)
See the documentation for sge.dsp.Game.event_step() for more information.

Game.**event_paused_key_press**(*key*, *char*)
See the documentation for sge.input.KeyPress for more information.

Game.**event_paused_key_release**(*key*)
See the documentation for sge.input.KeyRelease for more information.

Game.**event_paused_mouse_move**(*x*, *y*)
See the documentation for sge.input.MouseMove for more information.

Game.**event_paused_mouse_button_press**(*button*)
See the documentation for sge.input.MouseButtonPress for more information.

Game.**event_paused_mouse_button_release**(*button*)
See the documentation for sge.input.MouseButtonRelease for more information.

Game.**event_paused_joystick_axis_move**(*js_name*, *js_id*, *axis*, *value*)
See the documentation for sge.input.JoystickAxisMove for more information.

Game.**event_paused_joystick_hat_move**(*js_name*, *js_id*, *hat*, *x*, *y*)
See the documentation for sge.input.JoystickHatMove for more information.

Game.**event_paused_joystick_trackball_move**(*js_name*, *js_id*, *ball*, *x*, *y*)
See the documentation for sge.input.JoystickTrackballMove for more information.

Game.**event_paused_joystick_button_press**(*js_name*, *js_id*, *button*)
See the documentation for sge.input.JoystickButtonPress for more information.

Game.**event_paused_joystick_button_release**(*js_name*, *js_id*, *button*)
See the documentation for sge.input.JoystickButtonRelease for more information.

Game.**event_paused_gain_keyboard_focus**()
See the documentation for sge.input.KeyboardFocusGain for more information.

Game.**event_paused_lose_keyboard_focus**()
See the documentation for sge.input.KeyboardFocusLose for more information.

Game.**event_paused_gain_mouse_focus**()
See the documentation for sge.input.MouseFocusGain for more information.

Game.**event_paused_lose_mouse_focus**()
See the documentation for sge.input.MouseFocusLose for more information.

Game.**event_paused_close**()
See the documentation for sge.dsp.Game.event_close() for more information.

6.1.2 sge.dsp.Room

class sge.dsp.**Room**(*objects=()*, *width=None*, *height=None*, *views=None*, *background=None*, *background_x=0*, *background_y=0*, *object_area_width=None*, *object_area_height=None*)
This class stores the settings and objects found in a room. Rooms are used to create separate parts of the game, such as levels and menu screens.

width
The width of the room in pixels. If set to None, sge.game.width is used.

height
The height of the room in pixels. If set to None, sge.game.height is used.

views
A list containing all sge.dsp.View objects in the room.

background
The sge.gfx.Background object used.

background_x
The horizontal position of the background in the room.

background_y
The vertical position of the background in the room.

object_area_width
The width of this room's object areas in pixels. If set to None, sge.game.width is used. For optimum performance, this should generally be about the average width of objects in the room which check for collisions.

object_area_height
The height of this room's object areas in pixels. If set to None, sge.game.height is used. For optimum performance, this should generally be about the average height of objects in the room which check for collisions.

alarms
A dictionary containing the alarms of the room. Each value decreases by 1 each frame (adjusted for delta timing if it is enabled). When a value is at or below 0, event_alarm() is executed with alarm_id set to the respective key, and the item is deleted from this dictionary.

objects
A list containing all sge.dsp.Object objects in the room. (Read-only)

object_areas
A 2-dimensional list of object areas, indexed in the following way:

```
object_areas[x][y]
```

Where `x` is the horizontal location of the left edge of the area in the room divided by `object_area_width`, and `y` is the vertical location of the top edge of the area in the room divided by `object_area_height`.

For example, if `object_area_width` is `32` and `object_area_height` is `48`, then `object_areas[2][4]` indicates the object area with an x location of 64 and a y location of 192.

Each object area is a set containing `sge.dsp.Object` objects whose sprites or bounding boxes reside within the object area.

Object areas are only created within the room, i.e. the horizontal location of an object area will always be less than `width`, and the vertical location of an object area will always be less than `height`. Depending on the size of collision areas and the size of the room, however, the last row and/or the last column of collision areas may partially reside outside of the room.

Note: It is generally easier to use `get_objects_at()` than to access this list directly.

---

object_area_void
A set containing `sge.dsp.Object` objects whose sprites or bounding boxes reside within any area not covered by the room's object area.

Note: Depending on the size of object areas and the size of the room, the "void" area may not include the entirety of the outside of the room. There may be some space to the right of and/or below the room which is covered by collision areas.

---

rd
Reserved dictionary for internal use by the SGE. (Read-only)

sge.dsp.Room Methods

`Room.`**`__init__`**(*objects=()*, *width=None*, *height=None*, *views=None*, *background=None*, *background_x=0*, *background_y=0*, *object_area_width=None*, *object_area_height=None*)
Arguments:

- `views` – A list containing all `sge.dsp.View` objects in the room. If set to `None`, a new view will be created with `x=0`, `y=0`, and all other arguments unspecified, which will become the first view of the room.
- `background` – The `sge.gfx.Background` object used. If set to `None`, a new background will be created with no layers and the color set to black.

All other arguments set the respective initial attributes of the room. See the documentation for `sge.dsp.Room` for more information.

`Room.`**`add`**(*obj*)
Add an object to the room.

Arguments:

- `obj` – The `sge.dsp.Object` object to add.

Warning: This method modifies the contents of `objects`. Do not call this method during a loop through `objects`; doing so may cause problems with the loop. To get around this, you can create a shallow copy of `objects` to iterate through instead, e.g.:

```
for obj in self.objects[:]:
    self.add(obj.friend)
```

`Room.`**`remove`**(*obj*)

Remove an object from the room.

Arguments:

- `obj` – The `sge.dsp.Object` object to remove.

Warning: This method modifies the contents of `objects`. Do not call this method during a loop through `objects`; doing so may cause problems with the loop. To get around this, you can create a shallow copy of `objects` to iterate through instead, e.g.:

```
for obj in self.objects[:]:
    self.remove(obj)
```

`Room.`**`start`**(*transition=None*, *transition_time=1500*, *transition_arg=None*)

Start the room.

Arguments:

- `transition` – The type of transition to use. Should be one of the following:
  - `None` (no transition)
  - `"fade"` (fade to black)
  - `"dissolve"`
  - `"pixelate"`
  - `"wipe_left"` (wipe right to left)
  - `"wipe_right"` (wipe left to right)
  - `"wipe_up"` (wipe bottom to top)
  - `"wipe_down"` (wipe top to bottom)
  - `"wipe_upleft"` (wipe bottom-right to top-left)
  - `"wipe_upright"` (wipe bottom-left to top-right)
  - `"wipe_downleft"` (wipe top-right to bottom-left)
  - `"wipe_downright"` (wipe top-left to bottom-right)
  - `"wipe_matrix"`
  - `"iris_in"`
  - `"iris_out"`

  If an unsupported value is given, default to `None`.
- `transition_time` – The time the transition should take in milliseconds. Has no effect if `transition` is `None`.

- `transition_arg` – An arbitrary argument that can be used by the following transitions:
  - `"wipe_matrix"` – The size of each square in the matrix transition as a tuple in the form `(w, h)`, where `w` is the width and `h` is the height. Default is `(4, 4)`.
  - `"iris_in"` and `"iris_out"` – The position of the center of the iris as a tuple in the form `(x, y)`, where `x` is the horizontal location relative to the window and `y` is the vertical location relative to the window. Default is the center of the window.

Room.**get_objects_at**(*x*, *y*, *width*, *height*)
Return a set of objects near a particular area.

Arguments:

- `x` – The horizontal location relative to the room of the left edge of the area.
- `y` – The vertical location relative to the room of the top edge of the area.
- `width` – The width of the area in pixels.
- `height` – The height of the area in pixels.

Note: This function does not ensure that objects returned are actually *within* the given area. It simply combines all object areas that need to be checked into a single set. To ensure that an object is actually within the area, you must check the object manually, or use `sge.collision.rectangle()` instead.

Room.**project_dot**(*x*, *y*, *z*, *color*)
Project a single-pixel dot onto the room.

Arguments:

- `x` – The horizontal location relative to the room to project the dot.
- `y` – The vertical location relative to the room to project the dot.
- `z` – The Z-axis position of the projection in the room.

See the documentation for `sge.gfx.Sprite.draw_dot()` for more information.

Room.**project_line**(*x1*, *y1*, *x2*, *y2*, *z*, *color*, *thickness=1*, *anti_alias=False*)
Project a line segment onto the room.

Arguments:

- `x1` – The horizontal location relative to the room of the first endpoint of the projected line segment.
- `y1` – The vertical location relative to the room of the first endpoint of the projected line segment.
- `x2` – The horizontal location relative to the room of the second endpoint of the projected line segment.
- `y2` – The vertical location relative to the room of the second endpoint of the projected line segment.
- `z` – The Z-axis position of the projection in the room.

See the documentation for `sge.gfx.Sprite.draw_line()` for more information.

Room.**project_rectangle**(*x*, *y*, *z*, *width*, *height*, *fill=None*, *outline=None*, *outline_thickness=1*)
Project a rectangle onto the room.

Arguments:

- `x` – The horizontal location relative to the room to project the rectangle.
- `y` – The vertical location relative to the room to project the rectangle.
- `z` – The Z-axis position of the projection in the room.

See the documentation for `sge.gfx.Sprite.draw_rectangle()` for more information.

`Room.`**`project_ellipse`**(*x*, *y*, *z*, *width*, *height*, *fill=None*, *outline=None*, *outline_thickness=1*, *anti_alias=False*)
Project an ellipse onto the room.

Arguments:

- `x` – The horizontal location relative to the room to position the imaginary rectangle containing the ellipse.
- `y` – The vertical location relative to the room to position the imaginary rectangle containing the ellipse.
- `z` – The Z-axis position of the projection in the room.
- `width` – The width of the ellipse.
- `height` – The height of the ellipse.
- `outline_thickness` – The thickness of the outline of the ellipse.
- `anti_alias` – Whether or not anti-aliasing should be used.

See the documentation for `sge.gfx.Sprite.draw_ellipse()` for more information.

`Room.`**`project_circle`**(*x*, *y*, *z*, *radius*, *fill=None*, *outline=None*, *outline_thickness=1*, *anti_alias=False*)
Project a circle onto the room.

Arguments:

- `x` – The horizontal location relative to the room to position the center of the circle.
- `y` – The vertical location relative to the room to position the center of the circle.
- `z` – The Z-axis position of the projection in the room.

See the documentation for `sge.gfx.Sprite.draw_circle()` for more information.

`Room.`**`project_polygon`**(*points*, *z*, *fill=None*, *outline=None*, *outline_thickness=1*, *anti_alias=False*)
Draw a polygon on the sprite.

Arguments:

- `points` – A list of points relative to the room to position each of the polygon's angles. Each point should be a tuple in the form `(x, y)`, where x is the horizontal location and y is the vertical location.
- `z` – The Z-axis position of the projection in the room.

See the documentation for `sge.gfx.Sprite.draw_polygon()` for more information.

`Room.`**`project_sprite`**(*sprite*, *image*, *x*, *y*, *z*, *blend_mode=None*)
Project a sprite onto the room.

Arguments:

- `x` – The horizontal location relative to the room to project `sprite`.
- `y` – The vertical location relative to the room to project `sprite`.
- `z` – The Z-axis position of the projection in the room.

See the documentation for `sge.gfx.Sprite.draw_sprite()` for more information.

`Room.`**`project_text`**(*font*, *text*, *x*, *y*, *z*, *width=None*, *height=None*, *color=sge.gfx.Color("white")*, *halign='left'*, *valign='top'*, *anti_alias=True*)
Project text onto the room.

Arguments:

- `x` – The horizontal location relative to the room to project the text.

- y – The vertical location relative to the room to project the text.
- z – The Z-axis position of the projection in the room.

See the documentation for sge.gfx.Sprite.draw_text() for more information.

sge.dsp.Room Event Methods

Room.**event_room_start**()
Called when the room is started for the first time. It is always called after any sge.dsp.Game.event_game_start() and before any sge.dsp.Object.event_create occurring at the same time.

Room.**event_room_resume**()
Called when the room is started after it has already previously been started. It is always called before any sge.dsp.Object.event_create() occurring at the same time.

Room.**event_room_end**()
Called when another room is started or the game ends while this room is the current room. It is always called before any sge.dsp.Game.event_game_end() occurring at the same time.

Room.**event_step**(*time_passed*, *delta_mult*)
See the documentation for sge.dsp.Game.event_step() for more information.

Room.**event_alarm**(*alarm_id*)
See the documentation for sge.dsp.Room.alarms for more information.

