Platipy Documentation

Release 0.9.6

Robert Deaton, Austin Bart

August 27, 2014

Contents

1 News **3**
- 1.1 News . . . 3

2 Building Games **5**
- 2.1 Introduction and Target Audience . . . 5
- 2.2 Tutorials . . . 8

3 Spyral Documentation **43**
- 3.1 Complete API Reference . . . 43
- 3.2 Spyral API Appendices . . . 67
- 3.3 Spyral API Cheat Sheets . . . 70

4 Further References **73**
- 4.1 Game Development Resources . . . 73
- 4.2 Educational Game Design . . . 73

5 The Platipy Project **81**
- 5.1 OLPC Contributor Application . . . 81
- 5.2 Games Gallery . . . 85
- 5.3 Open Tasks . . . 85

6 Release Information **87**
- 6.1 Latest Versions . . . 87
- 6.2 Changelogs . . . 88
- 6.3 Contact Developers / Submit Changes . . . 91

Python Module Index **93**

The platipy project is an effort to document efforts to build a game in Python and Spyral for the OLPC XO (and other platforms, such as Windows, Mac, and the Raspberry Pi). Spyral is a library/engine for developing 2D games in Python 2.X, with a focus in rapid development and clean design. Any system that runs Pygame should be able to run Spyral. Read the *introduction* if you're interested in the motivation and some details behind choices made, or skip right to the *guides* if you're ready to start programming.

CHAPTER 1

News

1.1 News

1.1.1 Release of Spyral 0.9.6

4:08pm April 11, 2014

Today we are pleased to announce the first release of Spyral 0.9.6, a bugfix of our first release using the new API. This API represents a years work of changes to the very core of Spyral, coming with new, powerful features like improved event handling, Views, Animations, and a host of other cool stuff.

CHAPTER 2

Building Games

2.1 Introduction and Target Audience

This website is a collection of wisdom learned about developing games for the OLPC XO, with a special focus on the goals of the students taking CISC-374 at the University of Delaware. We won't talk too much about developing individual games, as that varies widely from year to year and team to team, but instead we'll focus on the differences and particular challenges that are faced when placing these games on the OLPC, and some ways to overcome those challenges. Each section will conclude with links for additional reading that will often include more depth and breadth than will be presented here.

While we will focus mostly on the needs of the course at the University of Delaware, most of the material is generally applicable for developing non-PyGTK activities for the OLPC XO. We will begin with a quick introduction to the One-Laptop-Per-Child project and the XO laptop, its flagship product. Spyral, and this material, is also suitable for other platforms, such as the Raspberry Pi (which ships with Pygame installed!).

2.1.1 The OLPC XO

The XO was chosen as a development environment for a number of reasons. Probably the most important reason is that the middle school with which CISC374 coordinates received a donation that allows them to assign one XO to every student at the school. There is also a large educational development community around the XO and the OLPC foundation, so the hope is that the results of this course can be released to the greater community.

Sugar and Activities

Sugar is the user interface of the XO laptop. It is designed very differently from traditional systems like Windows or OS X. Built on Fedora, there are several concepts unique to the system that you should understand before designing for it.

Key Concepts

Activities Basically, an application or program. Activities are meant to be used by humans. When they run, they take up the whole screen, and it is expected that you only use one Activity at a time.

Views There are three "Views" on the XO: "Home", "Neighborhood", "Group". The **Home View** is a listing of all the Activities available on the XO. The **Neighborhood View** is a list of all the visible networks you can connect to. You will never use the **Group View**. You can switch between Views by using the buttons in the top-left of the Frame.

Frame If you hold your mouse in the corner of the screen (or press the black rectangle in the top-right corner of your keyboard), the Frame will appear. On the top bar, there are buttons to switch between the Views and running Activites. On the bottom, you can see currently connected networks, USB drives, and other system information.

Journal A record of activity on the XO. When you start an Activity, an entry is added to the journal. When you download a file, it is stored in the journal. When you resume a stopped Activity, its state is restored from the journal.

Important Built-in Activities

Terminal The most important Activity, this is a Linux terminal. You can use it to access the underlying Linux filesystem.

Log This is the second most important Activity. When you get error messages, this is where they will show up. The entries on the left are from the Activities and Services that are running or have finished since you turned on the XO. Click on them to see their output. This is the only way to get to error messages!

Browse When you need to browse the internet, you'll use this web browser. It's basically a dumbed-down version of Firefox. You can also install Opera or Firefox if you want, but it probably won't be necessary.

Maze The maze is a fun little activity where you guide a shape out of a maze. Everytime you complete a level, a larger one is generated. For some reason, people love the Maze a lot, and it makes for a good distraction.

If you're interested in more activities, there are a large number of them at the Sugar Activity Repository.

Connecting to the Internet

To get internet access, go to the Neighborhood (in the top-left of the Frame). You should see a bunch of circles. If you hover over these circles, you can see that they represent available networks. Choose your network, and if prompted, put in your passphrase.

If you have problems connecting, try some of these fixes.

- Give the XO a few minutes after you've reached the home screen. Sometimes it just takes a bit of time to get itself oriented.

- Keep the "ears" (the little swiveling bars on the XO that cover the USB ports) upwards. That's where the wifi antennae are.
- Try removing your network history. Right click the XO symbol on the Home View, choose "Control Panel", and go to "Network". There should be a button that promises to "Discard network history". Don't worry if clicking it doesn't appear to do anything. It's a silent fix. Just try connecting again.
- Smack your XO until you feel better.
- A recent research study discovered that when 10+ XO laptops are put together, the mesh network on even a single XO can completely halt the regular internet connection. Disabling the mesh on all the laptops (which can be done with a startup script) should restore functionality. Refer to the following discussion for more information.
- Try the stuff on this page.

2.1.2 Development Options

Python

As with most platforms, there are a number of options available for development. The first choice we needed to make for the XO was language. Python is the language of choice for a number of reasons. First, the vast majority of activities and most of Sugar itself are written in Python, so using the preferred language makes finding examples and documentation significantly easier. Additionally, given the tight schedule for the course, a dynamic language that allows rapid development is a good choice.

Pygame vs PyGTK+

With Python chosen, we have two choices for a user interface that will work with Sugar: pygame and PyGTK+. PyGTK+ is a set of bindings for GTK+, a common GUI toolkit for Desktop applications. While some games may be written easily within the confines of PyGTK+, many of the game ideas which have been proposed would not, requiring much custom development of widgets and internal knowledge of GTK+.

Pygame, however, is a library made specificially for game development. It provides simple and direct ways of drawing and managing 2D images on the screen, making it a great choice for making simple games. It does, however, also have a few downsides. Most notably, hardware support with pygame is notoriously lacking, and further limited by the XO's lack of OpenGL drivers on some models. Additionally, compared to PyGTK+, pygame is a second-class citizen in OLPC development, requiring a number of hacks and workarounds. To remedy this situation, a custom library built on top of pygame, called Spyral, has been developed for this course.

Spyral

In addition to Python and Pygame, the recommendation for this course is Spyral, a library built on top of pygame to provide a number of features which are useful for rapid and efficient game development. Most importantly, spyral helps provide the following:

- Some built-in core concepts of game design. Pygame is really just a wrapper for doing 2D drawing, with a few nice features like sound and input support, but doesn't provide much in terms of higher level game design concepts. Spyral provides a scene system, improved game clocks, an events system, and much more.
- An optimized method of drawing. Because pygame on the XO is not hardware accelerated, pygame's software rendering is the slowest part of every game. Spyral provides a no-hassle method of doing dirty rendering which can increase performance significantly for sprite-based 2D games

Spyral is a complete wrapper on-top of pygame, meaning that the usage of pygame should be completely hidden from the user. For advanced users, pygame is in full use behind the scenes, and with clever reading of the spyral source code, you can use it in your games, but we feel that spyral should be sufficient for most users in this course. If you find yourself in need of a feature, please contact the developers by raising a new issue on the Spyral Github .

2.1.3 Additional Reading

- For some additional information about the OLPC XO, visit the OLPC Wiki, though be warned that there is a lot of out of date information floating around various areas.
- For some additional motivation for Python and PyGTK+ and Pygame, the FLOSS Manuals guide to "Make your own sugar activities" is a good read.
- For a more in-depth look at the motivation behind spyral, see our *Contributor Application*.

2.2 Tutorials

2.2.1 Getting Started

Required Software

Warning: XO laptops do not run the most current version of most of their software. If you want to be the best XO developer possible, you'll want to use the earlier versions. If you cannot get older versions installed, Python 2.7 and the latest version of pygame are acceptable, but make sure you are testing often on the XO. There are also programs that can search your source code and make sure it is 2.5 compliant .

Spyral has several dependencies. Moreover, the installation process is very different on Mac and Windows. Please follow the directions that are appropriate for your system.

Mac

1. Install Python
2. Install setuptools
3. Install pip
4. Install greenlets
5. Install pygame
6. Install parsley
7. `Download Pong`

Unix

1. sudo apt-get install python-setuptools
2. sudo apt-get install python-pip
3. sudo apt-get install python-pygame
4. sudo pip install greenlet

5. sudo pip install parsley
6. `Download Pong`

Windows

There are several steps to getting python and spyral fully running on your PC.

Note: even if you have a 64-bit machine, use the 32-bit version of Python (it's still more stable).

---

1. **Python**: Python is the name of the language and the interpeter. Use 2.7 since 2.5 is no longer easily available on Windows.
2. **Greenlets**: Greenlets is a powerful module for adding multiprocessing. Download version 0.4.2, for windows 32-bit 2.7 Python, of greenlets .
3. **Pygame**: Pygame is the game development library for Python that is installed on the XOs. Download the latest version of pygame .
4. **SetupTools**: This is a python module for quickly installing new python modules. Download version 2.2, for windows 32-bit 2.7 Python of setuptools .
5. **Pip**: SetupTools is a requirement for an even better installer named Pip. Download version 1.5.4, for windows 32-bit 2.7 Python, of pip .
6. **Add Python to your Path**: To be able to directly run python from the command line, you must add the path to Python to your System's Path. This Stack Overflow gives general directions, but you might need to google search for directions that work for your version. Make sure you add both the Python folder itself and the Scripts subfolder folder!
7. **Parsley**: This is a module that Spyral uses to handle Spyral Style Files. Use the following on the command line: `pip install parsley`
8. **Download Pong Example**: `Download Pong`

Note: Use PowerShell instead of the default Windows Command Line. It has a lot of unix features like `ls` and should already be installed on your system.

Setting up the Launcher

The Dev Launcher, named Example.activity, is a tool provided for you that takes a lot of the boilerplate out of writing your game. The download links can be found at *Downloads*, and extract it to whereever you will be working.

Once you've extracted the launcher, you should rename the Example.activity directory to a name of your choice. Since you probably don't yet know the name of your game, you can name it after yourself for now. Once it has been renamed, you should run *init.py* in a terminal. It will prompt you for some values for setting up your activity. Right now, these aren't too important, but come back to this section later when you're ready to pick a name and run your activity on the XO.

The launcher contains a variety of other folders and directories. Many of them will be important later, and a few of them you can ignore entirely. Here's a summary of what's included:

File	Description
activity.py	The activity launcher required for the XO. You should never have to edit this file.
dev_launcher.py	This is the launcher that you will use during development. It supports a variety of options, you should never have to edit this file.
init.py	A script which does some setup for running your game as an activity on the XO. You should never have to edit this file.
setup.py	A script which will provide a number of way for you to deploy your game for testing or when you are ready for release. We'll come back to setup.py in a later chapter.
activity/	This directory contains some metadata required for the XO. It can be modified directly, or generated for you by init.py. Until you have run init.py, this directory will be empty.
dist/	When you are building with *setup.py*, the output will go in here. Any files in this directory will be ignored when building.
game/	This is the directory where all your game assets will go. All the code, artwork, fonts, etc. should be placed in here. This is to facilitate updating the launcher in the future and keeping the directory structure clean.
libraries/	This directory contains any pure python libraries that you wish to distribute with your game.
locale/	This is a build output directory, like *dist*, except for built translations. You should never be placing things in here by hand
po/	This directory contains source files for translations. You can read more about this in the section on Translating
profiles/	This directory will contain the output from the performance profiler built into the development launcher.
skel/	This directory contains support files for init.py. You can safely ignore it.

Running the Example

With the launcher installed, you can run the example game which comes with it, a simple version of Pong. For running on your regular computers, the file dev_launcher.py is the way to launch the game. It comes with a few options, but for now there are two important ones which we'll worry about. The first is *-r*, which allows you to specify a resolution. By default, the launcher will autodetect your screen's resolution. Because the XO uses a screen resolution of 1200 by 900, all games which we write in that class will have that resolution. This means that on most of your machines, the image will be streched because the aspect ratio does not match. For development, you should pick a good resolution which fits within your screen, and pass that as an option to the dev_launcher. For instance, I usually run "python dev_launcher.py -r 800 600".

The second important launcher option is "-h". It will show you other options available in the launcher. We'll come back to those later.

Modifying the Example

Once you're ready to start modifying the example code, head into the *game* directory. Here, you will find the code which is actually of interest to you. In the next chapter, we'll build the game you see in the example from the ground up.

2.2.2 Your First Game: Pong

Now that you've got the example launcher up and working, let's start from scratch and write a game. For the example, we'll write Pong.

Note: To begin, clear out the *game/* directory and follow along as we rewrite it.

In order for the Launcher to find your game, you must make the *game/* directory be an importable python module by creating *game/__init__.py* (you can read more about Python Packaging here). The Launcher expects this file to have a *main* function which pushes the first `Scene` for the game on the `Director`'s stack.