Room.**event_key_press**(*key*, *char*)
See the documentation for sge.input.KeyPress for more information.

Room.**event_key_release**(*key*)
See the documentation for sge.input.KeyRelease for more information.

Room.**event_mouse_move**(*x*, *y*)
See the documentation for sge.input.MouseMove for more information.

Room.**event_mouse_button_press**(*button*)
Mouse button press event.

See the documentation for sge.input.MouseButtonPress for more information.

Room.**event_mouse_button_release**(*button*)
See the documentation for sge.input.MouseButtonRelease for more information.

Room.**event_joystick_axis_move**(*js_name*, *js_id*, *axis*, *value*)
See the documentation for sge.input.JoystickAxisMove for more information.

Room.**event_joystick_hat_move**(*js_name*, *js_id*, *hat*, *x*, *y*)
See the documentation for sge.input.JoystickHatMove for more information.

Room.**event_joystick_trackball_move**(*js_name*, *js_id*, *ball*, *x*, *y*)
See the documentation for sge.input.JoystickTrackballMove for more information.

Room.**event_joystick_button_press**(*js_name*, *js_id*, *button*)
See the documentation for sge.input.JoystickButtonPress for more information.

Room.**event_joystick_button_release**(*js_name*, *js_id*, *button*)
See the documentation for sge.input.JoystickButtonRelease for more information.

Room.**event_gain_keyboard_focus**()
See the documentation for sge.input.KeyboardFocusGain for more information.

Room.**event_lose_keyboard_focus**()
See the documentation for sge.input.KeyboardFocusLose for more information.

Room.**event_gain_mouse_focus**()
See the documentation for sge.input.MouseFocusGain for more information.

Room.**event_lose_mouse_focus**()
See the documentation for sge.input.MouseFocusLose for more information.

Room.**event_close**()
This is always called before any sge.dsp.Game.event_close() occurring at the same time.

See the documentation for sge.input.QuitRequest for more information.

Room.**event_paused_step**(*time_passed*, *delta_mult*)
See the documentation for sge.dsp.Game.event_step() for more information.

Room.**event_paused_key_press**(*key*, *char*)
See the documentation for sge.input.KeyPress for more information.

Room.**event_paused_key_release**(*key*)
See the documentation for sge.input.KeyRelease for more information.

Room.**event_paused_mouse_move**(*x*, *y*)
See the documentation for sge.input.MouseMove for more information.

Room.**event_paused_mouse_button_press**(*button*)
See the documentation for sge.input.MouseButtonPress for more information.

Room.**event_paused_mouse_button_release**(*button*)
See the documentation for sge.input.MouseButtonRelease for more information.

Room.**event_paused_joystick_axis_move**(*js_name*, *js_id*, *axis*, *value*)
See the documentation for sge.input.JoystickAxisMove for more information.

Room.**event_paused_joystick_hat_move**(*js_name*, *js_id*, *hat*, *x*, *y*)
See the documentation for sge.input.JoystickHatMove for more information.

Room.**event_paused_joystick_trackball_move**(*js_name*, *js_id*, *ball*, *x*, *y*)
See the documentation for sge.input.JoystickTrackballMove for more information.

Room.**event_paused_joystick_button_press**(*js_name*, *js_id*, *button*)
See the documentation for sge.input.JoystickButtonPress for more information.

Room.**event_paused_joystick_button_release**(*js_name*, *js_id*, *button*)
See the documentation for sge.input.JoystickButtonRelease for more information.

Room.**event_paused_gain_keyboard_focus**()
See the documentation for sge.input.KeyboardFocusGain for more information.

Room.**event_paused_lose_keyboard_focus**()
See the documentation for sge.input.KeyboardFocusLose for more information.

Room.**event_paused_gain_mouse_focus**()
See the documentation for sge.input.MouseFocusGain for more information.

Room.**event_paused_lose_mouse_focus**()
See the documentation for sge.input.MouseFocusLose for more information.

Room.**event_paused_close**()
See the documentation for sge.dsp.Room.event_close() for more information.

6.1.3 sge.dsp.View

class sge.dsp.**View**(*x*, *y*, *xport=0*, *yport=0*, *width=None*, *height=None*, *wport=None*, *hport=None*)

This class controls what the player sees in a room at any given time. Multiple views can exist in a room, and this can be used to create a split-screen effect.

x

The horizontal position of the view in the room. When set, if it brings the view outside of the room it is in, it will be re-adjusted so that the view is completely inside the room.

y

The vertical position of the view in the room. When set, if it brings the view outside of the room it is in, it will be re-adjusted so that the view is completely inside the room.

xport

The horizontal position of the view port on the window.

yport

The vertical position of the view port on the window.

width

The width of the view. When set, if it results in the view being outside of the room it is in, x will be adjusted so that the view is completely inside the room.

height

The height of the view. When set, if it results in the view being outside the room it is in, y will be adjusted so that the view is completely inside the room.

wport

The width of the view port. Set to None to make it the same as width. If this value differs from width, the image will be horizontally scaled so that it fills the port.

hport

The height of the view port. Set to None to make it the same as height. If this value differs from height, the image will be vertically scaled so that it fills the port.

rd

Reserved dictionary for internal use by the SGE. (Read-only)

sge.dsp.View Methods

View.**__init__**(*x*, *y*, *xport=0*, *yport=0*, *width=None*, *height=None*, *wport=None*, *hport=None*)

Arguments:

- width – The width of the view. If set to None, it will become sge.game.width - xport.
- height – The height of the view. If set to None, it will become sge.game.height - yport.

All other arugments set the respective initial attributes of the view. See the documentation for sge.dsp.View for more information.

6.1.4 sge.dsp.Object

class sge.dsp.**Object**(*x*, *y*, *z=0*, *sprite=None*, *visible=True*, *active=True*, *checks_collisions=True*, *tangible=True*, *bbox_x=None*, *bbox_y=None*, *bbox_width=None*, *bbox_height=None*, *regulate_origin=False*, *collision_ellipse=False*, *collision_precise=False*, *xvelocity=0*, *yvelocity=0*, *xacceleration=0*, *yacceleration=0*, *xdeceleration=0*, *ydeceleration=0*, *image_index=0*, *image_origin_x=None*, *image_origin_y=None*, *image_fps=None*, *image_xscale=1*, *image_yscale=1*, *image_rotation=0*, *image_alpha=255*, *image_blend=None*)

This class is used for game objects, such as the player, enemies, bullets, and the HUD. Generally, each type of object has its own subclass of sge.dsp.Object.

x

The horizontal position of the object in the room.

y

The vertical position of the object in the room.

z

The Z-axis position of the object in the room.

sprite

The sprite currently in use by this object. Can be either a sge.gfx.Sprite object or a sge.gfx.TileGrid object. Set to None for no sprite.

visible

Whether or not the object's sprite should be projected onto the screen.

active

Indicates whether the object is active (True) or inactive (False). While the object is active, it will exhibit normal behavior; events will be executed normally as will any other automatic functionality, such as adding xvelocity and yvelocity to x and y. If active is False, automatic functionality and normal events will be disabled.

Note: Inactive sge.dsp.Object objects are still visible by default and continue to be involved in collisions. In addition, collision events and destroy events still occur even if the object is inactive. If you wish for the object to not be visible, set visible to False. If you wish for the object to not perform collision events, set tangible to False.

---

checks_collisions

Whether or not the object should check for collisions automatically and cause collision events. If an object is not using collision events, setting this to False will give a boost in performance.

Note: This will not prevent automatic collision detection by other objects from detecting this object, and it will also not prevent this object's collision events from being executed. If you wish to disable collision detection entirely, set tangible to False.

---

tangible

Whether or not collisions involving the object can be detected. Setting this to False can improve performance if the object doesn't need to be involved in collisions.

Depending on the game, a useful strategy to boost performance can be to exclude an object from collision detection while it is outside the view. If you do this, you likely want to set active to False as well so that the object doesn't move in undesireable ways (e.g. through walls).

Note: If this is False, checks_collisions is implied to be False as well regardless of its actual value. This is because checking for collisions which can't be detected is meaningless.

bbox_x
The horizontal location of the bounding box relative to the object's position. If set to `None`, the value recommended by the sprite is used.

bbox_y
The vertical location of the bounding box relative to the object's position. If set to `None`, the value recommended by the sprite is used.

bbox_width
The width of the bounding box in pixels. If set to `None`, the value recommended by the sprite is used.

bbox_height
The height of the bounding box in pixels. If set to `None`, the value recommended by the sprite is used.

regulate_origin
If set to `True`, the origin is automatically adjusted to be the location of the pixel recommended by the sprite after transformation. This will cause rotation to be about the origin rather than being about the center of the image.

Note: The value of this attribute has no effect on the bounding box. If you wish for the bounding box to be adjusted as well, you must do so manually. As an alternative, you may want to consider using precise collision detection instead.

---

collision_ellipse
Whether or not an ellipse (rather than a rectangle) should be used for collision detection.

collision_precise
Whether or not precise (pixel-perfect) collision detection should be used. Note that this can be inefficient and does not work well with animated sprites.

bbox_left
The position of the left side of the bounding box in the room (same as `x` + `bbox_x`).

bbox_right
The position of the right side of the bounding box in the room (same as `bbox_left` + `bbox_width`).

bbox_top
The position of the top side of the bounding box in the room (same as `y` + `bbox_y`).

bbox_bottom
The position of the bottom side of the bounding box in the room (same as `bbox_top` + `bbox_height`).

xvelocity
The velocity of the object toward the right in pixels per frame.

yvelocity
The velocity of the object toward the bottom in pixels per frame.

speed
The total (directional) speed of the object in pixels per frame.

move_direction
The direction of the object's movement in degrees, with `0` being directly to the right and rotation in a positive direction being clockwise.

xacceleration
The acceleration of the object to the right in pixels per frame. If non-zero, movement as a result of `xvelocity` will be adjusted based on the kinematic equation, `v[f]^2 = v[i]^2 + 2*a*d`.

yacceleration
The acceleration of the object downward in pixels per frame. If non-zero, movement as a result of `yvelocity` will be adjusted based on the kinematic equation, `v[f]^2 = v[i]^2 + 2*a*d`.

xdeceleration
Like `xacceleration`, but its sign is ignored and it always causes the absolute value of `xvelocity` to decrease.

ydeceleration
Like `yacceleration`, but its sign is ignored and it always causes the absolute value of `yvelocity` to decrease.

image_index
The animation frame currently being displayed, with `0` being the first one.

image_origin_x
The horizontal location of the origin relative to the left edge of the images. If set to `None`, the value recommended by the sprite is used.

image_origin_y
The vertical location of the origin relative to the top edge of the images. If set to `None`, the value recommended by the sprite is used.

image_fps
The animation rate in frames per second. Can be negative, in which case animation will be reversed. If set to `None`, the value recommended by the sprite is used.

image_speed
The animation rate as a factor of `sge.game.fps`. Can be negative, in which case animation will be reversed. If set to `None`, the value recommended by the sprite is used.

image_xscale
The horizontal scale factor for the sprite. If this is negative, the sprite will also be mirrored horizontally.

image_yscale
The vertical scale factor for the sprite. If this is negative, the sprite will also be flipped vertically.

image_rotation
The rotation of the sprite in degrees, with rotation in a positive direction being clockwise.

If `regulate_origin` is `True`, the image is rotated about the origin. Otherwise, the image is rotated about its center.

image_alpha
The alpha value applied to the entire image, where `255` is the original image, `128` is half the opacity of the original image, `0` is fully transparent, etc.

image_blend
A `sge.gfx.Color` object representing the color to blend with the sprite (using RGBA Multiply blending). Set to `None` for no color blending.

alarms
A dictionary containing the alarms of the object. Each value decreases by 1 each frame (adjusted for delta timing if it is enabled). When a value is at or below 0, `event_alarm()` is executed with `alarm_id` set to the respective key, and the item is deleted from this dictionary.

mask
The current mask used for non-rectangular collision detection. See the documentation for `sge.collision.masks_collide()` for more information. (Read-only)

xstart
The initial value of `x` when the object was created. (Read-only)

ystart
The initial value of y when the object was created. (Read-only)

xprevious
The value of x at the end of the previous frame. (Read-only)

yprevious
The value of y at the end of the previous frame. (Read-only)

mask_x
The horizontal location of the mask in the room. (Read-only)

mask_y
The vertical location of the mask in the room. (Read-only)

rd
Reserved dictionary for internal use by the SGE. (Read-only)

sge.dsp.Object Methods

Object.**__init__**(*x*, *y*, *z=0*, *sprite=None*, *visible=True*, *active=True*, *checks_collisions=True*, *tangible=True*, *bbox_x=None*, *bbox_y=None*, *bbox_width=None*, *bbox_height=None*, *regulate_origin=False*, *collision_ellipse=False*, *collision_precise=False*, *xvelocity=0*, *yvelocity=0*, *xacceleration=0*, *yacceleration=0*, *xdeceleration=0*, *ydeceleration=0*, *image_index=0*, *image_origin_x=None*, *image_origin_y=None*, *image_fps=None*, *image_xscale=1*, *image_yscale=1*, *image_rotation=0*, *image_alpha=255*, *image_blend=None*)
Arugments set the respective initial attributes of the object. See the documentation for sge.dsp.Object for more information.

Object.**move_x**(*move*)
Move the object horizontally. This method can be overridden to conveniently define a particular way movement should be handled. Currently, it is used in the default implementation of event_update_position().

Arguments:

- move – The amount to add to x.

The default behavior of this method is the following code:

```
self.x += move
```

Object.**move_y**(*move*)
Move the object vertically. This method can be overridden to conveniently define a particular way movement should be handled. Currently, it is used in the default implementation of event_update_position().

Arguments:

- move – The amount to add to y.

The default behavior of this method is the following code:

```
self.y += move
```

Object.**collision**(*other=None*, *x=None*, *y=None*)
Return a list of objects colliding with this object.

Arguments:

- other – What to check for collisions with. Can be one of the following:
  - A sge.dsp.Object object.

– A list of `sge.dsp.Object` objects.

– A class derived from `sge.dsp.Object`.

– `None`: Check for collisions with all objects.

- `x` – The horizontal position to pretend this object is at for the purpose of the collision detection. If set to `None`, `x` will be used.
- `y` – The vertical position to pretend this object is at for the purpose of the collision detection. If set to `None`, `y` will be used.

`Object.`**`destroy`**`()`
Remove the object from the current room. `foo.destroy()` is identical to `sge.game.current_room.remove(foo)`.

classmethod `Object.`**`create`**`(`**args*, ***kwargs*`)`
Create an object of this class and add it to the current room.

`args` and `kwargs` are passed to the constructor method of `cls` as arguments. Calling `obj = cls.create(*args, **kwargs)` is the same as:

```
obj = cls(*args, **kwargs)
sge.game.current_room.add(obj)
```

sge.dsp.Object Event Methods

`Object.`**`event_create`**`()`
Called in the following cases:

- Right after the object is added to the current room.
- Right after a room starts for the first time after the object was added to it, if and only if the object was added to the room while it was not the current room. In this case, this event is called for each appropriate object after the respective room start event or room resume event, in the same order that the objects were added to the room.

`Object.`**`event_destroy`**`()`
Called right after the object is removed from the current room.

Note: If the object is removed from a room while it is not the current room, this method will not be called.

`Object.`**`event_step`**`(`*time_passed*, *delta_mult*`)`
Called each frame after automatic updates to objects (such as the effects of the speed variables), but before collision events.

See the documentation for `sge.dsp.Game.event_step()` for more information.

`Object.`**`event_alarm`**`(`*alarm_id*`)`
See the documentation for `sge.dsp.Object.alarms` for more information.

`Object.`**`event_animation_end`**`()`
Called when an animation cycle ends.

`Object.`**`event_key_press`**`(`*key*, *char*`)`
See the documentation for `sge.input.KeyPress` for more information.

`Object.`**`event_key_release`**`(`*key*`)`
See the documentation for `sge.input.KeyRelease` for more information.

Object.**event_mouse_move**(*x*, *y*)
See the documentation for sge.input.MouseMove for more information.

Object.**event_mouse_button_press**(*button*)
See the documentation for sge.input.MouseButtonPress for more information.

Object.**event_mouse_button_release**(*button*)
See the documentation for sge.input.MouseButtonRelease for more information.

Object.**event_joystick_axis_move**(*js_name*, *js_id*, *axis*, *value*)
See the documentation for sge.input.JoystickAxisMove for more information.

Object.**event_joystick_hat_move**(*js_name*, *js_id*, *hat*, *x*, *y*)
See the documentation for sge.input.JoystickHatMove for more information.

Object.**event_joystick_trackball_move**(*js_name*, *js_id*, *ball*, *x*, *y*)
See the documentation for sge.input.JoystickTrackballMove for more information.

Object.**event_joystick_button_press**(*js_name*, *js_id*, *button*)
See the documentation for sge.input.JoystickButtonPress for more information.

Object.**event_joystick_button_release**(*js_name*, *js_id*, *button*)
See the documentation for sge.input.JoystickButtonRelease for more information.

Object.**event_update_position**(*delta_mult*)
Called when it's time to update the position of the object. This method handles this functionality, so defining this will override the default behavior and allow you to handle the speed variables in a special way.

The default behavior of this method is the following code:

```
if delta_mult:
    vi = self.xvelocity
    vf = vi + self.xacceleration * delta_mult
    dc = abs(self.xdeceleration) * delta_mult
    if abs(vf) > dc:
        vf -= math.copysign(dc, vf)
    else:
        vf = 0
    self.xvelocity = vf
    self.move_x(((vi + vf) / 2) * delta_mult)

    vi = self.yvelocity
    vf = vi + self.yacceleration * delta_mult
    dc = abs(self.ydeceleration) * delta_mult
    if abs(vf) > dc:
        vf -= math.copysign(dc, vf)
    else:
        vf = 0
    self.yvelocity = vf
    self.move_y(((vi + vf) / 2) * delta_mult)
```

See the documentation for sge.dsp.Game.event_step() for more information.

Object.**event_collision**(*other*, *xdirection*, *ydirection*)
Called when this object collides with another object.

Arguments:

- other – The other object which was collided with.
- xdirection – The horizontal direction of the collision from the perspective of this object. Can be -1 (left), 1 (right), or 0 (no horizontal direction).

- `ydirection` – The vertical direction of the collision from the perspective of this object. Can be `-1` (up), `1` (down), or `0` (no vertical direction).

Directionless "collisions" (ones with both an xdirection and ydirection of `0`) are possible. These are typically collisions which were already occurring in the previous frame (continuous collisions).

`Object`.**event_paused_step**(*time_passed*, *delta_mult*)
See the documentation for `sge.dsp.Game.event_step()` for more information.

`Object`.**event_paused_key_press**(*key*, *char*)
See the documentation for `sge.input.KeyPress` for more information.

`Object`.**event_paused_key_release**(*key*)
See the documentation for `sge.input.KeyRelease` for more information.

`Object`.**event_paused_mouse_move**(*x*, *y*)
See the documentation for `sge.input.MouseMove` for more information.

`Object`.**event_paused_mouse_button_press**(*button*)
See the documentation for `sge.input.MouseButtonPress` for more information.

`Object`.**event_paused_mouse_button_release**(*button*)
See the documentation for `sge.input.MouseButtonRelease` for more information.

`Object`.**event_paused_joystick_axis_move**(*js_name*, *js_id*, *axis*, *value*)
See the documentation for `sge.input.JoystickAxisMove` for more information.

`Object`.**event_paused_joystick_hat_move**(*js_name*, *js_id*, *hat*, *x*, *y*)
See the documentation for `sge.input.JoystickHatMove` for more information.

`Object`.**event_paused_joystick_trackball_move**(*js_name*, *js_id*, *ball*, *x*, *y*)
See the documentation for `sge.input.JoystickTrackballMove` for more information.

`Object`.**event_paused_joystick_button_press**(*js_name*, *js_id*, *button*)
See the documentation for `sge.input.JoystickButtonPress` for more information.

`Object`.**event_paused_joystick_button_release**(*js_name*, *js_id*, *button*)
See the documentation for `sge.input.JoystickButtonRelease` for more information.

CHAPTER

SEVEN

SGE.GFX

Contents

- sge.gfx
  - sge.gfx Classes
    - sge.gfx.Color
    - sge.gfx.Sprite
      - sge.gfx.Sprite Methods
    - sge.gfx.Font
      - sge.gfx.Font Methods
    - sge.gfx.BackgroundLayer
      - sge.gfx.BackgroundLayer Methods
    - sge.gfx.Background
      - sge.gfx.Background Methods

This module provides classes related to rendering graphics.

7.1 sge.gfx Classes

7.1.1 sge.gfx.Color

class `sge.gfx.`**`Color`**(*value*)

This class stores color information.

Objects of this class can be converted to iterables indicating the object's `red`, `green`, `blue`, and `alpha` values, respectively; to integers which can be interpreted as a hexadecimal representation of the color, excluding alpha transparency; and to strings which indicate the English name of the color (in all lowercase) if possible, and `hex_string` otherwise.

`red`

The red component of the color as an integer, where `0` indicates no red intensity and `255` indicates full red intensity.

`green`

The green component of the color as an integer, where `0` indicates no green intensity and `255` indicates full green intensity.

`blue`

The blue component of the color as an integer, where `0` indicates no blue intensity and `255` indicates full blue intensity.

alpha
The alpha transparency of the color as an integer, where `0` indicates full transparency and `255` indicates full opacity.

hex_string
An HTML hex string representation of the color. (Read-only)

7.1.2 sge.gfx.Sprite

class `sge.gfx.`**Sprite**(*name=None*, *directory=''*, *width=None*, *height=None*, *transparent=True*, *origin_x=0*, *origin_y=0*, *fps=60*, *bbox_x=None*, *bbox_y=None*, *bbox_width=None*, *bbox_height=None*)

This class stores images and information about how the SGE is to use those images.

What image formats are supported depends on the implementation of the SGE, but image formats that are generally a good choice are PNG and JPEG. See the implementation-specific information for a full list of supported formats.

width
The width of the sprite.

Note: Changing this attribute will cause the sprite to be scaled horizontally. This is a destructive transformation: it can result in loss of pixel information, especially if it is done repeatedly. Because of this, it is advised that you do not adjust this value for routine scaling. Use the `image_xscale` attribute of a `sge.dsp.Object` object instead.

---

height
The height of the sprite.

Note: Changing this attribute will cause the sprite to be scaled vertically. This is a destructive transformation: it can result in loss of pixel information, especially if it is done repeatedly. Because of this, it is advised that you do not adjust this value for routine scaling. Use the `image_yscale` attribute of a `sge.dsp.Object` object instead.

---

transparent
Whether or not the image should be partially transparent, based on the image's alpha channel. If an image does not have an alpha channel, a colorkey will be used, with the transparent color being the color of the top-rightmost pixel of the respective frame.

If this is a `sge.gfx.Color` object, the indicated color will be used as a colorkey.

origin_x
The suggested horizontal location of the origin relative to the left edge of the images.

origin_y
The suggested vertical location of the origin relative to the top edge of the images.

fps
The suggested rate in frames per second to animate the image at. Can be negative, in which case animation will be reversed.

speed
The suggested rate to animate the image at as a factor of `sge.game.fps`. Can be negative, in which case animation will be reversed.

bbox_x
The horizontal location relative to the sprite of the suggested bounding box to use with it. If set to `None`, it will become equal to `-origin_x` (which is always the left edge of the image).

bbox_y
The vertical location relative to the sprite of the suggested bounding box to use with it. If set to `None`, it will become equal to `-origin_y` (which is always the top edge of the image).

bbox_width
The width of the suggested bounding box. If set to `None`, it will become equal to `width - bbox_x` (which is always everything on the image to the right of `bbox_x`).

bbox_height
The height of the suggested bounding box. If set to `None`, it will become equal to `height - bbox_y` (which is always everything on the image below `bbox_y`).

name
The name of the sprite given when it was created. (Read-only)

frames
The number of animation frames in the sprite. (Read-only)

rd
Reserved dictionary for internal use by the SGE. (Read-only)

sge.gfx.Sprite Methods

`Sprite.__init__`(*name=None*, *directory=''*, *width=None*, *height=None*, *transparent=True*, *origin_x=0*, *origin_y=0*, *fps=60*, *bbox_x=None*, *bbox_y=None*, *bbox_width=None*, *bbox_height=None*)
Arguments:

- `name` – The base name of the image files, used to find all individual image files that make up the sprite's animation'. One of the following rules will be used to find the images:
  - The base name plus a valid image extension. If this rule is used, the image will be loaded as a single-frame sprite.
  - The base name and an integer separated by either a hyphen (-) or an underscore (_) and followed by a valid image extension. If this rule is used, all images with names like this are loaded and treated as an animation, with the lower-numbered images being earlier frames.
  - The base name and an integer separated by either `-strip` or `_strip` and followed by a valid image extension. If this rule is used, the image will be treated as an animation read as a horizontal reel from left to right, split into the number of frames indicated by the integer.
  - If the base name is `None`, the sprite will be a fully transparent rectangle at the specified size (with both `width` and `height` defaulting to 32 if they are set to `None`). The SGE decides what to assign to the sprite's `name` attribute in this case, but it will always be a string.