The Director and Scenes

The `Director` and `Scenes` are the most fundamental way to organize a game in Spyral. At any given time, a Scene is running and controlling the game. The Director manages movement between Scenes. The top Scene on the stack is the current Scene, and transitions require:

- `Pushing` new Scenes on top of old ones.
- `Popping` the current Scene.
- `Replacing` the current Scene with a new one.

Your game will have many Scenes (perhaps representing a main menu, a character select screen, an individual level, or a pause menu), but there is only ever the one Director.

Our Pong game will eventually have two Scenes: a simple menu, and the actual Pong game. For now, let's make an empty class to represent the second of those two Scenes. Then we can have the *main* function push that Scene onto the top of the Director's stack. To keep our code organized, we'll split this into multiple files.

game/__init__.py

```
1 import spyral
2 from pong import Pong
3
4 def main():
5     spyral.director.push(Pong())
```

game/pong.py

```
1 import spyral
2
3
4 class Pong(spyral.Scene):
5     pass
```

For now, we will only add in a stub for the Scene's constructor (__init__). Notice how we call the constructor for the Pong classes parent (`spyral.Scene`) by using the `super` python keyword. Whenever you subclass in Python, you should call the super class in this way (More information). Scenes require a size on initialization, and all XO games should have the same size.

Note: If your monitor is not big enough to display a 1200x900 window, you can scale the resolution without affecting your game using the development launcher.

```
>>> .\dev-launcher.py -r 600 450
```

---

game/pong.py

```
import spyral

WIDTH = 1200
HEIGHT = 900
SIZE = (WIDTH, HEIGHT)

class Pong(spyral.Scene):
    def __init__(self):
        super(Pong, self).__init__(SIZE)
```

Before we can set our first scene property, we have to learn about Images.

Images

Images in spyral are the basic building blocks of drawing. They are conceptually simple, but there are many methods to manipulate them. It is worthwhile to spend some time reading the docs on `Images`. To make our background, we will

- create a new image using the Image constructor, sized to the Scene,
- assign it as the `background` for this Scene
- `fill` this image with black, and finally,

game/pong.py

```
import spyral

WIDTH = 1200
HEIGHT = 900
SIZE = (WIDTH, HEIGHT)

class Pong(spyral.Scene):
    def __init__(self):
        super(Pong, self).__init__(SIZE)

        self.background = spyral.Image(size=SIZE).fill((0, 0, 0))
```

Now that we have a background, we'll want to create Images that represent the paddles and ball in Pong. For this, we'll talk about Sprites.

Sprites

Sprites have an Image, along with some information about where and how to draw themselves. Sprites allow us to control things like positioning, scaling, rotation, and more. There are also more advanced Sprites, including ones that can do animation. For now, we'll work with basic sprites, but you can read more about the available sprites in `Sprites`.

All Sprites must have an image and live in a Scene. They cannot move between Scenes, and when a Scene ends, so do the sprites. As soon as Sprites are created, they will start being drawn by the scene (you can stop them from being drawn with the `visible` attribute).

For now, we'll

- create a new Paddle sprite,
- give the Paddle a new image (a solid rectangle),
- create two instances of the Paddle sprites within the scene, and,
- position the sprites close to the left and right of the screen, using the sprite's anchor attribute to improve positioning,

game/pong.py

```
import spyral

WIDTH = 1200
HEIGHT = 900
SIZE = (WIDTH, HEIGHT)

class Paddle(spyral.Sprite):
    def __init__(self, scene):
        super(Paddle, self).__init__(scene)

        self.image = spyral.Image(size=(20, 300)).fill((255, 255, 255))


class Pong(spyral.Scene):
    def __init__(self):
        super(Pong, self).__init__(SIZE)

        self.background = spyral.Image(size=SIZE).fill( (0, 0, 0) )

        self.left_paddle = Paddle(self)
        self.right_paddle = Paddle(self)

        self.left_paddle.anchor = 'midleft'
        self.left_paddle.pos = (20, HEIGHT / 2)

        self.right_paddle.anchor = 'midright'
        self.right_paddle.pos = (WIDTH - 20, HEIGHT / 2)
```

A good rule of thumb is to avoid manipulating sprites at the Scene level. So we'll refactor the positioning and anchors inside the Paddle constructor.

game/pong.py

```
import spyral

WIDTH = 1200
HEIGHT = 900
SIZE = (WIDTH, HEIGHT)

class Paddle(spyral.Sprite):
    def __init__(self, scene, side):
        super(Paddle, self).__init__(scene)

        self.image = spyral.Image(size=(20, 300)).fill((255, 255, 255))

        self.anchor = 'mid' + side
        if side == 'left':
            self.x = 20
        else:
            self.x = WIDTH - 20
        self.y = HEIGHT/2


class Pong(spyral.Scene):
    def __init__(self):
        super(Pong, self).__init__(SIZE)

        self.background = spyral.Image(size=SIZE).fill( (0, 0, 0) )

        self.left_paddle = Paddle(self, 'left')
        self.right_paddle = Paddle(self, 'right')
```

Moving the Ball

Next, we'll add a ball, but we'll treat it differently than the paddles. The ball is going to move on it's own, so we'll make a *Ball* class, inheriting from the *Sprite* class again. We already know how to position, set an image (using the `draw_circle` fuction), and anchor this new sprite.

game/pong.py

```
class Ball(spyral.Sprite):
    def __init__(self, scene):
        super(Ball, self).__init__(scene)

        self.image = spyral.Image(size=(20, 20))
        self.image.draw_circle((255, 255, 255), (10, 10), 10)
        self.anchor = 'center'
        self.pos = (WIDTH/2, HEIGHT/2)
```

To make the ball move every frame, we'll need to register a function of the ball with the *director.update* event. There are many possible events (see the *Event List* for a complete list), and you can even make your own (as we will see later). The *director.update* event is the most common, however. When a method is registered with this event, the method will be called every update.

Additionally, we need to perform some math to calculate the velocity of the ball. In order to reuse this function later, and to keep our code simpler, we can move it to new method that we'll name *reset*.

game/pong.py

```
class Ball(spyral.Sprite):
    def __init__(self, scene):
        super(Ball, self).__init__(scene)

        self.image = spyral.Image(size=(20, 20))
        self.image.draw_circle((255, 255, 255), (10, 10), 10)
        self.anchor = 'center'

        spyral.event.register('director.update', self.update)
        self.reset()

    def update(self, delta):
        self.x += delta * self.vel_x
        self.y += delta * self.vel_y

    def reset(self):
        # We'll start by picking a random angle for the ball to move
        # We repick the direction if it isn't headed for the left
        # or the right hand side
        theta = random.random()*2*math.pi
        while ((theta > math.pi/4 and theta < 3*math.pi/4) or
               (theta > 5*math.pi/4 and theta < 7*math.pi/4)):
            theta = random.random()*2*math.pi
        # In addition to an angle, we need a velocity. Let's have the
        # ball move at 300 pixels per second
        r = 300

        self.vel_x = r * math.cos(theta)
        self.vel_y = r * math.sin(theta)

        # We'll start the ball at the center. self.pos is actually the
        # same as accessing sprite.x and sprite.y individually
        self.pos = (WIDTH/2, HEIGHT/2)
```

Collision Detection

Next, we'd like to have our ball interact with the sides of the game board, and with the paddles. We'll do two different types of collision detection here just to showcase them. Which you use will depend largely on the game.

First, we'll have the ball bounce off the top and bottom of the screen. For this, we'll do simple checks on the y coordinate of the ball. You may remember that we used a center anchor on the ball, so the coordinates are relative to the center of the ball. To remedy this, we'll use the Sprite attribute `rect`, which gives us a rectangle that represents the drawn area of the sprite, and we can check it's top and bottom attributes. When we see that they have passed the ceiling or the floor, we'll flip the y component of the velocity.

game/pong.py

```
class Ball(spyral.Sprite):
    def __init__(self, scene):
        super(Ball, self).__init__(scene)

        self.image = spyral.Image(size=(20, 20))
        self.image.draw_circle((255, 255, 255), (10, 10), 10)
        self.anchor = 'center'

        spyral.event.register('director.update', self.update)
        self.reset()

    def update(self, delta):
        self.x += delta * self.vel_x
        self.y += delta * self.vel_y

        r = self.rect
        if r.top < 0:
            r.top = 0
            self.vel_y = -self.vel_y
        if r.bottom > HEIGHT:
            r.bottom = HEIGHT
            self.vel_y = -self.vel_y

    def reset(self):
        # We'll start by picking a random angle for the ball to move
```

Next, we'll have the ball collide with the two paddles. We will place the collision check at the Scene level, because it requires checking two Sprites. Every *director.update*, we'll check to see if the ball is colliding with either padel; if so, then we will call a method in the *Ball* class called *bounce* that flips the horizontal velocity of the ball. It will check for collisions using the `collide_sprites` method of scenes. Note that sprites also have a `collide_sprite` method.

game/pong.py

```
import spyral
import random
import math

WIDTH = 1200
HEIGHT = 900
SIZE = (WIDTH, HEIGHT)

class Ball(spyral.Sprite):
    def __init__(self, scene):
        super(Ball, self).__init__(scene)

        self.image = spyral.Image(size=(20, 20))
        self.image.draw_circle((255, 255, 255), (10, 10), 10)
        self.anchor = 'center'

        spyral.event.register('director.update', self.update)
        self.reset()

    def update(self, delta):
        self.x += delta * self.vel_x
        self.y += delta * self.vel_y

        r = self.rect
        if r.top < 0:
            r.top = 0
            self.vel_y = -self.vel_y
        if r.bottom > HEIGHT:
            r.bottom = HEIGHT
            self.vel_y = -self.vel_y

    def reset(self):
        # We'll start by picking a random angle for the ball to move
        # We repick the direction if it isn't headed for the left
        # or the right hand side
        theta = random.random()*2*math.pi
        while ((theta > math.pi/4 and theta < 3*math.pi/4) or
               (theta > 5*math.pi/4 and theta < 7*math.pi/4)):
            theta = random.random()*2*math.pi
        # In addition to an angle, we need a velocity. Let's have the
        # ball move at 300 pixels per second
        r = 300

        self.vel_x = r * math.cos(theta)
        self.vel_y = r * math.sin(theta)

        # We'll start the ball at the center. self.pos is actually the
        # same as accessing sprite.x and sprite.y individually
        self.pos = (WIDTH/2, HEIGHT/2)

    def bounce(self):
        self.vel_x = -self.vel_x

class Paddle(spyral.Sprite):
    def __init__(self, scene, side):
        super(Paddle, self).__init__(scene)

        self.image = spyral.Image(size=(20, 300)).fill((255, 255, 255))
        self.anchor = 'mid' + side
        if side == 'left':
            self.x = 20
        else:
```

User Input

User Input is handled the same way that *director.update* is - by registering a function with the event. To get started, we'll register another event on the scene: `system.quit`, which is fired when the user presses the exit button. Almost every game will want to respect this event.

game/pong.py

```
1  class Pong(spyral.Scene):
2      def __init__(self):
3          super(Pong, self).__init__(SIZE)
4
5          self.background = spyral.Image(size=SIZE).fill( (0, 0, 0) )
6
7          self.left_paddle = Paddle(self, 'left')
8          self.right_paddle = Paddle(self, 'right')
9          self.ball = Ball(self)
10
11         spyral.event.register("director.update", self.update)
12         spyral.event.register("system.quit", spyral.director.pop)
```

A much more interesting event is `input.keyboard.down.*`, which is fired whenever the keyboard is pressed. You can also register on specific keys, e.g., `input.keyboard.down.left` or `input.keyboard.keyboard.down.f`. A complete list of keys is available *Keyboard Keys*.

The left and right paddles need to move differently depending on which side they are on - the left paddle responds to *w* and *s*, and the right paddle responds to *up* and *down*. Also, we want the paddles to keep moving after the keys are released. We'll use a `moving` attribute to keep track of whether the paddle should move either `"up"` or `"down"`.

game/pong.py

```

        self.image = spyral.Image(size=(20, 300)).fill((255, 255, 255))

        self.anchor = 'mid' + side

        self.side = side
        self.moving = False
        self.reset()

        if self.side == 'left':
            spyral.event.register("input.keyboard.down.w", self.move_up)
            spyral.event.register("input.keyboard.down.s", self.move_down)
            spyral.event.register("input.keyboard.up.w", self.stop_move)
            spyral.event.register("input.keyboard.up.s", self.stop_move)
        else:
            spyral.event.register("input.keyboard.down.up", self.move_up)
            spyral.event.register("input.keyboard.down.down", self.move_down)
            spyral.event.register("input.keyboard.up.up", self.stop_move)
            spyral.event.register("input.keyboard.up.down", self.stop_move)
        spyral.event.register("director.update", self.update)

    def move_up(self):
        self.moving = 'up'

    def move_down(self):
        self.moving = 'down'

    def stop_move(self):
        self.moving = False

    def reset(self):
        if self.side == 'left':
            self.x = 20
        else:
            self.x = WIDTH - 20
        self.y = HEIGHT/2

    def update(self, dt):
        paddle_velocity = 250

        if self.moving == 'up':
            self.y -= paddle_velocity * dt

        elif self.moving == 'down':
            self.y += paddle_velocity * dt

        r = self.get_rect()
        if r.top < 0:
            r.top = 0
        if r.bottom > HEIGHT:
            r.bottom = HEIGHT
        self.pos = r.center


class Pong(spyral.Scene):
```

User Events

New events can be queued and registered in spyral as easily as system events. We'll `queue` a new event `pong.score` when the ball goes either on the left or right side of the screen. Notice that we pass in a `Event`, which we give a parameter named `scorer`. Functions registered to this event can take in a `scorer` parameter to find out who scored.