  If none of the above rules can be used, `IOError` is raised.
- `directory` – The directory to search for image files in.

All other arguments set the respective initial attributes of the sprite. See the documentation for `Sprite` for more information.

`Sprite.append_frame`()
Append a new blank frame to the end of the sprite.

`Sprite.insert_frame`(*frame*)
Insert a new blank frame into the sprite.

Arguments:

- `frame` – The frame of the sprite to insert the new frame in front of, where `0` is the first frame.

Sprite.**extend**(*sprite*)
Extend this sprite with the frames of another sprite.

If the size of the frames added is different from the size of this sprite, they are scaled to this sprite's size.

Arguments:

- `sprite` – The sprite to add the frames of to this sprite.

Sprite.**delete_frame**(*frame*)
Delete a frame from the sprite.

Arguments:

- `frame` – The frame of the sprite to delete, where `0` is the first frame.

Sprite.**get_pixel**(*x*, *y*, *frame=0*)
Return a `sge.gfx.Color` object indicating the color of a particular pixel on the sprite.

Arguments:

- `x` – The horizontal location relative to the sprite of the pixel to check.
- `y` – The vertical location relative to the sprite of the pixel to check.
- `frame` – The frame of the sprite to check, where `0` is the first frame.

Sprite.**get_pixels**(*frame=0*)
Return a two-dimensional list of :class'sge.gfx.Color' objects indicating the colors of a particular frame's pixels.

A returned list given the name `pixels` is indexed as `pixels[x][y]`, where `x` is the horizontal location of the pixel and `y` is the vertical location of the pixel.

Arguments:

- `frame` – The frame of the sprite to check, where `0` is the first frame.

Sprite.**draw_dot**(*x*, *y*, *color*, *frame=None*)
Draw a single-pixel dot on the sprite.

Arguments:

- `x` – The horizontal location relative to the sprite to draw the dot.
- `y` – The vertical location relative to the sprite to draw the dot.
- `color` – A `sge.gfx.Color` object representing the color of the dot.
- `frame` – The frame of the sprite to draw on, where `0` is the first frame; set to `None` to draw on all frames.

Sprite.**draw_line**(*x1*, *y1*, *x2*, *y2*, *color*, *thickness=1*, *anti_alias=False*, *frame=None*)
Draw a line segment on the sprite.

Arguments:

- `x1` – The horizontal location relative to the sprite of the first end point of the line segment.
- `y1` – The vertical location relative to the sprite of the first end point of the line segment.
- `x2` – The horizontal location relative to the sprite of the second end point of the line segment.
- `y2` – The vertical location relative to the sprite of the second end point of the line segment.
- `color` – A `sge.gfx.Color` object representing the color of the line segment.
- `thickness` – The thickness of the line segment.
- `anti_alias` – Whether or not anti-aliasing should be used.

- `frame` – The frame of the sprite to draw on, where `0` is the first frame; set to `None` to draw on all frames.

`Sprite.`**`draw_rectangle`**(*x*, *y*, *width*, *height*, *fill=None*, *outline=None*, *outline_thickness=1*, *frame=None*)

Draw a rectangle on the sprite.

Arguments:

- `x` – The horizontal location relative to the sprite to draw the rectangle.
- `y` – The vertical location relative to the sprite to draw the rectangle.
- `width` – The width of the rectangle.
- `height` – The height of the rectangle.
- `fill` – A `sge.gfx.Color` object representing the color of the fill of the rectangle.
- `outline` – A `sge.gfx.Color` object representing the color of the outline of the rectangle.
- `outline_thickness` – The thickness of the outline of the rectangle.
- `frame` – The frame of the sprite to draw on, where `0` is the first frame; set to `None` to draw on all frames.

`Sprite.`**`draw_ellipse`**(*x*, *y*, *width*, *height*, *fill=None*, *outline=None*, *outline_thickness=1*, *anti_alias=False*, *frame=None*)

Draw an ellipse on the sprite.

Arguments:

- `x` – The horizontal location relative to the sprite to position the imaginary rectangle containing the ellipse.
- `y` – The vertical location relative to the sprite to position the imaginary rectangle containing the ellipse.
- `width` – The width of the ellipse.
- `height` – The height of the ellipse.
- `fill` – A `sge.gfx.Color` object representing the color of the fill of the ellipse.
- `outline` – A `sge.gfx.Color` object representing the color of the outline of the ellipse.
- `outline_thickness` – The thickness of the outline of the ellipse.
- `anti_alias` – Whether or not anti-aliasing should be used.
- `frame` – The frame of the sprite to draw on, where `0` is the first frame; set to `None` to draw on all frames.

`Sprite.`**`draw_circle`**(*x*, *y*, *radius*, *fill=None*, *outline=None*, *outline_thickness=1*, *anti_alias=False*, *frame=None*)

Draw a circle on the sprite.

Arguments:

- `x` – The horizontal location relative to the sprite to position the center of the circle.
- `y` – The vertical location relative to the sprite to position the center of the circle.
- `radius` – The radius of the circle.
- `fill` – A `sge.gfx.Color` object representing the color of the fill of the circle.
- `outline` – A `sge.gfx.Color` object representing the color of the outline of the circle.
- `outline_thickness` – The thickness of the outline of the circle.
- `anti_alias` – Whether or not anti-aliasing should be used.
- `frame` – The frame of the sprite to draw on, where `0` is the first frame; set to `None` to draw on all frames.

Sprite.**draw_polygon**(*points*, *fill=None*, *outline=None*, *outline_thickness=1*, *anti_alias=False*, *frame=None*)

Draw a polygon on the sprite.

Arguments:

- `points` – A list of points relative to the sprite to position each of the polygon's angles. Each point should be a tuple in the form `(x, y)`, where x is the horizontal location and y is the vertical location.
- `fill` – A `sge.gfx.Color` object representing the color of the fill of the polygon.
- `outline` – A `sge.gfx.Color` object representing the color of the outline of the polygon.
- `outline_thickness` – The thickness of the outline of the polygon.
- `anti_alias` – Whether or not anti-aliasing should be used.
- `frame` – The frame of the sprite to draw on, where `0` is the first frame; set to `None` to draw on all frames.

Sprite.**draw_sprite**(*sprite*, *image*, *x*, *y*, *frame=None*, *blend_mode=None*)

Draw another sprite on the sprite.

Arguments:

- `sprite` – The `sge.gfx.Sprite` or `sge.gfx.TileGrid` object to draw with.
- `image` – The frame of `sprite` to draw with, where `0` is the first frame.
- `x` – The horizontal location relative to `self` to position the origin of `sprite`.
- `y` – The vertical location relative to `self` to position the origin of `sprite`.
- `frame` – The frame of the sprite to draw on, where `0` is the first frame; set to `None` to draw on all frames.
- `blend_mode` – The blend mode to use. Possible blend modes are:
  - `sge.BLEND_NORMAL`
  - `sge.BLEND_RGBA_ADD`
  - `sge.BLEND_RGBA_SUBTRACT`
  - `sge.BLEND_RGBA_MULTIPLY`
  - `sge.BLEND_RGBA_SCREEN`
  - `sge.BLEND_RGBA_MINIMUM`
  - `sge.BLEND_RGBA_MAXIMUM`
  - `sge.BLEND_RGB_ADD`
  - `sge.BLEND_RGB_SUBTRACT`
  - `sge.BLEND_RGB_MULTIPLY`
  - `sge.BLEND_RGB_SCREEN`
  - `sge.BLEND_RGB_MINIMUM`
  - `sge.BLEND_RGB_MAXIMUM`

  `None` is treated as `sge.BLEND_NORMAL`.

Sprite.**draw_text**(*font*, *text*, *x*, *y*, *width=None*, *height=None*, *color=sge.gfx.Color("white")*, *halign='left'*, *valign='top'*, *anti_alias=True*, *frame=None*)

Draw text on the sprite.

Arguments:

- font – The font to use to draw the text.
- text – The text (as a string) to draw.
- x – The horizontal location relative to the sprite to draw the text.
- y – The vertical location relative to the sprite to draw the text.
- width – The width of the imaginary rectangle the text is drawn in; set to None to make the rectangle as wide as needed to contain the text without additional line breaks. If set to something other than None, a line which does not fit will be automatically split into multiple lines that do fit.
- height – The height of the imaginary rectangle the text is drawn in; set to None to make the rectangle as tall as needed to contain the text.
- color – A sge.gfx.Color object representing the color of the text.
- halign – The horizontal alignment of the text and the horizontal location of the origin of the imaginary rectangle the text is drawn in. Can be set to one of the following:
  - "left" – Align the text to the left of the imaginary rectangle the text is drawn in. Set the origin of the imaginary rectangle to its left edge.
  - "center" – Align the text to the center of the imaginary rectangle the text is drawn in. Set the origin of the imaginary rectangle to its center.
  - "right" – Align the text to the right of the imaginary rectangle the text is drawn in. Set the origin of the imaginary rectangle to its right edge.
- valign – The vertical alignment of the text and the vertical location of the origin of the imaginary rectangle the text is drawn in. Can be set to one of the following:
  - "top" – Align the text to the top of the imaginary rectangle the text is drawn in. Set the origin of the imaginary rectangle to its top edge. If the imaginary rectangle is not tall enough to contain all of the text, cut text off from the bottom.
  - "middle" – Align the the text to the middle of the imaginary rectangle the text is drawn in. Set the origin of the imaginary rectangle to its middle. If the imaginary rectangle is not tall enough to contain all of the text, cut text off equally from the top and bottom.
  - "bottom" – Align the text to the bottom of the imaginary rectangle the text is drawn in. Set the origin of the imaginary rectangle to its top edge. If the imaginary rectangle is not tall enough to contain all of the text, cut text off from the top.
- anti_alias – Whether or not anti-aliasing should be used.
- frame – The frame of the sprite to draw on, where 0 is the first frame; set to None to draw on all frames.

Sprite.**draw_erase**(*x*, *y*, *width*, *height*, *frame=None*)

Erase part of the sprite.

Arguments:

- x – The horizontal location relative to the sprite of the area to erase.
- y – The vertical location relative to the sprite of the area to erase.
- width – The width of the area to erase.
- height – The height of the area to erase.
- frame – The frame of the sprite to erase from, where 0 is the first frame; set to None to erase from all frames.

Sprite.**draw_clear**(*frame=None*)
Erase everything from the sprite.

Arguments:

- `frame` – The frame of the sprite to clear, where `0` is the first frame; set to `None` to clear all frames.

Sprite.**draw_lock**()
Lock the sprite for continuous drawing.

Use this method to "lock" the sprite for being drawn on several times in a row. What exactly this does depends on the implementation and it may even do nothing at all, but calling this method before doing several draw actions on the sprite in a row gives the SGE a chance to make the drawing more efficient.

After you are done with continuous drawing, call `draw_unlock()` to let the SGE know that you are done drawing.

Warning: Do not cause a sprite to be used while it's locked. For example, don't leave it locked for the duration of a frame, and don't draw it or project it on anything. The effect of using a locked sprite could be as minor as graphical errors and as severe as crashing the program, depending on the SGE implementation. Always call `draw_unlock()` immediately after you're done drawing for a while.

Sprite.**draw_unlock**()
Unlock the sprite.

Use this method to "unlock" the sprite after it has been "locked" for continuous drawing by `draw_lock()`.

Sprite.**mirror**(*frame=None*)
Mirror the sprite horizontally.

Arguments:

- `frame` – The frame of the sprite to mirror, where `0` is the first frame; set to `None` to mirror all frames.

Sprite.**flip**(*frame=None*)
Flip the sprite vertically.

Arguments:

- `frame` – The frame of the sprite to flip, where `0` is the first frame; set to `None` to flip all frames.

Sprite.**resize_canvas**(*width*, *height*)
Resize the sprite by adding empty space instead of scaling.

After resizing the canvas:

1. The horizontal location of the origin is multiplied by the new width divided by the old width.
2. The vertical location of the origin is multiplied by the new height divided by the old height.
3. All frames are repositioned within the sprite such that their position relative to the origin is the same as before.

Arguments:

- `width` – The width to set the sprite to.
- `height` – The height to set the sprite to.

Sprite.**scale**(*xscale=1*, *yscale=1*, *frame=None*)
Scale the sprite to a different size.

Unlike changing `width` and `height`, this function does not result in the actual size of the sprite changing. Instead, any scaled frames are repositioned so that the pixel which was at the origin before scaling remains at the origin.

Arguments:

- `xscale` – The horizontal scale factor.
- `yscale` – The vertical scale factor.
- `frame` – The frame of the sprite to rotate, where `0` is the first frame; set to `None` to rotate all frames.

Note: This is a destructive transformation: it can result in loss of pixel information, especially if it is done repeatedly. Because of this, it is advised that you do not adjust this value for routine scaling. Use the `image_xscale` and `image_yscale` attributes of a `sge.dsp.Object` object instead.

Note: Because this function does not alter the actual size of the sprite, scaling up may result in some parts of the image being cropped off.

`Sprite.`**`rotate`**(*x*, *adaptive_resize=True*, *frame=None*)

Rotate the sprite about the center.

Arguments:

- `x` – The rotation amount in degrees, with rotation in a positive direction being clockwise.
- `adaptive_resize` – Whether or not the sprite should be resized to accomodate rotation. If this is `True`, rotation amounts other than multiples of 180 will result in the size of the sprite being adapted to fit the whole rotated image. The origin and any frames which have not been rotated will also be moved so that their location relative to the rotated image(s) is the same.
- `frame` – The frame of the sprite to rotate, where `0` is the first frame; set to `None` to rotate all frames.

Note: This is a destructive transformation: it can result in loss of pixel information, especially if it is done repeatedly. Because of this, it is advised that you do not adjust this value for routine rotation. Use the `image_rotation` attribute of a `sge.dsp.Object` object instead.

`Sprite.`**`copy`**()

Return a copy of the sprite.

`Sprite.`**`save`**(*fname*)

Save the sprite to an image file.

Arguments:

- `fname` – The path of the file to save the sprite to. If it is not a path that can be saved to, `IOError` is raised.

If the sprite has multiple frames, the image file saved will be a horizontal reel of each of the frames from left to right with no space in between the frames.

classmethod `Sprite.`**`from_tween`**(*sprite*, *frames*, *fps=None*, *xscale=None*, *yscale=None*, *rotation=None*, *blend=None*, *bbox_x=None*, *bbox_y=None*, *bbox_width=None*, *bbox_height=None*)

Create a sprite based on tweening an existing sprite.

"Tweening" refers to creating an animation from gradual transformations to an image. For example, this can be used to generate an animation of an object growing to a particular size. The animation generated is called a "tween".

Arguments:

- `sprite` – The sprite to base the tween on. If the sprite includes multiple frames, all frames will be used in sequence until the end of the tween.

The tween's origin is derived from this sprite's origin, adjusted appropriately to accomodate any size changes made. Whether or not the tween is transparent is also determined by whether or not this sprite is transparent.

- `frames` – The number of frames the to make the tween take up.
- `fps` – The suggested rate of animation for the tween in frames per second. If set to `None`, the suggested animation rate of the base sprite is used.
- `xscale` – The horizontal scale factor at the end of the tween. If set to `None`, horizontal scaling will not be included in the tweening process.
- `yscale` – The vertical scale factor at the end of the tween. If set to `None`, vertical scaling will not be included in the tweening process.
- `rotation` – The total clockwise rotation amount in degrees at the end of the tween. Can be negative to indicate counter-clockwise rotation instead. If set to `None`, rotation will not be included in the tweening process.
- `blend` – A `sge.gfx.Color` object representing the color to blend with the sprite (using RGBA Multiply blending) at the end of the tween. If set to `None`, color blending will not be included in the tweening process.

All other arguments set the respective initial attributes of the tween. See the documentation for `sge.gfx.Sprite` for more information.

classmethod `Sprite.`**`from_text`**(*font*, *text*, *width=None*, *height=None*, *color=sge.gfx.Color("white")*, *halign='left'*, *valign='top'*, *anti_alias=True*)

Create a sprite, draw the given text on it, and return the sprite. See the documentation for `sge.gfx.Sprite.draw_text()` for more information.

The sprite's origin is set based on `halign` and `valign`.

classmethod `Sprite.`**`from_tileset`**(*fname*, *x=0*, *y=0*, *columns=1*, *rows=1*, *xsep=0*, *ysep=0*, *width=1*, *height=1*, *origin_x=0*, *origin_y=0*, *transparent=True*, *fps=0*, *bbox_x=None*, *bbox_y=None*, *bbox_width=None*, *bbox_height=None*)

Return a sprite based on the tiles in a tileset.

Arguments:

- `fname` – The path to the image file containing the tileset.
- `x` – The horizontal location relative to the image of the top-leftmost tile in the tileset.
- `y` – The vertical location relative to the image of the top-leftmost tile in the tileset.
- `columns` – The number of columns in the tileset.
- `rows` – The number of rows in the tileset.
- `xsep` – The spacing between columns in the tileset.
- `ysep` – The spacing between rows in the tileset.
- `width` – The width of the tiles.
- `height` – The height of the tiles.

For all other arguments, see the documentation for `Sprite.__init__()`.

Each tile in the tileset becomes a subimage of the returned sprite, ordered first from left to right and then from top to bottom.

classmethod `Sprite.from_screenshot`(*x=0*, *y=0*, *width=None*, *height=None*)
Return the current display on the screen as a sprite.

Arguments:

- `x` – The horizontal location of the rectangular area to take a screenshot of.
- `y` – The vertical location of the rectangular area to take a screenshot of.
- `width` – The width of the area to take a screenshot of; set to None for all of the area to the right of `x` to be included.
- `height` – The height of the area to take a screenshot of; set to `None` for all of the area below `y` to be included.

If you only wish to save a screenshot (of the entire screen) to a file, the easiest way to do that is:

```
sge.gfx.Sprite.from_screenshot().save("foo.png")
```

7.1.3 sge.gfx.Font

class `sge.gfx.Font`(*name=None*, *size=12*, *underline=False*, *bold=False*, *italic=False*)
This class stores a font for use by text drawing methods.

Note that bold and italic rendering could be ugly. It is better to choose a bold or italic font rather than enabling bold or italic rendering, if possible.

size
The height of the font in pixels.

underline
Whether or not underlined rendering is enabled.

bold
Whether or not bold rendering is enabled.

Note: Using this option can be ugly. It is better to choose a bold font rather than enabling bold rendering, if possible.

---

italic
Whether or not italic rendering is enabled.

Note: Using this option can be ugly. It is better to choose an italic font rather than enabling italic rendering, if possible.

---

name
The name of the font as specified when it was created. (Read-only)

rd
Reserved dictionary for internal use by the SGE. (Read-only)

sge.gfx.Font Methods

`Font.__init__`(*name=None*, *size=12*, *underline=False*, *bold=False*, *italic=False*)
Arguments:

- `name` – The name of the font. Can be one of the following:
  - A string indicating the path to the font file.

– A string indicating the case-insensitive name of a system font, e.g. `"Liberation Serif"`.

– A list or tuple of strings indicating either a font file or a system font to choose from in order of preference.

If none of the above methods return a valid font, the SGE will choose the font.

All other arguments set the respective initial attributes of the font. See the documentation for `sge.gfx.Font` for more information.

Note: It is generally not a good practice to rely on system fonts. There are no standard fonts; a font which you have on your system is probably not on everyone's system. Rather than relying on system fonts, choose a font which is under a libre license (such as the GNU General Public License or the SIL Open Font License) and distribute it with your game; this will ensure that everyone sees text rendered the same way you do.

`Font.`**`get_width`**(*text*, *width=None*, *height=None*)

Return the width of a certain string of text when rendered.

See the documentation for `sge.gfx.Sprite.draw_text()` for more information.

`Font.`**`get_height`**(*text*, *width=None*, *height=None*)

Return the height of a certain string of text when rendered.

See the documentation for `sge.gfx.Sprite.draw_text()` for more information.

classmethod `Font.`**`from_sprite`**(*sprite*, *chars*, *hsep=0*, *vsep=0*, *size=12*, *underline=False*, *bold=False*, *italic=False*)

Return a font derived from a sprite.

Arguments:

- `sprite` – The `sge.gfx.Sprite` object to derive the font from.
- `chars` – A list of characters to set the sprite's frames to. For example, `['A', 'B', 'C']` would assign the first frame to the letter "A", the second frame to the letter "B", and the third frame to the letter "C". Any character not listed here will be rendered as its differently-cased counterpart if possible (e.g. "A" as "a") or as a blank space otherwise.
- `hsep` – The amount of horizontal space to place between characters when text is rendered.
- `vsep` – The amount of vertical space to place between lines when text is rendered.

All other arguments set the respective initial attributes of the font. See the documentation for `sge.gfx.Font` for more information.

The font's `name` attribute will be set to the name of the sprite the font is derived from.

The font's `size` attribute will indicate the height of the characters in pixels. The width of the characters will be adjusted proportionally.

7.1.4 sge.gfx.BackgroundLayer

class `sge.gfx.`**`BackgroundLayer`**(*sprite*, *x*, *y*, *z=0*, *xscroll_rate=1*, *yscroll_rate=1*, *repeat_left=False*, *repeat_right=False*, *repeat_up=False*, *repeat_down=False*)

This class stores a sprite and certain information for a layer of a background. In particular, it stores the location of the layer, whether the layer tiles horizontally, vertically, or both, and the rate at which it scrolls.

`sprite`

The sprite used for this layer. It will be animated normally if it contains multiple frames.

`x`

The horizontal location of the layer relative to the background.

y
The vertical location of the layer relative to the background.

z
The Z-axis position of the layer in the room.

xscroll_rate
The horizontal rate that the layer scrolls as a factor of the additive inverse of the horizontal movement of the view.

yscroll_rate
The vertical rate that the layer scrolls as a factor of the additive inverse of the vertical movement of the view.

repeat_left
Whether or not the layer should be repeated (tiled) to the left.

repeat_right
Whether or not the layer should be repeated (tiled) to the right.

repeat_up
Whether or not the layer should be repeated (tiled) upwards.

repeat_down
Whether or not the layer should be repeated (tiled) downwards.

rd
Reserved dictionary for internal use by the SGE. (Read-only)

sge.gfx.BackgroundLayer Methods

`BackgroundLayer.__init__`(*sprite*, *x*, *y*, *z=0*, *xscroll_rate=1*, *yscroll_rate=1*, *repeat_left=False*, *repeat_right=False*, *repeat_up=False*, *repeat_down=False*)
Arguments set the respective initial attributes of the layer. See the documentation for `sge.gfx.BackgroundLayer` for more information.

7.1.5 sge.gfx.Background

class `sge.gfx.`**Background**(*layers*, *color*)
This class stores the layers that make up the background (which contain the information about what images to use and where) as well as the color to use where no image is shown.

layers
A list containing all `sge.gfx.BackgroundLayer` objects used in this background. (Read-only)

color
A `sge.gfx.Color` object representing the color used in parts of the background where no layer is shown.

rd
Reserved dictionary for internal use by the SGE. (Read-only)

sge.gfx.Background Methods

`Background.__init__`(*layers*, *color*)
Arguments set the respective initial attributes of the background. See the documentation for `sge.gfx.Background` for more information.

CHAPTER

EIGHT

SGE.SND

Contents

- sge.snd
  - sge.snd Classes
    - sge.snd.Sound
      - sge.snd.Sound Methods
    - sge.snd.Music
      - sge.snd.Music Methods
  - sge.snd Functions
    - sge.snd.stop_all

This module provides classes related to the sound system.

8.1 sge.snd Classes

8.1.1 sge.snd.Sound

class `sge.snd.`**Sound**(*fname*, *volume=1*, *max_play=1*)

This class stores and plays sound effects. Note that this is inefficient for large music files; for those, use `sge.snd.Music` instead.

What sound formats are supported depends on the implementation of the SGE, but sound formats that are generally a good choice are Ogg Vorbis and uncompressed WAV. See the implementation-specific information for a full list of supported formats.

volume
The volume of the sound as a value from `0` to `1` (`0` for no sound, `1` for maximum volume).

max_play
The maximum number of instances of this sound playing permitted. If a sound is played while this number of the instances of the same sound are already playing, one of the already playing sounds will be stopped before playing the new instance. Set to `None` for no limit.

fname
The file name of the sound given when it was created. (Read-only)

length
The length of the sound in milliseconds. (Read-only)

playing
The number of instances of this sound playing. (Read-only)

rd
Reserved dictionary for internal use by the SGE. (Read-only)

sge.snd.Sound Methods

`Sound.__init__` (*fname*, *volume=1*, *max_play=1*)
Arguments:

- `fname` – The path to the sound file. If set to `None`, this object will not actually play any sound. If this is neither a valid sound file nor `None`, `IOError` is raised.

All other arguments set the respective initial attributes of the sound. See the documentation for `sge.snd.Sound` for more information.

`Sound.play` (*loops=1*, *volume=1*, *balance=0*, *maxtime=None*, *fade_time=None*)
Play the sound.