We also register the `reset` method with this `pong.score` event on the Paddles and Ball, so that they are reset when someone scores. Finally, we register an `increase_score` method on the Scene, so that we can keep track of the score of the game. Notice how we have created a new `model` dictionary outside of the Scene; this model can hold the global state, and be saved and loaded more easily if we someday wanted to enable saving.

game/pong.py

```
import spyral
import random
import math

WIDTH = 1200
HEIGHT = 900
SIZE = (WIDTH, HEIGHT)

model = {"left": 0, "right": 0}

class Ball(spyral.Sprite):
    def __init__(self, scene):
        super(Ball, self).__init__(scene)

        self.image = spyral.Image(size=(20, 20))
        self.image.draw_circle((255, 255, 255), (10, 10), 10)
        self.anchor = 'center'

        spyral.event.register('director.update', self.update)
        spyral.event.register('pong.score', self.reset)
        self.reset()

    def update(self, delta):
        self.x += delta * self.vel_x
        self.y += delta * self.vel_y

        r = self.rect
        if r.top < 0:
            r.top = 0
            self.vel_y = -self.vel_y
        if r.bottom > HEIGHT:
            r.bottom = HEIGHT
            self.vel_y = -self.vel_y
        if r.left < 0:
            spyral.event.queue("pong.score", spyral.Event(scorer="left"))
        if r.right > WIDTH:
            spyral.event.queue("pong.score", spyral.Event(scorer="right"))

    def reset(self):
        # We'll start by picking a random angle for the ball to move
        # We repick the direction if it isn't headed for the left
        # or the right hand side
        theta = random.random()*2*math.pi
        while ((theta > math.pi/4 and theta < 3*math.pi/4) or
               (theta > 5*math.pi/4 and theta < 7*math.pi/4)):
            theta = random.random()*2*math.pi
        # In addition to an angle, we need a velocity. Let's have the
        # ball move at 300 pixels per second
        r = 300

        self.vel_x = r * math.cos(theta)
        self.vel_y = r * math.sin(theta)

        # We'll start the ball at the center. self.pos is actually the
        # same as accessing sprite.x and sprite.y individually
        self.pos = (WIDTH/2, HEIGHT/2)

    def bounce(self):
        self.vel_x = -self.vel_x

class Paddle(spyral.Sprite):
    def __init__(self, scene, side):
        super(Paddle, self).__init__(scene)
```

2.2.3 Animations Tutorial

Animations are a useful feature for making Sprites move and change. They work by interpolating a property over time. When you interpolate, you mathematically calculate changes from an initial value to a final value. As an example, here is a simple game where a sprite moves horizontally across the room using an Animation over the `x` attribute for 1.5 seconds.

game/animating.py

```
import spyral
import sys
from spyral import Animation, easing

WIDTH = 1200
HEIGHT = 900
SIZE = (WIDTH, HEIGHT)

class Block(spyral.Sprite):
    def __init__(self, scene):
        spyral.Sprite.__init__(self, scene)
        self.image = spyral.Image(size=(64, 64)).fill((255, 0, 0))
        self.anchor = 'center'
        self.y = HEIGHT / 2

        animation = Animation('x', easing.Linear(0, WIDTH), duration = 1.5)
        self.animate(animation)

class Game(spyral.Scene):
    def __init__(self):
        spyral.Scene.__init__(self, SIZE)
        self.background = spyral.Image(size=SIZE).fill((0,0,0))

        spyral.event.register("system.quit", sys.exit)

        self.block = Block(self)
```

Note: If you have an animation running when the game starts, the first few frames might not be drawn as the program loads. That means your animation might already be in progress by the time you're able to see it. If you are bothered by this, have the animation triggered by a mouse or keyboard event.

We create an `Animation` object, and then we pass it into the `animate` method of a Sprite. We could very easily make the sprite move vertically simply by changing the attribute, which you'll notice is given as a `str`.

```
animation = Animation('y', easing.Linear(0, HEIGHT), duration = 1.5)
self.block.animate(animation)
```

A simplest animation requires:

- an attribute (e.g., `x`, `scale`, `image`, or even one you choose yourself)
- an easing (discussed next)
- and a duration (e.g., 1.5 seconds)

Easings

Remember back in Algebra, when you were given two points, and had to find a line that fit them? And then in Algebra 2, you were taught that you could fit curves to multiple points. This is similar to an `Easing` in Animations; a mathematical function over a given interval that Spyral will use in its calculations. Easings are actually a very common term: to get an idea of the variety of easings, check out this page of easings.

Spyral natively supports a number of easings. For instance, the `QuadraticIn` can be used to start slowly and then go faster.

```
animation = Animation('y', easing.QuadraticIn(0, HEIGHT), duration = 1.5)
```

The `QuadraticOut` starts fast and then slows down:

```
animation = Animation('y', easing.QuadraticIn(0, HEIGHT), duration = 1.5)
```

Not all of the easings have an explicit `start` and `end` though; consider the `Sine` easing, which takes in an amplitude instead. First the attribute will oscillate to the positive `amplitude`, and then to the negative `amplitude`. Notice that we also use a new parameter of the `Animation` named `shift`, that sets the initial value of the attribute.

```
animation = Animation('x', easing.Sine(WIDTH/4), duration = 1.5, shift=WIDTH/2)
```

Attributes

Animations can be used for more than just positions. For example, to stretch the Sprite horizontally:

```
animation = Animation('scale_x', easing.Linear(1.0, 2.0), duration = 1.5)
```

Of course, some attributes are not numbers, they are `Vec2Ds`: for instance, `pos`. Then you must use a Tuple easing Function.

```
animation = Animation('pos', easing.LinearTuple((0, 0) , (WIDTH, HEIGHT)), duration = 1.5)
```

And some attributes take on discrete values: `visible` takes on either `True` or `False`, and `image` could take on one of a list of images. For these animations, you can use the `Iterate` easing. This can be used to achieve blinking:

```
animation = Animation('visible', easing.Iterate([True, False]), duration = .5)
```

Or for running through a sequence of images:

```
filenames = ["walk0.png", "walk1.png", "walk2.png"]
images = [spyral.Image(filename=f) for f in filenames]
animation = Animation('image', easing.Iterate(images), duration = 1.5)
```

You can even iterate over your own custom variable. If you had a happiness level for your sprite, you might make it fluctuate between -10 and 10 by:

```
animation = Animation('happiness', easing.Sine(10), duration = 16)
```

Animation Events

Sometimes you need to perform an action when an animation is completed or has started. Fortunately, animations trigger their own *Animation Events*:

game/animating.py

```
class Game(spyral.Scene):
    def __init__(self):
        spyral.Scene.__init__(self, SIZE)
        self.background = spyral.Image(size=SIZE).fill((0,0,0))

        spyral.event.register("system.quit", sys.exit)
        spyral.event.register("Block.x.animation.start", self.hello)
        spyral.event.register("Block.x.animation.end", self.goodbye)

        self.block = Block(self)

    def hello(self, sprite):
        print "Hello", sprite

    def goodbye(self, sprite):
        print "Goodbye", sprite
```

Notice that the naming schema is:

- <the name of the Sprite's class>.
- <the name of the attribute>.
- animation.
- <either `start` or `end`>

A common pattern is to have a Finite-State Machine control the behavior of a Sprite in conjunction with animations. For instance, if you had a turret that charges up and then fires, you could control this behavior with an FSM.

game/animating.py

```
class Turret(spyral.Sprite):
    def __init__(self, scene):
        spyral.Sprite.__init__(self, scene)
        self.image = spyral.Image(size=(64, 64)).fill((255, 0, 0))
        self.anchor = 'center'
        self.pos = (WIDTH/2, HEIGHT/2)
        self.load_images()

        self.charging_ani = Animation('image', easing.Iterate(self.charging_images), 4)
        self.firing_ani = Animation('image', easing.Iterate(self.firing_images), 2)
        self.charge()
        spyral.register('Turret.image.animation.end', self.update_state)

    def update_state(self, sprite):
        if sprite == self:
            # If you have more states, using a dictionary would be more elegant
            # e.g., self.state_functions[self.state]()
            if self.state == 'charging':
                self.fire()
            elif self.state == 'firing':
                self.charge()

    def fire(self):
        self.state = 'firing'
        self.animate(self.firing_ani)

    def charge(self):
        self.state = 'charging'
        self.animate(self.charging_ani)

    def load_images(self):
        self.charging_images = [] #Images go here
        self.firing_images = [] #Images go here
```

Notice how we test the `sprite` parameter to make sure that the given sprite is `self` - all Turrets fire the `Turret.image.animation.end` event, so we need to handle each individual turret separately. Also notice how we use a `str` to identify the state - this is good, pythonic practice.

Combining Animations

You can combine two animations into a new one very easily. For instance, to make one animation run after another, + them together:

```
first_animation = Animation('x', easing.Linear(0, WIDTH), duration = 1.5)
second_animation = Animation('scale_x', easing.Linear(1.0, 2.0), duration = 1.5)
animation = first_animation + second_animation
```

To make them run at the same time, in parallel, use the `&`:

```
animation = first_animation & second_animation
```

A special kind of animation is the DelayAnimation, which let's you add delays.

```
half_second_delay = DelayAnimation(.5)
move_x = Animation('x', easing.Linear(0, WIDTH), duration = 1)
```

```
scale_x = Animation('scale_x', easing.Linear(1.0, 2.0), duration = 1.5)
animation = (half_second_delay + move_x) & scale_x
```

Looping and Stopping animations

Animations normally end after one iteration, but you can make them loop infinitely by setting an Animation's `loop` parameter to `True`. This is extremely useful for things like pointing arrows meant to grab users' attention.

```
animation = Animation('x', easing.Sine(WIDTH/4), duration = 1.5, shift=WIDTH/2, loop=True)
```

If you need to stop an animation, you can do it by passing in a specific animation to `stop_animation`:

```
def __init__(self, scene):
    ...
    self.moving_animation = Animation('x', easing.Linear(0, 600), duration = 3.0)
    self.animate(self.moving_animation)
    spyral.register.event("input.mouse.down", self.stop_moving)

def stop_moving(self):
    self.stop_animation(self.moving_animation)
```

Or you can stop all the animations with `stop_all_animations` :

```
spyral.register.event("input.mouse.down", self.block.stop_all_animations)
```

Follow the Cursor

Now we can combine what we know to make a cute game where the block chases the cursor.

game/animating.py

```
import spyral
import sys
from spyral import Animation, easing

WIDTH = 1200
HEIGHT = 900
SIZE = (WIDTH, HEIGHT)

class Block(spyral.Sprite):
    def __init__(self, scene):
        spyral.Sprite.__init__(self, scene)
        self.image = spyral.Image(size=(64, 64)).fill((255, 0, 0))
        self.anchor = 'center'
        self.pos = (WIDTH/2, HEIGHT/2)

class Game(spyral.Scene):
    def __init__(self):
        spyral.Scene.__init__(self, SIZE)
        self.background = spyral.Image(size=SIZE).fill((0,0,0))
        spyral.event.register("system.quit", sys.exit)
        self.block = Block(self)

        spyral.event.register("input.mouse.motion", self.follow)

    def follow(self, pos):
        self.block.stop_all_animations()
        animation = Animation('pos', easing.LinearTuple(self.block.pos, pos), duration = 1.0)
        self.block.animate(animation)
```

Custom Easings

You can create your own Easings; more examples are given in the source code for the Easing module.

```
def MyEasing(start=0.0, finish=1.0):
    """
    Linearly increasing: f(x) = x
    """
        def my_easing(sprite, delta):
            return (finish - start) * (delta) + start
    return my_easing
    animation = Animation('x', MyEasing(0, WIDTH), duration = 1.5)
```

If you end up creating any Easings of your own (e.g., QuadraticInTuple), please share them!

Conclusion

Animations cover a wide range of use cases, from movement to image changes, and beyond. But don't let the great power go to your head: some actions will always be slow on the XO laptops. For instance, animating over the `angle` attribute. Basically, you want to avoid dynamic drawing as much as possible. As you use more animations, test your creation on the XO laptop directly to see how it performs.

If you want to see all the easings and animations in action, there is an example in the Spyral github.

2.2.4 Becoming a Pythonista

This chapter is will provide a very brief review of some important python concepts which will be useful and necessary. The basic concepts will be for those who are unfamiliar with Python, but a good refresher for those who don't write in Python every day. Intermediate concepts will be great for programmers of all levels to refresh on some Python idioms. The code below will focus on Python 2.5+, since that is the version on most XO laptops.

Basic Concepts

PEP8 and The Zen of Python

One of the most important aspects of developing in Python is the Python community. Code is meant to be read, used, and worked on by many people. As building most games is going to end up being a group project, code style is of particular importance. PEP8 provides a style guide for Python code. It is a lengthy document, and not everything it has to say will be immediately applicable, but come back to it as you learn and grow as a developer. Some guidelines you may choose to ignore in your own code with no reprecussions, but some guidelines are absolutely essential. Some guidelines that are essential to follow for this course:

- Use 4 spaces per indentation level. (Good editors will allow you to set soft-tabs to four spaces. Figure this out before you continue. When working with a team, indentation style is non-negotiable.)
- Rules on blank lines
- Naming conventions are particularly important.

The Zen of Python is a list of guiding principles behind the design of Python, and a good list of guiding principles for any project written in python. The most common way to find it is a little easter egg. In a python interpreter

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Control Flow in Python

The `if` statement acts predictably in Python.

```
>>> if <conditional>:
...     statement
... elif <conditional>:
...     statement
... else:
...     statement
```

When it comes to loops, the most preferred idiom in Python is a `for` loop , used in the style that most other languages refer to as a "for each" loop.

```
>>> for <item> in <sequence>:
...     statement
```

Occasionally, you will use the other looping mechanism, `while` (but you probably shouldn't, second-guess any use of it).

```
>>> while <conditional>:
...     statement
```

Instead of a "do-until" loop like most languages have, a common idiom is

```
>>> for <item> in <sequence>:
...     statement
...     if <conditional>:
...         break

>>> while True:
...     statement
...     if <conditional>:
...         break
```

A final useful keyword is `pass`, which simply ends execution of the branch. This is often used to define stubs and "to-do" code.

```
>>> if <conditional>:
...      pass # TODO: make this "statement"
```

"Variables" in Python

Don't mistake Python for having variables, because that's not really true. Instead, there are "names" and "references. There is a good pictorial explanation of this concept here.