Arguments:

- `loops` – The number of times to play the sound; set to `None` or `0` to loop indefinitely.
- `volume` – The volume to play the sound at as a factor of `self.volume` (`0` for no sound, `1` for `self.volume`).
- `balance` – The balance of the sound effect on stereo speakers as a float from `-1` to `1`, where `0` is centered (full volume in both speakers), `1` is entirely in the right speaker, and `-1` is entirely in the left speaker.
- `maxtime` – The maximum amount of time to play the sound in milliseconds; set to `None` for no limit.
- `fade_time` – The time in milliseconds over which to fade the sound in; set to `None` or `0` to immediately play the sound at full volume.

`Sound.stop` (*fade_time=None*)
Stop the sound.

Arguments:

- `fade_time` – The time in milliseconds over which to fade the sound out before stopping; set to `None` or `0` to immediately stop the sound.

`Sound.pause` ()
Pause playback of the sound.

`Sound.unpause` ()
Resume playback of the sound if paused.

8.1.2 sge.snd.Music

class `sge.snd.Music` (*fname*, *volume=1*)
This class stores and plays music. Music is very similar to sound effects, but only one music file can be played at a time, and it is more efficient for larger files than `sge.snd.Sound`.

What music formats are supported depends on the implementation of the SGE, but Ogg Vorbis is generally a good choice. See the implementation-specific information for a full list of supported formats.

Note: You should avoid the temptation to use MP3 files; MP3 is a patent-encumbered format, so many systems do not support it and royalties to the patent holders may be required for commercial use. There are many programs which can convert your MP3 files to the free Ogg Vorbis format.

---

volume
The volume of the music as a value from `0` to `1` (`0` for no sound, `1` for maximum volume).

fname
The file name of the music given when it was created. (Read-only)

length
The length of the music in milliseconds. (Read-only)

playing
Whether or not the music is playing. (Read-only)

position
The current position (time) playback of the music is at in milliseconds. (Read-only)

rd
Reserved dictionary for internal use by the SGE. (Read-only)

sge.snd.Music Methods

`Music.__init__` (*fname*, *volume=1*)
Arguments:

- `fname` – The path to the sound file. If set to `None`, this object will not actually play any music. If this is neither a valid sound file nor `None`, `IOError` is raised.

All other arguments set the respective initial attributes of the music. See the documentation for `sge.snd.Music` for more information.

`Music.play` (*start=0*, *loops=1*, *maxtime=None*, *fade_time=None*)
Play the music.

Arguments:

- `start` – The number of milliseconds from the beginning to start playing at.

See the documentation for `sge.snd.Sound.play()` for more information.

`Music.queue` (*start=0*, *loops=1*, *maxtime=None*, *fade_time=None*)
Queue the music for playback.

This will cause the music to be added to a list of music to play in order, after the previous music has finished playing.

See the documentation for `sge.snd.Music.play()` for more information.

static `Music.stop` (*fade_time=None*)
Stop the currently playing music.

See the documentation for `sge.snd.Sound.stop()` for more information.

static `Music.pause` ()
Pause playback of the currently playing music.

static `Music.unpause` ()
Resume playback of the currently playing music if paused.

static `Music.clear_queue` ()
Clear the music queue.

8.2 sge.snd Functions

8.2.1 sge.snd.stop_all

sge.snd.**stop_all**()
Stop playback of all sounds.

CHAPTER

NINE

SGE.COLLISION

Contents

- sge.collision
  - sge.collision Functions

This module provides easy-to-use collision detection functions, from basic rectangle-based collision detection to shape-based collision detection.

9.1 sge.collision Functions

`sge.collision.`**`rectangles_collide`**(*x1*, *y1*, *w1*, *h1*, *x2*, *y2*, *w2*, *h2*)

Return whether or not two rectangles collide.

Arguments:

- `x1` – The horizontal position of the first rectangle.
- `y1` – The vertical position of the first rectangle.
- `w1` – The width of the first rectangle.
- `h1` – The height of the first rectangle.
- `x2` – The horizontal position of the second rectangle.
- `y2` – The vertical position of the second rectangle.
- `w2` – The width of the second rectangle.
- `h2` – The height of the second rectangle.

`sge.collision.`**`masks_collide`**(*x1*, *y1*, *mask1*, *x2*, *y2*, *mask2*)

Return whether or not two masks collide.

Arguments:

- `x1` – The horizontal position of the first mask.
- `y1` – The vertical position of the first mask.
- `mask1` – The first mask (see below).
- `x2` – The horizontal position of the second mask.
- `y2` – The vertical position of the second mask.
- `mask2` – The second mask (see below).

mask1 and mask2 are both lists of lists of boolean values. Each value in the mask indicates whether or not a pixel is counted as a collision; the masks collide if at least one pixel at the same location is True for both masks.

Masks are indexed as mask[x][y], where x is the column and y is the row.

sge.collision.**rectangle**(*x*, *y*, *w*, *h*, *other=None*)
Return a list of objects colliding with a rectangle.

Arguments:

- x – The horizontal position of the rectangle.
- y – The vertical position of the rectangle.
- w – The width of the rectangle.
- h – The height of the rectangle.
- other – What to check for collisions with. See the documentation for sge.dsp.Object.collision() for more information.

sge.collision.**ellipse**(*x*, *y*, *w*, *h*, *other=None*)
Return a list of objects colliding with an ellipse.

Arguments:

- x – The horizontal position of the imaginary rectangle containing the ellipse.
- y – The vertical position of the imaginary rectangle containing the ellipse.
- w – The width of the ellipse.
- h – The height of the ellipse.
- other – What to check for collisions with. See the documentation for sge.dsp.Object.collision() for more information.

sge.collision.**circle**(*x*, *y*, *radius*, *other=None*)
Return a list of objects colliding with a circle.

Arguments:

- x – The horizontal position of the center of the circle.
- y – The vertical position of the center of the circle.
- radius – The radius of the circle.
- other – What to check for collisions with. See the documentation for sge.dsp.Object.collision() for more information.

sge.collision.**line**(*x1*, *y1*, *x2*, *y2*, *other=None*)
Return a list of objects colliding with a line segment.

Arguments:

- x1 – The horizontal position of the first endpoint of the line segment.
- y1 – The vertical position of the first endpoint of the line segment.
- x2 – The horizontal position of the second endpoint of the line segment.
- y2 – The vertical position of the second endpoint of the line segment.
- other – What to check for collisions with. See the documentation for sge.dsp.Object.collision() for more information.

CHAPTER

TEN

SGE.JOYSTICK

Contents

- sge.joystick
  - sge.joystick Functions

This module provides functions related to joystick input.

10.1 sge.joystick Functions

`sge.joystick.`**`refresh`**`()`
Refresh the SGE's knowledge of joysticks.

Call this method to allow the SGE to use joysticks that were plugged in while the game was running.

`sge.joystick.`**`get_axis`**`(`*joystick*, *axis*`)`
Return the position of a joystick axis as a float from `-1` to `1`, where `0` is centered, `-1` is all the way to the left or up, and `1` is all the way to the right or down. Return `0` if the requested joystick or axis does not exist.

Arguments:

- `joystick` – The number of the joystick to check, where `0` is the first joystick, or the name of the joystick to check.
- `axis` – The number of the axis to check, where `0` is the first axis of the joystick.

`sge.joystick.`**`get_hat_x`**`(`*joystick*, *hat*`)`
Return the horizontal position of a joystick hat (d-pad). Can be `-1` (left), `0` (centered), or `1` (right). Return `0` if the requested joystick or hat does not exist.

Arguments:

- `joystick` – The number of the joystick to check, where `0` is the first joystick, or the name of the joystick to check.
- `hat` – The number of the hat to check, where `0` is the first hat of the joystick.

`sge.joystick.`**`get_hat_y`**`(`*joystick*, *hat*`)`
Return the vertical position of a joystick hat (d-pad). Can be `-1` (up), `0` (centered), or `1` (down). Return `0` if the requested joystick or hat does not exist.

Arguments:

- `joystick` – The number of the joystick to check, where `0` is the first joystick, or the name of the joystick to check.

- `hat` – The number of the hat to check, where `0` is the first hat of the joystick.

`sge.joystick.`**`get_pressed`**(*joystick*, *button*)
Return whether or not a joystick button is pressed, or `False` if the requested joystick or button does not exist.

Arguments:

- `joystick` – The number of the joystick to check, where `0` is the first joystick, or the name of the joystick to check.
- `button` – The number of the button to check, where `0` is the first button of the joystick.

`sge.joystick.`**`get_joysticks`**()
Return the number of joysticks available.

`sge.joystick.`**`get_name`**(*joystick*)
Return the name of a joystick, or `None` if the requested joystick does not exist.

Arguments:

- `joystick` – The number of the joystick to check, where `0` is the first joystick, or the name of the joystick to check.

`sge.joystick.`**`get_id`**(*joystick*)
Return the number of a joystick, where `0` is the first joystick, or `None` if the requested joystick does not exist.

Arguments:

- `joystick` – The number of the joystick to check, where `0` is the first joystick, or the name of the joystick to check.

`sge.joystick.`**`get_axes`**(*joystick*)
Return the number of axes available on a joystick, or `0` if the requested joystick does not exist.

Arguments:

- `joystick` – The number of the joystick to check, where `0` is the first joystick, or the name of the joystick to check.

`sge.joystick.`**`get_hats`**(*joystick*)
Return the number of hats (d-pads) available on a joystick, or `0` if the requested joystick does not exist.

Arguments:

- `joystick` – The number of the joystick to check, where `0` is the first joystick, or the name of the joystick to check.

`sge.joystick.`**`get_trackballs`**(*joystick*)
Return the number of trackballs available on a joystick, or `0` if the requested joystick does not exist.

Arguments:

- `joystick` – The number of the joystick to check, where `0` is the first joystick, or the name of the joystick to check.

`sge.joystick.`**`get_buttons`**(*joystick*)
Return the number of buttons available on a joystick, or `0` if the requested joystick does not exist.

Arguments:

- `joystick` – The number of the joystick to check, where `0` is the first joystick, or the name of the joystick to check.

CHAPTER

ELEVEN

SGE.KEYBOARD

Contents

- sge.keyboard
  - sge.keyboard Functions

This module provides functions related to keyboard input.

11.1 sge.keyboard Functions

`sge.keyboard.`**`get_pressed`**`(`*`key`*`)`
Return whether or not a key is pressed.

See the documentation for `sge.input.KeyPress` for more information.

`sge.keyboard.`**`get_modifier`**`(`*`key`*`)`
Return whether or not a modifier key is being held.

Arguments:

- `key` – The identifier string of the modifier key to check; see the table below.

Modifier Key Name	Identifier String
Alt	`"alt"`
Left Alt	`"alt_left"`
Right Alt	`"alt_right"`
Ctrl	`"ctrl"`
Left Ctrl	`"ctrl_left"`
Right Ctrl	`"ctrl_right"`
Meta	`"meta"`
Left Meta	`"meta_left"`
Right Meta	`"meta_right"`
Shift	`"shift"`
Left Shift	`"shift_left"`
Right Shift	`"shift_right"`
Mode	`"mode"`
Caps Lock	`"caps_lock"`
Num Lock	`"num_lock"`

`sge.keyboard.`**`get_focused`**`()`
Return whether or not the game has keyboard focus.

sge.keyboard.**set_repeat**(*enabled=True*, *interval=0*, *delay=0*)
Set repetition of key press events.

Arguments:

- `enabled` – Whether or not to enable repetition of key press events.
- `interval` – The time in milliseconds in between each repeated key press event.
- `delay` – The time in milliseconds to wait after the first key press event before repeating key press events.

If `enabled` is set to true, this causes a key being held down to generate additional key press events as long as it remains held down.

sge.keyboard.**get_repeat_enabled**()
Return whether or not repetition of key press events is enabled.

See the documentation for `sge.keyboard.set_repeat()` for more information.

sge.keyboard.**get_repeat_interval**()
Return the interval in between each repeated key press event.

See the documentation for `sge.keyboard.set_repeat()` for more information.

sge.keyboard.**get_repeat_delay**()
Return the delay before repeating key press events.

See the documentation for `sge.keyboard.set_repeat()` for more information.

CHAPTER

TWELVE

SGE.MOUSE

Contents

- sge.mouse
  - sge.mouse Functions

This module provides functions related to the mouse input.

Some other mouse functionalities are provided through attributes of `sge.game.mouse`. These attributes are listed below.

The mouse can be in either absolute or relative mode. In absolute mode, the mouse has a position. In relative mode, the mouse only moves. Which mode the mouse is in depends on the values of `sge.game.grab_input` and `sge.game.mouse.visible`.