Numerics

There are two main kinds of numerical types in Python: `float` and `int`. Basically, `float` is used for decimal values and `int` is used for Integers. When possible, stick with `int`, because computers are not good at storing and comparing `float`. When performing operations between `float` and `int`, the result will be a `float`.

The operators + (addition), − (subtraction), and ⋆ (multiplication), all act predictably. Some other operations that are slightly more unusual are:

- `x / y` (division): quotient of `x` and `y`
- `x // y` (integer division): quotient of `x` and `y`, rounded down.
- `x % y` (remainder, or modulo): remainder of `x / y`
- `x ** y` (power): raises `x` to the power of `y`

- `abs(x)` (absolute value, or magnitude): forces `x` to be positive
- `int(x)` (convert to integer): converts `x` to integer
- `float(x)` (convert to float): converts `y` to float

Sequence Types

A sequence is a key concept in Python. There are many different kinds of sequences, but the basic idea is simply a bunch of data.

The list and the tuple are two of the most common sequence types. Lists are denoted by square brackets, while tuples are usually denoted by parenthesis, though they are not required. Both of them allow access by numeric keys, starting from 0.

```
>>> alist = [1,2,3]
>>> atuple = (1,2,3)
>>> atuple
(1, 2, 3)
>>> atuple = 1,2,3
>>> alist[1]
2
>>> atuple[1]
2
>>> alist[2]
3
>>> atuple[2]
3
```

The key difference between lists and tuples is that lists are mutable, and tuples are immutable.

```
>>> alist[2] = 4
>>> alist
[1, 2, 4]
>>> atuple[2] = 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> alist.append(1)
>>> atuple.append(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```

Lists also have a number of other useful methods. More on Lists.

Similar to a List is the `set`. A `set` is mutable, but has no specific ordering, and like the set in mathematics, contains only one copy of element. It's faster to test membership (`in`) with a set, so a `set` is a good choice if the order of the elements isn't important or you don't care about duplicates.

```
>>> prepositions = set(["to", "from", "on", "of"])
>>> 'dog' in prepositions
False
>>> prepositions.add('at')
>>> 'at' in prepositions
True
```

Strings

Strings in Python are actually just immutable sequences of characters. Python has a ton of built-in functions to work with strings. Remember, because Strings are immutable, you cannot modify them - instead, functions that work on strings return new strings.

You can concatenate (join) strings in python using the + operator. However, it is much preferred to **use interpolation** with `%` instead [1]. This method will allow you to provide named "arguments" to the string, which will be invaluable when it comes time to internationalize your game.

Compare the difference between concatenation:

```
>>> "Welcome, " + user + ", you are visitor #" + visitor + "."
"Welcome, Bob, you are visitor #3 to Platipy"
```

And interpolation:

```
>>> "Welcome, %(user)s, you are visitor #%(visitor)d to Platipy." %
...             {'user' : user, 'visitor' : visitor}
"Welcome, Bob, you are visitor #3 to Platipy"
```

You can use escape sequences inside of string literals. To prevent them from being escaped, you can prefix the string with an 'r' (great for dealing with regular expressions and windows file systems). You can also specify that the string should be unicode with a 'u' prefix.

```
>>> print "New\nLine"
New
Line
>>> print r"New\nLine"
New\nLine
>>> print u"Unicode"
Unicode
```

Sequence Unpacking

A useful Python feature is the ability to unpack a sequence, allowing for multiple assignment. You can unpack a tuple as follows:

```
>>> position = (5, 10)
>>> x, y = position
>>> x
5
>>> y
10
```

This also allows swapping without a temporary variable, due to the way evaluation and assignment works in Python.

```
>>> a,b = b,a
>>> a
2
>>> b
1
```

It is the comma that determines if an expression is a tuple, not parenthesis.

[1] In reality, `string.format` is more preferred for string interpolation, but is a feature not available in Python 2.5, so we suggest not getting accustomed to using it when targetting the OLPC XO.

```
>>> one_tuple = 5,
>>> not_tuple = (5)
>>> one_tuple
(5,)
>>> not_tuple
5
```

Tuple unpacking is wonderful, because it allows you to have elegant multiple returns from a function.

```
>>> x, y, width, height = image.get_dimensions()
```

Comprehensions

Comprehensions are a very powerful Python idiom that allows looping and filtering of data in a single expression. For a simple list comprehension, we can create a list of the squares of the integers from 0-9.

```
>>> squares = [x ** 2 for x in range(10)]
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

This is shorter than the equivalent loop

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x ** 2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

and also the preferred way of doing much of functional programming in Python. You may notice that this is the same as

```
>>> map(lambda x : x ** 2, range(10))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

In addition to mapping over sequences, comprehensions also support filtering

```
>>> odd_squares = [x ** 2 for x in range(10) if x % 2 == 1]
>>> odd_squares
[1, 9, 25, 49, 81]
```

Comprehensions also support iteration over multiple sequences simultaneously.

```
>>> [(x,y) for x in range(3) for y in range(4)]
[(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 2), (1, 3), (2, 0), (2, 1), (2, 2), (2, 3)]
```

The rule of thumb is that evaluation happens right to left in the for sequences, as the last for sequence would be like the innermost for loop.

Generator expressions are also a form of comprehension that does not have the same speed and memory overhead as list comprehensions up front. You'll see more about them in *Generators and Iterators*. If you're using Python 2.7, you also have access to dict and set comprehensions, which we won't talk about here.

Dictionaries

A dictionary, or a dict, is the standard mapping type in Python. Dicts can be created a few ways:

```
>>> {'key1' : 'value1', 'key2' : 'value2'}
{'key2': 'value2', 'key1': 'value1'}
>>> dict([('key1', 'value1'), ('key2', 'value2')])
{'key2': 'value2', 'key1': 'value1'}
>>> dict(key1 = 'value1', key2 = 'value2')
{'key2': 'value2', 'key1': 'value1'}
```

The keys in a dictionary can be any hashable object.

```
>>> a = { (0,1) : 1, 'a' : 4, 5 : 'test', (0, 'test') : 7 }
>>> a
{(0, 1): 1, 'a': 4, (0, 'test'): 7, 5: 'test'}
```

Note: While it is possible to include different data types in lists and dicts due to Python's loose-typing, it is almost always a bad practice and should be used with extreme care.

To retrieve values from a dictionary, you access them in the same way as lists and tuples.

```
>>> a[(0,1)]
1
>>> a[5]
'test'
```

You can also test if a key is in a dictionary using the *in* keyword:

```
>>> 'a' in a
True
>>> 4 in a
False
```

You can also add new members to the dictionary:

```
>>> a[7] = 12
>>> a
{(0, 1): 1, 'a': 4, (0, 'test'): 7, 5: 'test', 7: 12}
```

Dictionaries, like lists, provide many more useful features. See the Python tutorial's section on dicts.

Iterating Over Sequences

Back in `Control Flow`, we mentioned the `for` loop, and how it was used to iterate over sequences. It's very convenient!

```
>>> for a_dog_breed in ['Labrador', 'Corgi', 'Golden Retriever']:
...     print a_dog_breed
'Labrador'
'Corgi'
'Golden Retriever'
```

A very common use case is for iterating over a list of numbers. One way is to use `range` and it's generator equivalent `xrange` (we'll talk about how they are different in generators; for now, just use `xrange`).

```
>>> for x in xrange(3):
...     print x
0
1
2
```

The best way to iterate over a list and keep track of indices is to use the enumerate function.

```
>>> for index, name in enumerate(seasons)
...     print index, name
0 spring
1 summer
2 fall
3 winter
```

You can even iterate over dictionaries if you use the `items` function. >>> for key, value in {1: 'a', 2: 'b', 3: 'c'}.items() ... print key, value 1 a 2 b 3 c

Truth-Testing

There is no boolean type in Python. Anything can be evaluated for Truthiness in a conditional, however. Some things are always true, and some things are always false. You can test truthiness with the `bool` function.

```
>>> bool(True)           # True are special keywords
True
>>> bool(5)              # non-zero numbers are true
True
>>> bool(-5)             # only zero is false!
True
>>> bool([1,2,3])        # A non-empty sequence is true
True
>>> bool("Hello World")  # A non-empty string is true
True
>>> bool(bool)           # functions are first-order things!
True
```

Often, if you can think of it as "nothing", then it will evaluate to False.

```
>>> bool(False)  # False is a special keyword
False
>>> bool(0)      # zero is false
False
>>> bool([])     # empty list is false
False
>>> bool("")     # empty strings are false!
False
>>> bool(None)   # The special keyword None is false
False
```

There are quite a few built-in operators to test conditions. There are the usual suspects defined for most types (including non-numerics!): <, <=, >, >=, ==, and `!=`.

An unusual operator is `is`, which tests reference equality, meaning that both operands are identical objects (refer to the exact same thing). == is a value equality comparison (whether the two objects compute to the same thing). You will only use `is` for testing against `None` and testing object identity. Otherwise, use ==. Otherwise, you will find yourself in strange situations:

```
>>> 10 == 10
True
>>> 10 is 10 # accidentally works because of an internal python detail
True
>>> 1000 == 10**3
True
>>> 1000 is 10**3 # behaves unexpectedly!
False
```

Additionally, Python does contain boolean operators, but they are not `&&`, `||`, and `!` like many other languages, they are `and`, `or`, and `not`. They are short-circuit operators like most other languages.

Finally, you can use `in` to test membership.

```
>>> 5 in [1,2,3,4]
False
>>> 3 in [1,2,3,4]
True
```

Typing in Python

There are many types in Python, and you can always find out an expression's type by using the `type(x)` function.

```
>>> type(5)
<type 'int'>
>>> type(5.0)
<type 'float'>
>>> type("Hello World")
<type 'str'>
>>> type(u"Hello Unicode World")
<type 'unicode'>
>>> type([1,2,3])
<type 'list'>
>>> type(None)
<type 'NoneType'>
>>> type(type(None))
<type 'type'>
```

For more information on built-in types and truth value testing, see the Python tutorial's section on Built-in Types.

Functions

Defining a function is simple in python.

```
>>> def my_function(argument1, argument2):
...     statement
```

You usually want to return something.

```
>>> def mean(first, second):
...     return (first + second) / 2
```

You can also have default arguments for your parameters.

```
>>> def mean(first= 0, second= 9):
...     return (first + second) / 2
>>> mean()
5
```

Be wary, however, of mutable default arguments. You should almost always use None instead of mutable types, and check against None to set the actual default argument. >>> def foo(l=[]): ... l.append(1) ... return l ... >>> foo() [1] >>> foo() [1, 1]

And you can even have arbitrary arguments.

```
>>> def mean(*numbers): #numbers will be a tuple!
...   return sum(numbers) / len(numbers)
>>> mean(1, 8, 10, 15)
8
```

You can use named parameters when calling a function.

```
>>> mean(first= 10, second= 14)
12
```

And you can also accept arbitrary named parameters.

```
>>> def foo(*args, **kwargs):
...     print args
...     print kwargs
...
>>> foo(1,2,3, a=4, b=5)
(1, 2, 3)
{'a': 4, 'b': 5}
```

Python treats functions as first-class objects, which means you can pass them around like anything else:

```
>>> average = mean
>>> average
<function mean at 0x000000000>
>>> mean(5,9)
7
>>> average(5,9)
7
>>> bool(mean)
True
```

Closures

Functions in Python have access to names which are in their calling scope.

```
>>> def make_incrementor(start = 0):
...     def inc(amount):
...         return start + amount
...     return inc
...
>>> i = make_incrementor()
>>> i(5)
5
>>> i2 = make_incrementor(5)
>>> i2(5)
10
```

Exceptions

Python's exceptions are the same as most other languages

```
>>> try:
...    dangerous_statement
... except NameError, e:    # accept a specific type of exception
...    print e
... except Exception, e:    # accept all exceptions. You should almost never do this
```

```
...     print "Oh no!"
... finally:                # cleanup code that should run regardless of exception, even when there
...     print 'Always run this bit'
```

Don't use the `as` keyword, it was introduced in Python 3.

Generators and Iterators

Iterators are objects which define how iterating, or looping, over a sequence goes, but can also be used for general iteration purposes. To get an iterator of an object, you call *iter(obj)*. The returned object will have a *next()* method which will return the next item in the sequence or iterator. When there are no more items to iterate over, it will throw a *StopIteration* exception.

```
>>> l = [1,2]
>>> alist = [1,2]
>>> i = iter(alist)
>>> i.next()
1
>>> i.next()
2
>>> i.next()
```

Generator is the name of the pattern used to create iterators, but also refers to two convenient ways to create iterators. First, as an example of an iterator, let's write a simplified version of the *xrange* generator that takes only one argument and always starts from 0.

```
>>> class xrange(object):
...     def __init__(self, n):
...         self.n = n
...         self.cur = 0
...
...     def __iter__(self):
...         return self
...
...     def next(self):
...         if self.cur < self.n:
...             ret = self.cur
...             self.cur += 1
...             return ret
...         else:
...             raise StopIteration()
...
>>> xrange(5)
<__main__.xrange object at 0x10b130cd0>
>>> list(xrange(5))
[0, 1, 2, 3, 4]
```

We see immediately that this is a bit cumbersome and has a lot of boilerplate. Generator functions are a much simpler way to write this generator. In a generator function, the *yield* keyword returns a value, an the Python interpreter remembers where evaluation stopped when yield was called. On subsequent calls to the function, control returns to where *yield* was called. *xrange* now looks like the following.

```
>>> def xrange(n):
...     cur = 0
...     while cur < n:
...         yield cur
...         cur += 1
```

```
...
>>> list(xrange(5))
[0, 1, 2, 3, 4]
```

You can even call yield in more than one place in the code, if you wish. This simplifies the creation of generators quite a bit.

Generator expressions are also commonplace. They use the same syntax as list comprehensions, but use *()* in place of *[]*. This allows for memory efficient use of generators and iterators for manipulating data.

```
>>> gen = (x ** 2 for x in range(6))
>>> gen
<generator object <genexpr> at 0x10b11deb0>
>>> list(gen)
[0, 1, 4, 9, 16, 25]
```

For more advanced tricks with generators and iterators, see the *itertools* module.

Object Oriented Programming

Python has classes!

```
>>> class <name>(object):
...   <body>
```

After you have a class, you can make instances of it:

```
>>> class Dog(object):
...     pass
>>> my_dog = Dog()
```

Classes usually have methods. Methods are functions which always take an instance of the class as the first argument. By convention, this is always named self. Accessing methods or member variables is done by using `self.<name>`

```
>>> class Dog(object):
...     def sniff(self):
...         print "Smells funny"
>>> Spot = Dog()
```

The constructor for a class is named `__init__`.

```
>>> class Dog(object):
...     def __init__(self):
...         self.breed = "Labrador"
...     def paint_it_black(self):
...         self.breed = "Black Lab"
```

Don't try and put properties outside of the `__init__` or other function, unless you want them to be *class* properties instead of *instance* attributes. Read about the distinction here

```
>>> class Animal(object):
...     def breathe(self):
...         print "*Gasp*"
>>> class Dog(Animal):
...     pass
>>> my_dog = Dog()
>>> my_dog.breathe()
*Gasp*
```

There are lots of other details about Classes that you should read up about on the Python Docs.

If __name__ == "__main__":

If you want to see if a script is being called as main, you can use the foloowing at the bottom of your file:

```
>>> if __name__ == "__main__":
...     pass # main stuff
```

In this class, we'll be using the launcher. So don't bother using this!

Assertions

Python has assertions, which are useful for verifying argument types, data structure invariants, and generally making assumptions explicit in your programs. The syntax is straightforward.

```
>>> assert 1 == True
>>> assert 0 == True
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

The Python Wiki has a good article on using assertions effectively

Built-in Documentation and Docstrings

In the interpreter, it is often useful to quickly check and see some documentation on objects you're working with. The built-in help function can quickly provide some information and a list of methods on both Python's built-in classes, and user-defined classes which are documented properly.

```
>>> a = [1,2,3]
>>> help(a)
Help on list object:
class list(object)
 |  list() -> new empty list
 |  list(iterable) -> new list initialized from iterable's items
 |
 |  Methods defined here:
 |
 |  __add__(...)
 |      x.__add__(y) <==> x+y
```

> For your own classes and functions, you should provide docstrings so that this functionality works, and also so that anyone reading your code has this information available. If a class, function, or method definition has a string before any other code, that string is interpreted as the docstring, and stored in `.__doc__` for that object. By convention, docstrings are written as triple-quoted strings (`"""string"""`)
>
> Help on function bake_bread in module __main__:

```
>>> def bake_bread(self, ingredients):
...     """
...     This function bakes a loaf of bread given an iterable of ingredients.
...     """
...     pass
...
>>> bake_bread.__doc__
'\n    This function bakes a loaf of bread given an iterable of ingredients.\n    '
>>> help(bake_bread)
```

```
bake_bread(self, ingredients)
    This function bakes a loaf of bread given an iterable of ingredients.
```

Importing, Modules, and Packages

This article does a good job describing importing in Python.

Advanced Concepts

There are a large number of advanced concepts in python. Most of them will not be necessary to achieve success in this course, so feel free to skip the rest of this chapter.

New- and Old- Style Classes

For more information, consult this page .

Decorators

Decorators are a tricky but useful feature that require you to really know how functions exist in Python. For a quick introduction, consult this 12-step guide .

Sadly, Python 2.5 does not support decorators.

Metaclasses

Metaclasses are a complicated subject that get at the heart of how classes work in python. For a lengthy explanation, consult this page .

Context Managers

Context Managers are not natively built into Python 2.5, but can be enabled with the use of a future import. To read more about them, consult this page .

Descriptors

Spyral uses Descriptors extensively in order to make magic properties that behave more like functions. If you're curious how setting the `x` property of a `spyral.Sprite` affects its `pos` property, then read the following documentation on Descriptors .

Additional Reading

Important Modules

itertools

Itertools provides many useful functions for manipulating iterables (e.g., lists, sets). Often, if you find yourself writing a complicated list handling routine, there will be an existing solution in this module .

random

Most interesting games will require extensive knowledge of how randomness works in computers. To read more about how this is done in Python, refer to this page .

operator

In a functional programming course, you often pass common operators as arguments (e.g., `+` in `(foldr + 0 my-list)`). This can be done in Python by using the operator module. Every operation in Python maps to a function in this class.

logging

Python has some built-in utilities for logging , although eventually the Dev launcher will provide these mechanisms.

collections

Although the built-in Python datatypes (`set`, `list`, `tuple`, etc.) are very useful, sometimes you need something else. Read over the extra collections available, and pay particular attention to `defaultdict` and `Counter`.

os and sys

Python uses two modules for connecting to your OS and interpreter . You'll wish you had these modules when you take Operating Systems and have to write your own shell.

pdb

The Python Debugger is a useful tool for post-mortem analysis of why your program crashed. Eventually, this will be integrated into the Dev Launcher.

json

JSON is a brillantly simple format for exchanging data between applications, and it has functionally replaced XML for most of the web's communication. XML is almost never the answer when you need to transmit or store data; JSON almost always is. Although later version of python bundle a module for manipulating JSON, you will need SimpleJson (which uses the same interface) for the XO laptop.

JSON is a better alternative for saving and restoring state than Pickling, which can lead to security vulnerabilities. This tutorial on the Python JSON library covers some of the differences, and is a useful guide for someone starting with the library.

Additional Reading

http://www.doughellmann.com/PyMOTW/py-modindex.html

Third-Party Modules

Requests

Requests is often considered one of the most beautiful Python libraries, and makes accessing web-based resources trivial. If your game requires connection to the internet (which is not recommended given the spotty internet with XO laptops), this is a requirement.

BeautifulSoup

If you need to get data out of an HTML page, the BeautifulSoup library is your friend.

Additional Reading

- Hidden Features of Python on StackOverflow is a great QA that just details some of Python's great features. Many of them have been listed here, a few haven't.

CHAPTER 3

Spyral Documentation

3.1 Complete API Reference

The following is complete documentation on all public functions and classes within Spyral. If you need further information on how the internals of a class work, please consider checking the source code attached.

3.1.1 Actors

class spyral.**Actor**

Actors are a powerful mechanism for quickly adding multiprocessing behavior to your game through Greenlets . Any object that subclasses the Actor mixin can implement a *main* method that will run concurrently. You can put a non-terminating loop into it, and it will work like magic, allowing other actors and the main game itself to keep processing:

```
class MyActor(spyral.Actor):
    def main(self, delta):
        while True:
            print "Acting!"
```

When an instance of the above class is created in a scene, it will continuously print "Acting!" until the scene ends. Like a Sprite, An Actor belongs to the Scene that was currently active when it was created.

main(*delta*)

The main function is executed continuously until either the program ends or the main function ends. While the Actor's scene is not on the top of the stack, the Actor is paused; it will continue when the Scene is back on the top of the Directory's stack.

Parameters **delta** (*float*) – The amount of time that has passed since this method was last invoked.

run_animation(*animation*)

Run this animation, without blocking other Actors, until the animation completes.

wait(*delta=0*)

Switches execution from this Actor for *delta* frames to the other Actors. Returns the amount of time that this actor was left waiting.

Parameters **delta** (*number*) – the number of frames(?) to wait.

Return type float

3.1.2 Animations

class spyral.**Animation**(*property*, *easing*, *duration=1.0*, *absolute=True*, *shift=None*, *loop=False*)

Creates an animation on *property*, with the specified *easing*, to last *duration* in seconds.

The following example shows a Sprite with an animation that will linearly change its 'x' property from 0 to 100 over 2 seconds.:

```
from spyral import Sprite, Animation, easing
...
my_sprite = Sprite(my_scene)
my_animation = Animation('x', easing.Linear(0, 100), 2.0)
my_sprite.animate(my_animation)
```

Animations can be appended one after another with the + operator, and can be run in parallel with the *&* operator.

```
>>> from spyral import Animation, easing
>>> first  = Animation('x', easing.Linear(0, 100), 2.0)
>>> second = Animation('y', easing.Linear(0, 100), 2.0)
# Sequential animations
>>> right_angle = first + second
# Parallel animations
>>> diagonal = first & second
```

Parameters

- **property** (string) – The property of the sprite to change (e.g., 'x')
- **easing** (Easing) – The easing (rate of change) of the property.
- **duration** (float) – How many seconds to play the animation
- **absolute** (boolean) – (**Unimplemented?**) Whether to position this relative to the sprite's offset, or to absolutely position it on the screen.
- **shift** (None, a Vec2D, or a number) – How much to offset the animation (a number if the property is scalar, a Vec2D if the property is "pos", and None if there is no offset.
- **loop** (boolean) – Whether to loop indefinitely

evaluate(*sprite*, *progress*)

For a given *sprite*, complete *progress*'s worth of this animation. Basically, complete a step of the animation. Returns a dictionary representing the changed property and its new value, e.g.: `{"x": 100}`. Typically, you will use the sprite's animate function instead of calling this directly.

Parameters

- **sprite** (Sprite) – The Sprite that will be manipulated.
- **progress** (*float*) – The amount of progress to make on this animation.

Return type dict

3.1.3 Director

The director handles initializing and running your game. It is also responsible for keeping track of the game's scenes. If you are not using the Example.Activity, then you will need to call init at the very start of your game, before you try to run run your first scene.

```
spyral.director.init((640, 480))
# Scene and sprite creation
spyral.director.run(scene=MyScene())
```

Note that the director provides three ways to change the current scene:

- Pushing new Scenes on top of old ones.
- Popping the current Scene, revealing the one underneath (or ending the game if this is the only scene on the stack)
- Replacing the current Scene with a new one.

spyral.director.**get_scene**()
Returns the currently running scene; this will be the Scene on the top of the director's stack.

Return type Scene

Returns The currently running Scene, or *None*.

spyral.director.**get_tick**()
Returns the current tick number, where ticks happen on each update, not on each frame. A tick is a "tick of the clock", and will happen many (usually 30) times per second.

Return type int

Returns The current number of ticks since the start of the game.

spyral.director.**init**(*size=(0, 0)*, *max_ups=30*, *max_fps=30*, *fullscreen=False*, *caption='My Spyral Game'*)
Initializes the director. This should be called at the very beginning of your game.

Parameters

- **size** (Vec2D) – The resolution of the display window. (0,0) uses the screen resolution
- **max_fps** (int) – The number of times that the director.update event will occur per frame. This will remain the same, even if fps drops.
- **max_ups** (int) – The number of frames per second that should occur when your game is run.
- **fullscreen** (bool) – Whether your game should start in fullscreen mode.
- **caption** (str) – The caption that will be displayed in the window. Typically the name of your game.

spyral.director.**pop**()
Pop the top scene off the stack, returning control to the next scene on the stack. If the stack is empty, the program will quit. This does return control, so remember to return immediately after calling it.

spyral.director.**push**(*scene*)
Place *scene* on the top of the stack, and move control to it. This does return control, so remember to return immediately after calling it.

Parameters **scene** (Scene) – The new scene.

spyral.director.**quit**()
Cleanly quits out of spyral by emptying the stack.

spyral.director.**replace**(*scene*)
Replace the currently running scene on the stack with *scene*. Execution will continue after this is called, so make sure you return; otherwise you may find unexpected behavior:

```
    spyral.director.replace(Scene())
    print "This will be printed!"
    return
```

Parameters **scene** (Scene) – The new scene.

spyral.director.**run**(*sugar=False*, *profiling=False*, *scene=None*)
Begins running the game, starting with the scene on top of the stack. You can also pass in a *scene* to push that scene on top of the stack. This function will run until your game ends, at which point execution will end too.

Parameters

- **sugar** (*bool*) – Whether to run the game for Sugar. This is only to the special XO Launcher; it is safe to ignore.
- **profiling** (*bool*) – Whether to enable profiling mode, where this function will return on every scene change so that scenes can be profiled independently.
- **scene** (Scene) – The first scene.

3.1.4 Easings

This module provides a set of built-in easings which can be used by any game. Additionally, custom easings can be built. An easing should be a function (or callable) which takes in a sprite, and a time delta which is normalized to [0,1], and returns the state of easing at that time. See the source code of this module for some example implementations. Built-in easings are stateless, so the same animation can be used many times or on many different objects. Custom easings do not have to be stateless.