`sge.game.mouse.`**`x`**
`sge.game.mouse.`**`y`**
: If the mouse is in absolute mode and within a view port, these attributes indicate the position of the mouse in the room, based on its proximity to the view it is in. Otherwise, they will return `-1`.

  These attributes can be assigned to safely, but doing so will not have any effect.

`sge.game.mouse.`**`z`**
: The Z-axis position of the mouse cursor's projection in relation to other window projections. The default value is `10000`.

`sge.game.mouse.`**`sprite`**
: Determines what sprite will be used to represent the mouse cursor. Set to `None` for the default mouse cursor.

`sge.game.mouse.`**`visible`**
: Controls whether or not the mouse cursor is visible. If this is `False` and `sge.game.grab_input` is `True`, the mouse will be in relative mode. Otherwise, the mouse will be in absolute mode.

12.1 sge.mouse Functions

`sge.mouse.`**`get_pressed`**`(`*`button`*`)`
: Return whether or not a mouse button is pressed.

  See the documentation for `sge.input.MouseButtonPress` for more information.

`sge.mouse.`**`get_x`**`()`
: Return the horizontal location of the mouse cursor.

The location returned is relative to the window, excluding any scaling, pillarboxes, and letterboxes. If the mouse is in relative mode, this function returns `None`.

`sge.mouse.`**`get_y`**`()`
Return the vertical location of the mouse cursor.

The location returned is relative to the window, excluding any scaling, pillarboxes, and letterboxes. If the mouse is in relative mode, this function returns `None`.

`sge.mouse.`**`set_x`**`(`*value*`)`
Set the horizontal location of the mouse cursor.

The location returned is relative to the window, excluding any scaling, pillarboxes, and letterboxes. If the mouse is in relative mode, this function has no effect.

`sge.mouse.`**`set_y`**`(`*value*`)`
Set the vertical location of the mouse cursor.

The location returned is relative to the window, excluding any scaling, pillarboxes, and letterboxes. If the mouse is in relative mode, this function has no effect.

CHAPTER

THIRTEEN

INDICES AND TABLES

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

S

sge, 1
sge.collision, 81
sge.dsp, 41
sge.gfx, 63
sge.input, 35
sge.joystick, 83
sge.keyboard, 85
sge.mouse, 87
sge.snd, 77

Symbols

__init__() (sge.dsp.Game method), 43
__init__() (sge.dsp.Object method), 59
__init__() (sge.dsp.Room method), 49
__init__() (sge.dsp.View method), 55
__init__() (sge.gfx.Background method), 75
__init__() (sge.gfx.BackgroundLayer method), 75
__init__() (sge.gfx.Font method), 73
__init__() (sge.gfx.Sprite method), 65
__init__() (sge.snd.Music method), 79
__init__() (sge.snd.Sound method), 78

A

active (sge.dsp.Object attribute), 56
add() (sge.dsp.Room method), 49
alarms (sge.dsp.Game attribute), 42
alarms (sge.dsp.Object attribute), 58
alarms (sge.dsp.Room attribute), 48
alpha (sge.gfx.Color attribute), 63
append_frame() (sge.gfx.Sprite method), 65
axis (sge.input.JoystickAxisMove attribute), 39

B

Background (class in sge.gfx), 75
background (sge.dsp.Room attribute), 48
background_x (sge.dsp.Room attribute), 48
background_y (sge.dsp.Room attribute), 48
BackgroundLayer (class in sge.gfx), 74
ball (sge.input.JoystickTrackballMove attribute), 39
bbox_bottom (sge.dsp.Object attribute), 57
bbox_height (sge.dsp.Object attribute), 57
bbox_height (sge.gfx.Sprite attribute), 65
bbox_left (sge.dsp.Object attribute), 57
bbox_right (sge.dsp.Object attribute), 57
bbox_top (sge.dsp.Object attribute), 57
bbox_width (sge.dsp.Object attribute), 57
bbox_width (sge.gfx.Sprite attribute), 65
bbox_x (sge.dsp.Object attribute), 57
bbox_x (sge.gfx.Sprite attribute), 64
bbox_y (sge.dsp.Object attribute), 57
bbox_y (sge.gfx.Sprite attribute), 65
blue (sge.gfx.Color attribute), 63
bold (sge.gfx.Font attribute), 73
button (sge.input.JoystickButtonPress attribute), 39
button (sge.input.JoystickButtonRelease attribute), 40
button (sge.input.MouseButtonPress attribute), 38
button (sge.input.MouseButtonRelease attribute), 38

C

char (sge.input.KeyPress attribute), 35
checks_collisions (sge.dsp.Object attribute), 56
circle() (in module sge.collision), 82
clear_queue() (sge.snd.Music static method), 79
collision() (sge.dsp.Object method), 59
collision_ellipse (sge.dsp.Object attribute), 57
collision_events_enabled (sge.dsp.Game attribute), 42
collision_precise (sge.dsp.Object attribute), 57
Color (class in sge.gfx), 63
color (sge.gfx.Background attribute), 75
copy() (sge.gfx.Sprite method), 71
create() (sge.dsp.Object class method), 60
current_room (sge.dsp.Game attribute), 43

D

delete_frame() (sge.gfx.Sprite method), 66
delta (sge.dsp.Game attribute), 42
delta_max (sge.dsp.Game attribute), 42
delta_min (sge.dsp.Game attribute), 42
destroy() (sge.dsp.Object method), 60
draw_circle() (sge.gfx.Sprite method), 67
draw_clear() (sge.gfx.Sprite method), 69
draw_dot() (sge.gfx.Sprite method), 66
draw_ellipse() (sge.gfx.Sprite method), 67
draw_erase() (sge.gfx.Sprite method), 69
draw_line() (sge.gfx.Sprite method), 66
draw_lock() (sge.gfx.Sprite method), 70
draw_polygon() (sge.gfx.Sprite method), 67
draw_rectangle() (sge.gfx.Sprite method), 67
draw_sprite() (sge.gfx.Sprite method), 68
draw_text() (sge.gfx.Sprite method), 68
draw_unlock() (sge.gfx.Sprite method), 70

E

ellipse() (in module sge.collision), 82

end() (sge.dsp.Game method), 43
event_alarm() (sge.dsp.Game method), 46
event_alarm() (sge.dsp.Object method), 60
event_alarm() (sge.dsp.Room method), 53
event_animation_end() (sge.dsp.Object method), 60
event_close() (sge.dsp.Game method), 47
event_close() (sge.dsp.Room method), 54
event_collision() (sge.dsp.Object method), 61
event_create() (sge.dsp.Object method), 60
event_destroy() (sge.dsp.Object method), 60
event_gain_keyboard_focus() (sge.dsp.Game method), 47
event_gain_keyboard_focus() (sge.dsp.Room method), 53
event_gain_mouse_focus() (sge.dsp.Game method), 47
event_gain_mouse_focus() (sge.dsp.Room method), 54
event_joystick_axis_move() (sge.dsp.Game method), 46
event_joystick_axis_move() (sge.dsp.Object method), 61
event_joystick_axis_move() (sge.dsp.Room method), 53
event_joystick_button_press() (sge.dsp.Game method), 47
event_joystick_button_press() (sge.dsp.Object method), 61
event_joystick_button_press() (sge.dsp.Room method), 53
event_joystick_button_release() (sge.dsp.Game method), 47
event_joystick_button_release() (sge.dsp.Object method), 61
event_joystick_button_release() (sge.dsp.Room method), 53
event_joystick_hat_move() (sge.dsp.Game method), 46
event_joystick_hat_move() (sge.dsp.Object method), 61
event_joystick_hat_move() (sge.dsp.Room method), 53
event_joystick_trackball_move() (sge.dsp.Game method), 47
event_joystick_trackball_move() (sge.dsp.Object method), 61
event_joystick_trackball_move() (sge.dsp.Room method), 53
event_key_press() (sge.dsp.Game method), 46
event_key_press() (sge.dsp.Object method), 60
event_key_press() (sge.dsp.Room method), 53
event_key_release() (sge.dsp.Game method), 46
event_key_release() (sge.dsp.Object method), 60
event_key_release() (sge.dsp.Room method), 53
event_lose_keyboard_focus() (sge.dsp.Game method), 47
event_lose_keyboard_focus() (sge.dsp.Room method), 53
event_lose_mouse_focus() (sge.dsp.Game method), 47
event_lose_mouse_focus() (sge.dsp.Room method), 54
event_mouse_button_press() (sge.dsp.Game method), 46
event_mouse_button_press() (sge.dsp.Object method), 61
event_mouse_button_press() (sge.dsp.Room method), 53
event_mouse_button_release() (sge.dsp.Game method), 46
event_mouse_button_release() (sge.dsp.Object method), 61
event_mouse_button_release() (sge.dsp.Room method), 53
event_mouse_collision() (sge.dsp.Game method), 47
event_mouse_move() (sge.dsp.Game method), 46
event_mouse_move() (sge.dsp.Object method), 60
event_mouse_move() (sge.dsp.Room method), 53
event_paused_close() (sge.dsp.Game method), 48
event_paused_close() (sge.dsp.Room method), 54
event_paused_gain_keyboard_focus() (sge.dsp.Game method), 48
event_paused_gain_keyboard_focus() (sge.dsp.Room method), 54
event_paused_gain_mouse_focus() (sge.dsp.Game method), 48
event_paused_gain_mouse_focus() (sge.dsp.Room method), 54
event_paused_joystick_axis_move() (sge.dsp.Game method), 47
event_paused_joystick_axis_move() (sge.dsp.Object method), 62
event_paused_joystick_axis_move() (sge.dsp.Room method), 54
event_paused_joystick_button_press() (sge.dsp.Game method), 47
event_paused_joystick_button_press() (sge.dsp.Object method), 62
event_paused_joystick_button_press() (sge.dsp.Room method), 54
event_paused_joystick_button_release() (sge.dsp.Game method), 47
event_paused_joystick_button_release() (sge.dsp.Object method), 62
event_paused_joystick_button_release() (sge.dsp.Room method), 54
event_paused_joystick_hat_move() (sge.dsp.Game method), 47
event_paused_joystick_hat_move() (sge.dsp.Object method), 62
event_paused_joystick_hat_move() (sge.dsp.Room method), 54
event_paused_joystick_trackball_move() (sge.dsp.Game method), 47
event_paused_joystick_trackball_move() (sge.dsp.Object method), 62
event_paused_joystick_trackball_move() (sge.dsp.Room method), 54
event_paused_key_press() (sge.dsp.Game method), 47
event_paused_key_press() (sge.dsp.Object method), 62
event_paused_key_press() (sge.dsp.Room method), 54
event_paused_key_release() (sge.dsp.Game method), 47
event_paused_key_release() (sge.dsp.Object method), 62
event_paused_key_release() (sge.dsp.Room method), 54

event_paused_lose_keyboard_focus() (sge.dsp.Game method), 48
event_paused_lose_keyboard_focus() (sge.dsp.Room method), 54
event_paused_lose_mouse_focus() (sge.dsp.Game method), 48
event_paused_lose_mouse_focus() (sge.dsp.Room method), 54
event_paused_mouse_button_press() (sge.dsp.Game method), 47
event_paused_mouse_button_press() (sge.dsp.Object method), 62
event_paused_mouse_button_press() (sge.dsp.Room method), 54
event_paused_mouse_button_release() (sge.dsp.Game method), 47
event_paused_mouse_button_release() (sge.dsp.Object method), 62
event_paused_mouse_button_release() (sge.dsp.Room method), 54
event_paused_mouse_move() (sge.dsp.Game method), 47
event_paused_mouse_move() (sge.dsp.Object method), 62
event_paused_mouse_move() (sge.dsp.Room method), 54
event_paused_step() (sge.dsp.Game method), 47
event_paused_step() (sge.dsp.Object method), 62
event_paused_step() (sge.dsp.Room method), 54
event_room_end() (sge.dsp.Room method), 53
event_room_resume() (sge.dsp.Room method), 53
event_room_start() (sge.dsp.Room method), 53
event_step() (sge.dsp.Game method), 46
event_step() (sge.dsp.Object method), 60
event_step() (sge.dsp.Room method), 53
event_update_position() (sge.dsp.Object method), 61
extend() (sge.gfx.Sprite method), 65

F

flip() (sge.gfx.Sprite method), 70
fname (sge.snd.Music attribute), 79
fname (sge.snd.Sound attribute), 77
Font (class in sge.gfx), 73
fps (sge.dsp.Game attribute), 42
fps (sge.gfx.Sprite attribute), 64
frames (sge.gfx.Sprite attribute), 65
from_screenshot() (sge.gfx.Sprite class method), 72
from_sprite() (sge.gfx.Font class method), 74
from_text() (sge.gfx.Sprite class method), 72
from_tileset() (sge.gfx.Sprite class method), 72
from_tween() (sge.gfx.Sprite class method), 71
fullscreen (sge.dsp.Game attribute), 41