Visualizations of these easings are available at http://easings.net .

spyral.easing.**Arc**(*center=(0, 0)*, *radius=1*, *theta_start=0*, *theta_end=6.283185307179586*)
Increasing according to a circular curve for two properties.

spyral.easing.**CubicIn**(*start=0.0*, *finish=1.0*)
Cubically increasing, starts very slow : f(x) = x^3

spyral.easing.**CubicInOut**(*start=0.1*, *finish=1.0*)
Cubically increasing, starts and ends very slowly but very fast in the middle.

spyral.easing.**CubicOut**(*start=0.0*, *finish=1.0*)
Cubically increasing, starts very fast : f(x) = 1 + (x-1)^3

spyral.easing.**Iterate**(*items*, *times=1*)
Iterate over a list of items. This particular easing is very useful for creating image animations, e.g.:

```
walk_images = [spyral.Image('f1.png'), spyral.Image('f2.png'), spyral.Image('f3.png')]
walking_animation = Animation('image', easing.Iterate(walk_images), 2.0, loop=True)
my_sprite.animate(walking_animation)
```

Parameters

- **items** (*list*) – A list of items (e.g., a list of Images).
- **times** (*int*) – The number of times to iterate through the list.

spyral.easing.**Linear**(*start=0.0*, *finish=1.0*)
Linearly increasing: f(x) = x

spyral.easing.**LinearTuple**(*start=(0, 0)*, *finish=(0, 0)*)
Linearly increasing, but with two properites instead of one.

spyral.easing.**Polar**(*center=(0, 0)*, *radius=<function <lambda> at 0x7f58f8d9e578>*, *theta_start=0*, *theta_end=6.283185307179586*)
Similar to an Arc, except the radius should be a function of time.

spyral.easing.**QuadraticIn**(*start=0.0*, *finish=1.0*)
Quadratically increasing, starts slower : f(x) = x ^ 2

spyral.easing.**QuadraticInOut**(*start=0.0*, *finish=1.0*)
Quadratically increasing, starts and ends slowly but fast in the middle.

spyral.easing.**QuadraticOut**(*start=0.0*, *finish=1.0*)
Quadratically increasing, starts faster : f(x) = 2x - x^2

spyral.easing.**Sine**(*amplitude=1.0*, *phase=0*, *end_phase=6.283185307179586*)
Depending on the arguments, moves at a different pace according to the sine function.

3.1.5 Events

This module contains functions and classes for creating and issuing events. For a list of the events that are built into Spyral, check the *Event List*.

spyral.event.**keys**
A special attribute for accessing the constants associated with a given key. For instance, `spyral.keys.down` and `spyral.keys.f`. This is useful for testing for keyboard events. A complete list of all the key constants can be found in the *Keyboard Keys* appendix.

spyral.event.**mods**
A special attribute for accessing the constants associated with a given mod key. For instance, `spyral.mods.lshift` (left shift) and `spyral.mods.ralt` (Right alt). This is useful for testing for keyboard events. A complete list of all the key constants can be found in the *Keyboard Modifiers* appendix.

class spyral.event.**Event**(***kwargs*)
A class for building for attaching data to an event. Keyword arguments will be named attributes of the Event when it is passed into `queue`:

```
collision_event = Event(ball=ball, paddle=paddle)
spyral.event.queue("ball.collides.paddle", collision_event)
```

spyral.event.**clear_namespace**(*namespace*, *scene=None*)
Clears all handlers from namespaces that are at least as specific as the provided *namespace*.

Parameters

- **namespace** (*str*) – The complete namespace.
- **scene** (`Scene` or `None`) – The scene to clear the namespace of; if it is `None`, then it will be attached to the currently running scene.

spyral.event.**handle**(*event_name*, *event=None*, *scene=None*)
Instructs spyral to execute the handlers for this event right now. When you have a custom event, this is the function you call to have the event occur.

Parameters

- **event_name** (*str*) – The type of event (e.g., `"system.quit"`, `"input.mouse.up"`, or `"pong.score"`.

- **event** (Event) – An Event object that holds properties for the event.
- **scene** (Scene or None.) – The scene to queue this event on; if None is given, the currently executing scene will be used.

spyral.event.**queue**(*event_name*, *event=None*, *scene=None*)

Queues a new event in the system, meaning that it will be run at the next available opportunity.

Parameters

- **event_name** (*str*) – The type of event (e.g., "system.quit", "input.mouse.up", or "pong.score".
- **event** (Event) – An Event object that holds properties for the event.
- **scene** (Scene or *None*.) – The scene to queue this event on; if *None* is given, the currently executing scene will be used.

spyral.event.**register**(*event_namespace*, *handler*, *args=None*, *kwargs=None*, *priority=0*, *scene=None*)

Registers an event *handler* to a namespace. Whenever an event in that *event_namespace* is fired, the event *handler* will execute with that event.

Parameters

- **event_namespace** (*str*) – the namespace of the event, e.g. "input.mouse.left.click" or "pong.score".
- **handler** (*function*) – A function that will handle the event. The first argument to the function will be the event.
- **args** (*sequence*) – any additional arguments that need to be passed in to the handler.
- **kwargs** (*dict*) – any additional keyword arguments that need to be passed into the handler.
- **priority** (*int*) – the higher the *priority*, the sooner this handler will be called in reaction to the event, relative to the other event handlers registered.
- **scene** (Scene or None) – The scene to register this event on; if it is None, then it will be attached to the currently running scene.

spyral.event.**register_dynamic**(*event_namespace*, *handler_string*, *args=None*, *kwargs=None*, *priority=0*, *scene=None*)

Similar to spyral.event.register() function, except that instead of passing in a function, you pass in the name of a property of this scene that holds a function.

Example:

```
class MyScene(Scene):
    def __init__(self):
        ...
        self.register_dynamic("orc.dies", "future_function")
        ...
```

Parameters

- **event_namespace** (*str*) – The namespace of the event, e.g. "input.mouse.left.click" or "pong.score".
- **handler** (*str*) – The name of an attribute on this scene that will hold a function. The first argument to the function will be the event.
- **args** (*sequence*) – any additional arguments that need to be passed in to the handler.
- **kwargs** (*dict*) – any additional keyword arguments that need to be passed into the handler.

- **priority** (*int*) – the higher the *priority*, the sooner this handler will be called in reaction to the event, relative to the other event handlers registered.
- **scene** (`Scene` or `None`) – The scene to register this event on; if it is `None`, then it will be attached to the currently running scene.

`spyral.event.`**`register_multiple`**(*event_namespace*, *handlers*, *args=None*, *kwargs=None*, *priority=0*, *scene=None*)

Similar to `spyral.event.register()` function, except a sequence of *handlers* are be given instead of just one.

Parameters

- **event_namespace** (*string*) – the namespace of the event, e.g. `"input.mouse.left.click"` or `"pong.score"`.
- **handler** (*list of functions*) – A list of functions that will be run on this event.
- **args** (*sequence*) – any additional arguments that need to be passed in to the handler.
- **kwargs** (*dict*) – any additional keyword arguments that need to be passed into the handler.
- **priority** (*int*) – the higher the *priority*, the sooner this handler will be called in reaction to the event, relative to the other event handlers registered.
- **scene** (`Scene` or `None`) – The scene to register this event on; if it is `None`, then it will be attached to the currently running scene.

`spyral.event.`**`register_multiple_dynamic`**(*event_namespace*, *handler_strings*, *args=None*, *kwargs=None*, *priority=0*, *scene=None*)

Similar to `spyral.Scene.register()` function, except a sequence of strings representing handlers can be given instead of just one.

Parameters

- **event_namespace** (*string*) – the namespace of the event, e.g. `"input.mouse.left.click"` or `"pong.score"`.
- **handler** (*list of strings*) – A list of names of an attribute on this scene that will hold a function. The first argument to the function will be the event.
- **args** (*sequence*) – any additional arguments that need to be passed in to the handler.
- **kwargs** (*dict*) – any additional keyword arguments that need to be passed into the handler.
- **priority** (*int*) – the higher the *priority*, the sooner this handler will be called in reaction to the event, relative to the other event handlers registered.
- **scene** (`Scene` or `None`) – The scene to register this event on; if it is `None`, then it will be attached to the currently running scene.

`spyral.event.`**`unregister`**(*event_namespace*, *handler*, *scene=None*)

Unregisters a registered handler for that namespace. Dynamic handler strings are supported as well.

Parameters

- **event_namespace** (*str*) – An event namespace
- **handler** (*a function or string.*) – The handler to unregister.
- **scene** (`Scene` or `None`) – The scene to unregister the event; if it is `None`, then it will be attached to the currently running scene.

3.1.6 Event Handlers

Event Handlers are used to process events from the system and pass them into Spyral. In addition to the default EventHandler, there are other event handlers for recording and restoring events to a file; using these events, you could generate demos or functional tests of your game. EventHandlers are an advanced feature that can be set through a private attribute of scenes: `spyral.Scene._event_source`

Note: Eventually, these event handlers will be set through the dev launcher.

---

class `spyral.event.`**EventHandler**
Base event handler class.

get(*types*=[ ])
Gets events from the event handler. Types is an optional iterable which has types which you would like to get.

tick()
Should be called at the beginning of update cycle. For the event handler which is part of a scene, this function will be called automatically. For any additional event handlers, you must call this function manually.

class `spyral.event.`**LiveEventHandler**(*output_file=None*)
An event handler which pulls events from the operating system.

The optional output_file argument specifies the path to a file where the event handler will save a custom json file that can be used with the *ReplayEventHandler* to show replays of a game in action, or be used for other clever purposes.

Note: If you use the output_file parameter, this function will reseed the random number generator, save the seed used. It will then be restored by the ReplayEventHandler.

---

class `spyral.event.`**ReplayEventHandler**(*input_file*)
An event handler which replays the events from a custom json file saved by the *LiveEventHandler*.

pause()
Pauses the replay of the events, making tick() a noop until resume is called.

resume()
Resumes the replay of events.

3.1.7 Fonts

class `spyral.`**Font**(*font_path*, *size*, *default_color=(0, 0, 0)*)
Font objects are how you get text onto the screen. They are loaded from TrueType Font files (*.ttf); system fonts are not supported for asthetic reasons. If you need direction on what the different size-related properties of a Font object, check out the Font example.

Parameters

- **font_path** (*str*) – The location of the *.ttf file.
- **size** (*int*) – The size of the font; font sizes refer to the height of the font in pixels.
- **color** (*A three-tuple.*) – A three-tuple of RGB values ranging from 0-255. Defaults to black `(0, 0, 0)`.

ascent
The height in pixels from the font baseline to the top of the font. Read-only.

descent
The height in pixels from the font baseline to the bottom of the font. Read-only.

get_metrics(*text*)
Returns a list containing the font metrics for each character in the text. The metrics is a tuple containing the minimum x offset, maximum x offset, minimum y offset, maximum y offset, and the advance offset of the character. `[(minx, maxx, miny, maxy, advance), (minx, maxx, miny, maxy, advance), ...]`

Parameters **text** (*str*) – The text to gather metrics on.

Return type *list* of tuples.

get_size(*text*)
Returns the size needed to render the text without actually rendering the text. Useful for word-wrapping. Remember to keep in mind font kerning may be used.

Parameters **text** (*str*) – The text to get the size of.

Returns The size (width and height) of the text as it would be rendered.

Return type `Vec2D`

height
The average height in pixels for each glyph in the font. Read-only.

linesize
The height in pixels for a line of text rendered with the font. Read-only.

render(*text*, *color=None*, *underline=False*, *italic=False*, *bold=False*)
Renders the given *text*. Italicizing and bolding are artificially added, and may not look good for many fonts. It is preferable to load a bold or italic font where possible instead of using these options.

Parameters

- **text** (*str*) – The text to render. Some characters might not be able to be rendered (e.g., "\n").
- **color** (*A three-tuple.*) – A three-tuple of RGB values ranging from 0-255. Defaults to the default Font color.
- **underline** (*bool*) – Whether to underline this text. Note that the line will always be 1 pixel wide, no matter the font size.
- **italic** (*bool*) – Whether to artificially italicize this font by angling it.
- **bold** (*bool*) – Whether to artificially embolden this font by stretching it.

Return type `Image`

3.1.8 Forms

class `spyral.`**Form**(*scene*)
Bases: `spyral.view.View`

Forms are a subclass of `Views` that hold a set of *Widgets*. Forms will manage focus and event delegation between the widgets, ensuring that only one widget is active at a given time. Forms are defined using a special class-based syntax:

```
class MyForm(spyral.Form):
    name = spyral.widgets.TextInput(100, "Current Name")
    remember_me = spyral.widgets.Checkbox()
```

```
    save = spyral.widgets.ToggleButton("Save")

my_form = MyForm()
```

When referencing widgets in this way, the "Widget" part of the widget's name is dropped: `spyral.widgets.ButtonWidget` becomes `spyral.widgets.Button`. Every widget in a form is accessible as an attribute of the form:

```
>>> print my_form.remember_me.value
"up"
```

Parameters **scene** (`Scene` or `View`.) – The Scene or View that this Form belongs to.

add_widget(*name*, *widget*, *tab_order=None*)

Adds a new widget to this form. When this method is used to add a Widget to a Form, you create the Widget as you would create a normal Sprite. It is preferred to use the class-based method instead of this; consider carefully whether you can achieve dynamicity through visibility and disabling.

```
>>> my_widget = spyral.widgets.ButtonWidget(my_form, "save")
>>> my_form.add_widget("save", my_widget)
```

Parameters

- **name** (*str*) – A unique name for this widget.
- **widget** (*Widget*) – The new Widget.
- **tab_order** (*int*) – Sets the tab order for this widget explicitly. If tab-order is None, it is set to one higher than the highest tab order.

blur()

Defocuses the entire form.

focus(*widget=None*)

Sets the focus to be on a specific widget. Focus by default goes to the first widget added to the form.

Parameters **widget** (*Widget*) – The widget that is gaining focus; if None, then the first widget gains focus.

next(*wrap=True*)

Focuses on the next widget in tab order.

Parameters **wrap** (*bool*) – Whether to continue to the first widget when the end of the tab order is reached.

previous(*wrap=True*)

Focuses the previous widget in tab order.

Parameters **wrap** (*bool*) – Whether to continue to the last widget when the first of the tab order is reached.

values

A dictionary of the values for all the fields, mapping the name of each widget with the value associated with that widget. Read-only.

3.1.9 Images

A module for manipulating Images, which are specially wrapped Pygame surfaces.

class spyral.image.**Image**(*filename=None*, *size=None*)

The image is the basic drawable item in spyral. They can be created either by loading from common file formats, or by creating a new image and using some of the draw methods. Images are not drawn on their own, they are placed as the *image* attribute on Sprites to be drawn.

Almost all of the methods of an Image instance return the Image itself, enabling commands to be chained in a fluent interface.

Parameters

- **size** (Vec2D) – If size is passed, creates a new blank image of that size to draw on. If you do not specify a size, you *must* pass in a filename.
- **filename** (*str*) – If filename is set, the file with that name is loaded. The appendix has a list of the *valid image formats*. If you do not specify a filename, you *must* pass in a size.

copy()

Returns a copy of this image that can be changed while preserving the original.

Returns A new image.

crop(*position*, *size=None*)

Removes the edges of an image, keeping the internal rectangle specified by position and size.

Parameters

- **position** (a Vec2D or a Rect.) – The upperleft corner of the internal rectangle that will be preserved.
- **size** (Vec2D or None.) – The size of the internal rectangle to preserve. If a Rect was passed in for position, this should be None.

Returns This image.

draw_arc(*color*, *start_angle*, *end_angle*, *position*, *size=None*, *border_width=0*, *anchor='topleft'*)

Draws an elliptical arc on this image.

Parameters

- **color** (*a three-tuple of ints.*) – a three-tuple of RGB values ranging from 0-255. Example: (255, 128, 0) is orange.
- **start_angle** (*float*) – The starting angle, in radians, of the arc.
- **end_angle** (*float*) – The ending angle, in radians, of the arc.
- **position** (Vec2D or Rect) – The starting position of the ellipse (top-left corner). If position is a Rect, then size should be *None*.
- **size** (Vec2D) – The size of the ellipse; should not be given if position is a rect.
- **border_width** (*int*) – The width of the ellipse. If it is 0, the ellipse is filled with the color specified.
- **anchor** (*str*) – The anchor parameter is an *anchor position*.

Returns This image.

draw_circle(*color*, *position*, *radius*, *width=0*, *anchor='topleft'*)

Draws a circle on this image.

Parameters

- **color** (*a three-tuple of ints.*) – a three-tuple of RGB values ranging from 0-255. Example: (255, 128, 0) is orange.

- **position** (Vec2D) – The center of this circle
- **radius** (*int*) – The radius of this circle
- **width** (*int*) – The width of the circle. If it is 0, the circle is filled with the color specified.
- **anchor** (*str*) – The anchor parameter is an *anchor position*.

Returns This image.

draw_ellipse(*color*, *position*, *size=None*, *border_width=0*, *anchor='topleft'*)
Draws an ellipse on this image.

Parameters

- **color** (*a three-tuple of ints.*) – a three-tuple of RGB values ranging from 0-255. Example: (255, 128, 0) is orange.
- **position** (Vec2D or Rect) – The starting position of the ellipse (top-left corner). If position is a Rect, then size should be *None*.
- **size** (Vec2D) – The size of the ellipse; should not be given if position is a rect.
- **border_width** (*int*) – The width of the ellipse. If it is 0, the ellipse is filled with the color specified.
- **anchor** (*str*) – The anchor parameter is an *anchor position*.

Returns This image.

draw_image(*image*, *position=(0, 0)*, *anchor='topleft'*)
Draws another image over this one.

Parameters

- **image** (Image) – The image to overlay on top of this one.
- **position** (Vec2D) – The position of this image.
- **anchor** (*str*) – The anchor parameter is an *anchor position*.

Returns This image.

draw_lines(*color*, *points*, *width=1*, *closed=False*)
Draws a series of connected lines on a image, with the vertices specified by points. This does not draw any sort of end caps on lines.

Parameters

- **color** (*a three-tuple of ints.*) – a three-tuple of RGB values ranging from 0-255. Example: (255, 128, 0) is orange.
- **points** (A list of Vec2D s.) – A list of points that will be connected, one to another.
- **width** (*int*) – The width of the lines.
- **closed** (*bool*) – If closed is True, the first and last point will be connected. If closed is True and width is 0, the shape will be filled.

Returns This image.

draw_point(*color*, *position*, *anchor='topleft'*)
Draws a point on this image.

Parameters

- **color** (*a three-tuple of ints.*) – a three-tuple of RGB values ranging from 0-255. Example: (255, 128, 0) is orange.

- **position** (Vec2D) – The position of this point.
- **anchor** (*str*) – The anchor parameter is an *anchor position*.

Returns This image.

draw_rect(*color*, *position*, *size=None*, *border_width=0*, *anchor='topleft'*)
Draws a rectangle on this image.

Parameters

- **color** (*a three-tuple of ints.*) – a three-tuple of RGB values ranging from 0-255. Example: (255, 128, 0) is orange.
- **position** (Vec2D or Rect) – The starting position of the rect (top-left corner). If position is a Rect, then size should be *None*.
- **size** (Vec2D) – The size of the rectangle; should not be given if position is a rect.
- **border_width** (*int*) – The width of the border to draw. If it is 0, the rectangle is filled with the color specified.
- **anchor** (*str*) – The anchor parameter is an *anchor position*.

Returns This image.

fill(*color*)
Fills the entire image with the specified color.

Parameters **color** (*a three-tuple of ints.*) – a three-tuple of RGB values ranging from 0-255. Example: (255, 128, 0) is orange.

Returns This image.

flip(*flip_x=True*, *flip_y=True*)
Flips the image horizontally, vertically, or both.

Parameters

- **flip_x** (*bool*) – whether to flip horizontally.
- **flip_y** (*bool*) – whether to flip vertically.

Returns This image.

height
The height of this image in pixels (int). Read-only.

rotate(*angle*)
Rotates the image by angle degrees clockwise. This may change the image dimensions if the angle is not a multiple of 90.

Successive rotations degrate image quality. Save a copy of the original if you plan to do many rotations.

Parameters **angle** (*float*) – The number of degrees to rotate.

Returns This image.

scale(*size*)
Scales the image to the destination size.

Parameters **size** (Vec2D) – The new size of the image.

Returns This image.

size
The (width, height) of the image (Vec2D <spyral.Vec2D). Read-only.

width
The width of this image in pixels (int). Read-only.

spyral.image.**from_conglomerate**(*sequence*)
A function that generates a new image from a sequence of (image, position) pairs. These images will be placed onto a singe image large enough to hold all of them. More explicit and less convenient than `from_seqeuence`.

Parameters **sequence** (*List of image, position pairs.*) – A list of (image, position) pairs, where the positions are `Vec2D` s.

Returns A new `Image`

spyral.image.**from_sequence**(*images*, *orientation='right'*, *padding=0*)
A function that returns a new Image from a list of images by placing them next to each other.

Parameters

- **images** (List of `Image`) – A list of images to lay out.
- **orientation** (*str*) – Either 'left', 'right', 'above', 'below', or 'square' (square images will be placed in a grid shape, like a chess board).
- **padding** (*int or a list of ints.*) – The padding between each image. Can be specified as a scalar number (for constant padding between all images) or a list (for different paddings between each image).

Returns A new `Image`

spyral.image.**render_nine_slice**(*image*, *size*)
Creates a new image by dividing the given image into a 3x3 grid, and stretching the sides and center while leaving the corners the same size. This is ideal for buttons and other rectangular shapes.

Parameters

- **image** (`Image`) – The image to stretch.
- **size** (`Vec2D`) – The new (width, height) of this image.

Returns A new `Image` similar to the old one.

3.1.10 Keyboard

The keyboard modules provides an interface to adjust the keyboard's repeat rate.

spyral.keyboard.**repeat**
When the keyboard repeat is enabled, keys that are held down will keep generating new events over time. Defaults to *False*.

spyral.keyboard.**delay**
int to control how many milliseconds before the repeats start.

spyral.keyboard.**interval**
int to control how many milliseconds to wait between repeated events.

3.1.11 Mouse

The mouse modules provides an interface to adjust the mouse cursor.

spyral.mouse.**visible**
Bool that adjust whether the mouse cursor should be shown. This is useful if you want to, for example, use a Sprite instead of the regular mouse cursor.

spyral.mouse.**cursor**
str value that lets you choose from among the built-in options for cursors. The options are:

- `"arrow"` : the regular arrow-shaped cursor
- `"diamond"` : a diamond shaped cursor
- `"x"` : a broken X, useful for indicating disabled states.
- `"left"`: a triangle pointing to the left
- `"right"`: a triangle pointing to the right

Warning: Custom non-Sprite mouse cursors are currently not supported.

3.1.12 Rects

class spyral.**Rect**(**args*)
Rect represents a rectangle and provides some useful features. Rects can be specified 3 ways in the constructor:

1. Four numbers, `x`, `y`, `width`, `height`
2. Two tuples, `(x, y)` and *(width, height)*
3. Another rect, which is copied

```
>>> rect1 = spyral.Rect(10, 10, 64, 64)              # Method 1
>>> rect2 = spyral.Rect((10, 10), (64, 64))          # Method 2
>>> rect3 = spyral.Rect(rect1.topleft, rect1.size)   # Method 2
>>> rect4 = spyral.Rect(rect3)                       # Method 3
```

Rects support all the usual *anchor points* as attributes, so you can both get `rect.center` and assign to it. Rects also support attributes of `right`, `left`, `top`, `bottom`, `x`, and `y`.

```
>>> rect1.x
10
>>> rect1.centerx
42.0
>>> rect1.width
64
>>> rect1.topleft
Vec2D(10, 10)
>>> rect1.bottomright
Vec2D(74, 74)
>>> rect1.center
Vec2D(42.0, 42.0)
>>> rect1.size
Vec2D(64, 64)
```

clip(*other*)
Returns a Rect which is cropped to be completely inside of other. If the other does not overlap with this rect, a rect of size 0 is returned.

Parameters **other** (`Rect`) – The other Rect.

Returns A new `Rect`

clip_ip(*other*)
Modifies this rect to be cropped completely inside of other. If the other does not overlap with this rect, this rect will have a size of 0.

Parameters **other** (Rect) – The other Rect.

collide_point (*point*)

Parameters **point** (Vec2D) – The point.

Returns A *bool* indicating whether the point is contained within this rect.

collide_rect (*other*)
Returns *True* if this rect collides with the other rect.

Parameters **other** (Rect) – The other Rect.

Returns A *bool* indicating whether this rect is contained within another.

contains (*other*)
Returns *True* if the other rect is contained inside this rect.

Parameters **other** (Rect) – The other Rect.

Returns A *bool* indicating whether this rect is contained within another.

copy ()
Returns a copy of this rect

Returns A new Rect

inflate (*width*, *height*)
Returns a copy of this rect inflated by *width* and *height*.

Parameters

- **width** (*float*) – The amount to add horizontally.
- **height** (*float*) – The amount to add vertically.

Returns A new Rect

inflate_ip (*width*, *height*)
Inflates this rect by *width*, *height*.

Parameters

- **width** (*float*) – The amount to add horizontally.
- **height** (*float*) – The amount to add vertically.

move (*x*, *y*)
Returns a copy of this rect offset by *x* and *y*.

Parameters

- **x** (*float*) – The horizontal offset.
- **y** (*float*) – The vertical offset.

Returns A new Rect

move_ip (*x*, *y*)
Moves this rect by *x* and *y*.

Parameters

- **x** (*float*) – The horizontal offset.
- **y** (*float*) – The vertical offset.

union(*other*)
Returns a new rect which represents the union of this rect with other – in other words, a new rect is created that can fit both original rects.

Parameters **other** (Rect) – The other Rect.

Returns A new Rect

union_ip(*other*)
Modifies this rect to be the union of it and the other – in other words, this rect will expand to include the other rect.

Parameters **other** (Rect) – The other Rect.

3.1.13 Scenes

class spyral.**Scene**(*size=None*, *max_ups=None*, *max_fps=None*)
Creates a new Scene. When a scene is not active, no events will be processed for it. Scenes are the basic units that are executed by spyral for your game, and should be subclassed and filled in with code which is relevant to your game. The Director, is a manager for Scenes, which maintains a stacks and actually executes the code.

Parameters

- **size** (*width, height*) – The *size* of the scene internally (or "virtually"). This is the coordinate space that you place Sprites in, but it does not have to match up 1:1 to the window (which could be scaled).
- **max_ups** (*int*) – Maximum updates to process per second. By default, *max_ups* is pulled from the director.
- **max_fps** (*int*) – Maximum frames to draw per second. By default, *max_fps* is pulled from the director.

add_style_function(*name*, *function*)
Adds a new function that will then be available to be used in a stylesheet file.

Example:

```
import random
class MyScene(spyral.Scene):
    def __init__(self):
        ...
        self.load_style("my_style.spys")
        self.add_style_function("randint", random.randint)
        # inside of style file you can now use the randint function!
        ...
```

Parameters

- **name** (*string*) – The name the function will go by in the style file.
- **function** (*function*) – The actual function to add to the style file.

background
The background of this scene. The given Image must be the same size as the Scene. A background will be handled intelligently by Spyral; it knows to only redraw portions of it rather than the whole thing, unlike a Sprite.

collide_point(*sprite*, *point*)
Returns whether the sprite is colliding with the point.

Parameters

- **sprite** (`Sprite`) – A sprite
- **point** (`Vec2D`) – A point

Returns A `bool`

collide_rect(*sprite*, *rect*)
Returns whether the sprite is colliding with the rect.

Parameters

- **sprite** (`Sprite`) – A sprite
- **rect** (`Rect`) – A rect

Returns A `bool`

collide_sprites(*first*, *second*)
Returns whether the first sprite is colliding with the second.

Parameters

- **first** (`Sprite` or a `View`) – A sprite or view
- **second** (`Sprite` or a `View`) – Another sprite or view

Returns A `bool`

height
The height of this scene. Read-only number.

layers
A list of strings representing the layers that are available for this scene. The first layer is at the bottom, and the last is at the top.