G

Game (class in sge.dsp), 41
get_axes() (in module sge.joystick), 84
get_axis() (in module sge.joystick), 83
get_buttons() (in module sge.joystick), 84
get_focused() (in module sge.keyboard), 85
get_hat_x() (in module sge.joystick), 83
get_hat_y() (in module sge.joystick), 83
get_hats() (in module sge.joystick), 84
get_height() (sge.gfx.Font method), 74
get_id() (in module sge.joystick), 84
get_joysticks() (in module sge.joystick), 84
get_modifier() (in module sge.keyboard), 85
get_name() (in module sge.joystick), 84
get_objects_at() (sge.dsp.Room method), 51
get_pixel() (sge.gfx.Sprite method), 66
get_pixels() (sge.gfx.Sprite method), 66
get_pressed() (in module sge.joystick), 84
get_pressed() (in module sge.keyboard), 85
get_pressed() (in module sge.mouse), 87
get_repeat_delay() (in module sge.keyboard), 86
get_repeat_enabled() (in module sge.keyboard), 86
get_repeat_interval() (in module sge.keyboard), 86
get_trackballs() (in module sge.joystick), 84
get_width() (sge.gfx.Font method), 74
get_x() (in module sge.mouse), 87
get_y() (in module sge.mouse), 88
grab_input (sge.dsp.Game attribute), 42
green (sge.gfx.Color attribute), 63

H

hat (sge.input.JoystickHatMove attribute), 39
height (sge.dsp.Game attribute), 41
height (sge.dsp.Room attribute), 48
height (sge.dsp.View attribute), 55
height (sge.gfx.Sprite attribute), 64
hex_string (sge.gfx.Color attribute), 64
hport (sge.dsp.View attribute), 55

I

image_alpha (sge.dsp.Object attribute), 58
image_blend (sge.dsp.Object attribute), 58
image_fps (sge.dsp.Object attribute), 58
image_index (sge.dsp.Object attribute), 58
image_origin_x (sge.dsp.Object attribute), 58
image_origin_y (sge.dsp.Object attribute), 58
image_rotation (sge.dsp.Object attribute), 58
image_speed (sge.dsp.Object attribute), 58
image_xscale (sge.dsp.Object attribute), 58
image_yscale (sge.dsp.Object attribute), 58
input_events (sge.dsp.Game attribute), 42
insert_frame() (sge.gfx.Sprite method), 65
italic (sge.gfx.Font attribute), 73

J

JoystickAxisMove (class in sge.input), 38

JoystickButtonPress (class in sge.input), 39
JoystickButtonRelease (class in sge.input), 39
JoystickHatMove (class in sge.input), 39
JoystickTrackballMove (class in sge.input), 39
js_id (sge.input.JoystickAxisMove attribute), 39
js_id (sge.input.JoystickButtonPress attribute), 39
js_id (sge.input.JoystickButtonRelease attribute), 40
js_id (sge.input.JoystickHatMove attribute), 39
js_id (sge.input.JoystickTrackballMove attribute), 39
js_name (sge.input.JoystickAxisMove attribute), 39
js_name (sge.input.JoystickButtonPress attribute), 39
js_name (sge.input.JoystickButtonRelease attribute), 40
js_name (sge.input.JoystickHatMove attribute), 39
js_name (sge.input.JoystickTrackballMove attribute), 39

K

key (sge.input.KeyPress attribute), 35
key (sge.input.KeyRelease attribute), 38
KeyboardFocusGain (class in sge.input), 40
KeyboardFocusLose (class in sge.input), 40
KeyPress (class in sge.input), 35
KeyRelease (class in sge.input), 38

L

layers (sge.gfx.Background attribute), 75
length (sge.snd.Music attribute), 79
length (sge.snd.Sound attribute), 77
line() (in module sge.collision), 82

M

mask (sge.dsp.Object attribute), 58
mask_x (sge.dsp.Object attribute), 59
mask_y (sge.dsp.Object attribute), 59
masks_collide() (in module sge.collision), 81
max_play (sge.snd.Sound attribute), 77
mirror() (sge.gfx.Sprite method), 70
mouse (sge.dsp.Game attribute), 43
MouseButtonPress (class in sge.input), 38
MouseButtonRelease (class in sge.input), 38
MouseFocusGain (class in sge.input), 40
MouseFocusLose (class in sge.input), 40
MouseMove (class in sge.input), 38
move_direction (sge.dsp.Object attribute), 57
move_x() (sge.dsp.Object method), 59
move_y() (sge.dsp.Object method), 59
Music (class in sge.snd), 78

N

name (sge.gfx.Font attribute), 73
name (sge.gfx.Sprite attribute), 65

O

Object (class in sge.dsp), 56
object_area_height (sge.dsp.Room attribute), 48
object_area_void (sge.dsp.Room attribute), 49
object_area_width (sge.dsp.Room attribute), 48
object_areas (sge.dsp.Room attribute), 48
objects (sge.dsp.Room attribute), 48
origin_x (sge.gfx.Sprite attribute), 64
origin_y (sge.gfx.Sprite attribute), 64

P

pause() (sge.dsp.Game method), 43
pause() (sge.snd.Music static method), 79
pause() (sge.snd.Sound method), 78
play() (sge.snd.Music method), 79
play() (sge.snd.Sound method), 78
playing (sge.snd.Music attribute), 79
playing (sge.snd.Sound attribute), 77
position (sge.snd.Music attribute), 79
project_circle() (sge.dsp.Game method), 45
project_circle() (sge.dsp.Room method), 52
project_dot() (sge.dsp.Game method), 44
project_dot() (sge.dsp.Room method), 51
project_ellipse() (sge.dsp.Game method), 45
project_ellipse() (sge.dsp.Room method), 52
project_line() (sge.dsp.Game method), 44
project_line() (sge.dsp.Room method), 51
project_polygon() (sge.dsp.Game method), 45
project_polygon() (sge.dsp.Room method), 52
project_rectangle() (sge.dsp.Game method), 45
project_rectangle() (sge.dsp.Room method), 51
project_sprite() (sge.dsp.Game method), 45
project_sprite() (sge.dsp.Room method), 52
project_text() (sge.dsp.Game method), 46
project_text() (sge.dsp.Room method), 52
pump_input() (sge.dsp.Game method), 43

Q

queue() (sge.snd.Music method), 79
QuitRequest (class in sge.input), 40

R

rd (sge.dsp.Object attribute), 59
rd (sge.dsp.Room attribute), 49
rd (sge.dsp.View attribute), 55
rd (sge.gfx.Background attribute), 75
rd (sge.gfx.BackgroundLayer attribute), 75
rd (sge.gfx.Font attribute), 73
rd (sge.gfx.Sprite attribute), 65
rd (sge.snd.Music attribute), 79
rd (sge.snd.Sound attribute), 77
rectangle() (in module sge.collision), 82
rectangles_collide() (in module sge.collision), 81
red (sge.gfx.Color attribute), 63
refresh() (in module sge.joystick), 83
refresh() (sge.dsp.Game method), 44

regulate_origin (sge.dsp.Object attribute), 57
regulate_speed() (sge.dsp.Game method), 44
remove() (sge.dsp.Room method), 50
repeat_down (sge.gfx.BackgroundLayer attribute), 75
repeat_left (sge.gfx.BackgroundLayer attribute), 75
repeat_right (sge.gfx.BackgroundLayer attribute), 75
repeat_up (sge.gfx.BackgroundLayer attribute), 75
resize_canvas() (sge.gfx.Sprite method), 70
Room (class in sge.dsp), 48
rotate() (sge.gfx.Sprite method), 71

S

save() (sge.gfx.Sprite method), 71
scale (sge.dsp.Game attribute), 42
scale() (sge.gfx.Sprite method), 70
scale_proportional (sge.dsp.Game attribute), 42
scale_smooth (sge.dsp.Game attribute), 42
set_repeat() (in module sge.keyboard), 85
set_x() (in module sge.mouse), 88
set_y() (in module sge.mouse), 88
sge (module), 1
sge.BLEND_NORMAL (in module sge), 3
sge.BLEND_RGB_ADD (in module sge), 3
sge.BLEND_RGB_MAXIMUM (in module sge), 3
sge.BLEND_RGB_MINIMUM (in module sge), 3
sge.BLEND_RGB_MULTIPLY (in module sge), 3
sge.BLEND_RGB_SCREEN (in module sge), 3
sge.BLEND_RGB_SUBTRACT (in module sge), 3
sge.BLEND_RGBA_ADD (in module sge), 3
sge.BLEND_RGBA_MAXIMUM (in module sge), 3
sge.BLEND_RGBA_MINIMUM (in module sge), 3
sge.BLEND_RGBA_MULTIPLY (in module sge), 3
sge.BLEND_RGBA_SCREEN (in module sge), 3
sge.BLEND_RGBA_SUBTRACT (in module sge), 3
sge.collision (module), 81
sge.dsp (module), 41
sge.game (in module sge), 3
sge.gfx (module), 63
sge.IMPLEMENTATION (in module sge), 3
sge.input (module), 35
sge.joystick (module), 83
sge.keyboard (module), 85
sge.mouse (module), 87
sge.snd (module), 77
size (sge.gfx.Font attribute), 73
Sound (class in sge.snd), 77
speed (sge.dsp.Object attribute), 57
speed (sge.gfx.Sprite attribute), 64
Sprite (class in sge.gfx), 64
sprite (sge.dsp.Object attribute), 56
sprite (sge.gfx.BackgroundLayer attribute), 74
sprite (sge.mouse.sge.game.mouse attribute), 87
start() (sge.dsp.Game method), 43
start() (sge.dsp.Room method), 50
start_room (sge.dsp.Game attribute), 43
stop() (sge.snd.Music static method), 79
stop() (sge.snd.Sound method), 78
stop_all() (in module sge.snd), 80

T

tangible (sge.dsp.Object attribute), 56
transparent (sge.gfx.Sprite attribute), 64

U

underline (sge.gfx.Font attribute), 73
unpause() (sge.dsp.Game method), 43
unpause() (sge.snd.Music static method), 79
unpause() (sge.snd.Sound method), 78

V

value (sge.input.JoystickAxisMove attribute), 39
View (class in sge.dsp), 55
views (sge.dsp.Room attribute), 48
visible (sge.dsp.Object attribute), 56
visible (sge.mouse.sge.game.mouse attribute), 87
volume (sge.snd.Music attribute), 78
volume (sge.snd.Sound attribute), 77

W

width (sge.dsp.Game attribute), 41
width (sge.dsp.Room attribute), 48
width (sge.dsp.View attribute), 55
width (sge.gfx.Sprite attribute), 64
window_icon (sge.dsp.Game attribute), 42
window_text (sge.dsp.Game attribute), 42
wport (sge.dsp.View attribute), 55

X

x (sge.dsp.Object attribute), 56
x (sge.dsp.View attribute), 55
x (sge.gfx.BackgroundLayer attribute), 74
x (sge.input.JoystickHatMove attribute), 39
x (sge.input.JoystickTrackballMove attribute), 39
x (sge.input.MouseMove attribute), 38
x (sge.mouse.sge.game.mouse attribute), 87
xacceleration (sge.dsp.Object attribute), 57
xdeceleration (sge.dsp.Object attribute), 58
xport (sge.dsp.View attribute), 55
xprevious (sge.dsp.Object attribute), 59
xscroll_rate (sge.gfx.BackgroundLayer attribute), 75
xstart (sge.dsp.Object attribute), 58
xvelocity (sge.dsp.Object attribute), 57

Y

y (sge.dsp.Object attribute), 56
y (sge.dsp.View attribute), 55
y (sge.gfx.BackgroundLayer attribute), 74

y (sge.input.JoystickHatMove attribute), 39
y (sge.input.JoystickTrackballMove attribute), 39
y (sge.input.MouseMove attribute), 38
y (sge.mouse.sge.game.mouse attribute), 87
yacceleration (sge.dsp.Object attribute), 57
ydeceleration (sge.dsp.Object attribute), 58
yport (sge.dsp.View attribute), 55
yprevious (sge.dsp.Object attribute), 59
yscroll_rate (sge.gfx.BackgroundLayer attribute), 75
ystart (sge.dsp.Object attribute), 58
yvelocity (sge.dsp.Object attribute), 57

Z

z (sge.dsp.Object attribute), 56
z (sge.gfx.BackgroundLayer attribute), 75
z (sge.mouse.sge.game.mouse attribute), 87