Note that the layers can only be set once.

load_style(*path*)
Loads the style file in *path* and applies it to this Scene and any Sprites and Views that it contains. Most properties are stylable.

Parameters **path** (*str*) – The location of the style file to load. Should have the extension ".spys".

parent
Returns this scene. Read-only.

rect
Returns a `Rect` representing the position (0, 0) and size of this Scene.

redraw()
Force the entire visible window to be completely redrawn.

This is particularly useful for Sugar, which loves to put artifacts over our window.

scene
Returns this scene. Read-only.

size
Read-only property that returns a `Vec2D` of the width and height of the Scene's size. This is the coordinate space that you place Sprites in, but it does not have to match up 1:1 to the window (which could be scaled). This property can only be set once.

width
The width of this scene. Read-only number.

3.1.14 Sprites

class spyral.**Sprite**(*parent*)

Sprites are how images are positioned and drawn onto the screen. They aggregate together information such as where to be drawn, layering information, and more.

Parameters **parent** (View or Scene) – The parent that this Sprite will belong to.

anchor

Defines an *anchor point* where coordinates are relative to on the image. String.

angle

An angle to rotate the image by. Rotation is computed after scaling and flipping, and keeps the center of the original image aligned with the center of the rotated image.

animate(*animation*)

Animates this sprite given an animation. Read more about animation.

Parameters **animation** (Animation) – The animation to run.

collide_point(*point*)

Returns whether this sprite is currently colliding with the position. This uses the appropriate offsetting for the sprite within its views.

Parameters **point** (Vec2D) – The point (relative to the window dimensions).

Returns bool indicating whether this sprite is colliding with the position.

collide_rect(*rect*)

Returns whether this sprite is currently colliding with the rect. This uses the appropriate offsetting for the sprite within its views.

Parameters **rect** (Rect) – The rect (relative to the window dimensions).

Returns bool indicating whether this sprite is colliding with the rect.

collide_sprite(*other*)

Returns whether this sprite is currently colliding with the other sprite. This collision will be computed correctly regarding the sprites offsetting and scaling within their views.

Parameters **other** (Sprite) – The other sprite

Returns bool indicating whether this sprite is colliding with the other sprite.

flip_x

A boolean that determines whether the image should be flipped horizontally.

flip_y

A boolean that determines whether the image should be flipped vertically.

height

The height of the image after all transforms. Number.

image

The Image for this sprite.

kill()

When you no longer need a Sprite, you can call this method to have it removed from the Scene. This will not remove the sprite entirely from memory if there are other references to it; if you need to do that, remember to del the reference to it.

layer

String. The name of the layer this sprite belongs to. See *layering* for more.

mask
A `Rect` to use instead of the current image's rect for computing collisions. *None* if the image's rect should be used.

parent
The parent of this sprite, either a `View` or a `Scene`. Read-only.

pos
The position of a sprite in 2D coordinates, represented as a `Vec2D`

rect
Returns a `Rect` representing the position and size of this Sprite's image. Note that if you change a property of this rect that it will not actually update this sprite's properties:

```
>>> my_sprite.rect.top = 10
```

Does not adjust the y coordinate of *my_sprite*. Changing the rect will adjust the sprite however

```
>>> my_sprite.rect = spyral.Rect(10, 10, 64, 64)
```

scale
A scale factor for resizing the image. When read, it will always contain a `spyral.Vec2D` with an x factor and a y factor, but it can be set to a numeric value which wil ensure identical scaling along both axes.

scale_x
The x factor of the scaling that's kept in sync with scale. Number.

scale_y
The y factor of the scaling that's kept in sync with scale. Number.

scene
The top-level scene that this sprite belongs to. Read-only.

size
The size of the image after all transforms (`Vec2D`).

stop_all_animations()
Stops all animations currently running on this sprite.

stop_animation(*animation*)
Stops a given animation currently running on this sprite.

Parameters **animation** (`Animation`) – The animation to stop.

visible
A boolean indicating whether this sprite should be drawn.

width
The width of the image after all transforms. Number.

x
The x coordinate of the sprite, which will remain synced with the position. Number.

y
The y coordinate of the sprite, which will remain synced with the position. Number

3.1.15 Vec2Ds

class spyral.**Vec2D**(**args*)
Vec2D is a class that behaves like a 2-tuple, but with a number of convenient methods for vector calculation and

manipulation. It can be created with two arguments for x,y, or by passing a 2-tuple.

In addition to the methods documented below, Vec2D supports the following:

```
>>> from spyral import Vec2D
>>> v1 = Vec2D(1,0)
>>> v2 = Vec2D((0,1))     # Note 2-tuple argument!
```

Tuple access, or x,y attribute access

```
>>> v1.x
1
>>> v1.y
0
>>> v1[0]
1
>>> v1[1]
0
```

Addition, subtraction, and multiplication

```
>>> v1 + v2
Vec2D(1, 1)
>>> v1 - v2
Vec2D(1, -1)
>>> 3 * v1
Vec2D(3, 0)
>>> (3, 4) * (v1+v2)
Vec2D(3, 4)
```

Compatibility with standard tuples

```
>>> v1 + (1,1)
Vec2D(2, 1)
>>> (1,1) + v1
Vec2D(2, 1)
```

angle(*other*)
Returns the angle between this point and another point.

Parameters **other** (2-tuple or Vec2D) – the other point

Return type float

distance(*other*)
Returns the distance from this Vec2D to the other point.

Parameters **other** (2-tuple or Vec2D) – the other point

Return type float

dot(*other*)
Returns the dot product of this point with another.

Parameters **other** (2-tuple or Vec2D) – the other point

Return type int

floor()
Converts the components of this vector into ints, discarding anything past the decimal place.

Returns this Vec2D

static from_polar(**args*)
Takes in radius, theta or (radius, theta) and returns rectangular Vec2D.

Return type Vec2D

get_angle()
Return the angle this vector makes with the positive x axis.

Return type float

get_length()
Return the length of this vector.

Return type float

get_length_squared()
Return the squared length of this vector.

Return type int

normalized()
Returns a new vector based on this one, normalized to length 1. That is, it keeps the same angle, but its length is now 1.

Return type Vec2D

perpendicular()
Returns a new Vec2D perpendicular to this one.

Return type Vec2D

projection(*other*)
Returns the projection of this Vec2D onto another point.

Parameters **other** (2-tuple or Vec2D) – the other point

Return type float

rotated(*angle*, *center=(0, 0)*)
Returns a new vector from the old point rotated by *angle* radians about the optional *center*.

Parameters

- **angle** (*float*) – angle in radians.
- **center** (2-tuple or Vec2D) – an optional center

Return type Vec2D

to_polar()
Returns *Vec2D(radius, theta)* for this vector, where *radius* is the length and *theta* is the angle.

Return type Vec2D

3.1.16 Views

class spyral.**View**(*parent*)
Creates a new view with a scene or view as a parent. A view is a collection of Sprites and Views that can be collectively transformed - e.g., flipped, cropped, scaled, offset, hidden, etc. A View can also have a mask, in order to treat it as a single collidable object. Like a Sprite, a View cannot be moved between Scenes.

Parameters **parent** (View or Scene) – The view or scene that this View belongs in.

anchor
Defines an *anchor point* where coordinates are relative to on the view. String.

collide_point(*pos*)
Returns whether this view is colliding with the point.

Parameters **point** (Vec2D) – A point

Returns A bool

collide_rect(*rect*)
Returns whether this view is colliding with the rect.

Parameters **rect** (Rect) – A rect

Returns A bool

collide_sprite(*other*)
Returns whether this view is colliding with the sprite or view.

Parameters **other** (Sprite or a View) – A sprite or a view

Returns A bool

crop
A bool that determines whether the view should crop anything outside of it's size (default: True).

crop_height
The height of the cropped area. Number.

crop_size
The (width, height) of the area that will be cropped; anything outside of this region will be removed when the crop is active.

crop_width
The width of the cropped area. Number.

height
The height of the view. Number.

kill()
Completely remove any parent's links to this view. When you want to remove a View, you should call this function.

layer
The layer (a str) that this View is on, within its parent.

layers
A list of strings representing the layers that are available for this view. The first layer is at the bottom, and the last is at the top. For more information on layers, check out the *layers* appendix.

mask
Return this View's mask, a spyral.Rect representing the collidable area.

Return type Rect if this value has been set, otherwise it will be None.

output_height
The height of this view when drawn on the parent. Number.

output_size
The (width, height) of this view when drawn on the parent (Vec2D). Defaults to size of the parent.

output_width
The width of this view when drawn on the parent. Number.

parent
The first parent View or Scene that this View belongs to. Read-only.

pos
Returns the position (Vec2D) of this View within its parent.

rect
A Rect representing the position and size of this View. Can be set through a Rect, a 2-tuple of position and size, or a 4-tuple.

scale
A scale factor from the size to the output_size for the view. It will always contain a Vec2D with an x factor and a y factor, but it can be set to a numeric value which will be set for both coordinates.

scale_x
The x factor of the scaling. Kept in sync with scale. Number.

scale_y
The y factor of the scaling. Kept in sync with scale. Number.

scene
The top-most parent Scene that this View belongs to. Read-only.

size
The (width, height) of this view's coordinate space (Vec2D). Defaults to size of the parent.

visible
Whether or not this View and its children will be drawn (bool). Defaults to False.

width
The width of the view. Number.

x
The x coordinate of the view, which will remain synced with the position. Number.

y
The y coordinate of the view, which will remain synced with the position. Number.

3.1.17 Widgets

class spyral.widgets.**ButtonWidget** (*form*, *name*, *text='Okay'*)
A ButtonWidget is a simple button that can be pressed. It can have some text. If you don't specify an explicit width, then it will be sized according to it's text.

Parameters

- **form** (Form) – The parent form that this Widget belongs to.
- **name** (*str*) – The name of this widget.
- **text** (*str*) – The text that will be rendered on this button.

text
The text rendered on this button (str).

value
Whether or not this widget is currently "up" or "down".

class spyral.widgets.**CheckboxWidget** (*form*, *name*)
A CheckboxWidget is identical to a ToggleButtonWidget, only it doesn't have any text.

class spyral.widgets.**ToggleButtonWidget** (*form*, *name*, *text='Okay'*)
A ToggleButtonWidget is similar to a Button, except that it will stay down after it's been clicked, until it is clicked again.

Parameters

- **form** (Form) – The parent form that this Widget belongs to.
- **name** (*str*) – The name of this widget.
- **text** (*str*) – The text that will be rendered on this button.

class spyral.widgets.**TextInputWidget**(*form*, *name*, *width*, *value=''*, *default_value=True*, *text_length=None*, *validator=None*)

The TextInputWidget is used to get text data from the user, through an editable textbox.

Parameters

- **form** (Form) – The parent form that this Widget belongs to.
- **name** (*str*) – The name of this widget.
- **width** (*int*) – The rendered width in pixels of this widget.
- **value** (*str*) – The initial value of this widget.
- **default_value** (*bool*) – Whether to clear the text of this widget the first time it gains focus.
- **text_length** (*int*) – The maximum number of characters that can be entered into this box. If None, then there is no maximum.
- **validator** (*set*) – A set of characters that are allowed to be printed. Defaults to all regularly printable characters (which does not include tab and newlines).

anchor
Defines an *anchor point* <*anchors*> where coordinates are relative to on the view. String.

cursor_pos
The current index of the text cursor within this widget. A int.

nine_slice
The Image used to build the internal nine-slice image.

padding
A single int representing both the vertical and horizontal padding within this widget.

value
The current value of this widget, i.e, the text the user has input. When this value is changed, it triggers a form.<name>.<widget>.changed event. A str.

3.2 Spyral API Appendices

Appendices

3.2.1 Event List

There are many events that are built into Spyral. The following is a complete lists of them. You can `register` a handler for an event, and even `queue` your own custom events.

Director

`"director.update" : Event(delta)`
: **Parameters** **delta** (*float*) – The amount of time progressed since the last tick
Triggered by Every tick of the clock

`"director.pre_update" : Event()`
: **Triggered by** Every tick of the clock, before `"director.update"`

`"director.post_update" : Event()`
: **Triggered by** Every tick of the clock, after `"director.update"`

`"director.render" : Event()`
: **Triggered by** Every time the director draws the scene

`"director.pre_render" : Event()`
: **Triggered by** Directly before `"director.render"`

`"director.post_render" : Event()`
: **Triggered by** Directly after `"director.render"`

`"director.redraw" : Event()`
: **Triggered by** Every time the Director is forced to redraw the screen (e.g., if the window regains focus after being minimized).

`"director.scene.enter" : Event(scene)`
: **Parameters** **scene** (`Scene`) – The new scene
Triggered by Whenever a new scene is on top of the stack, e.g., a new scene is pushed, another scene is popped

`"director.scene.exit" : Event(scene)`
: **Parameters** **scene** (`Scene`) – The old scene
Triggered by Whenever a scene is slips off the stack, e.g., a new scene is pushed on top, a scene is popped

Animations

`"<sprite>.<attribute>.animation.start" : Event(animation, sprite)`
: **Parameters**
- **animation** (`Animation`) – The animation that is starting
- **sprite** (`Sprite`) – The sprite the animation is being played on

Triggered by A new animation starting on a sprite.

`"<sprite>.<attribute>.animation.end" : Event(animation, sprite)`
: **Parameters**
- **animation** (`Animation`) – The animation that is starting
- **sprite** (`Sprite`) – The sprite the animation is being played on

Triggered by An animation on a sprite ending.

User Input

`"input.mouse.down[.left | .right | .middle | .scroll_up | .scroll_down]" : Even (pos, butt`
: **Parameters**
- **pos** (*2-tuple*) – The location of the mouse cursor
- **button** (*str*) – Either `"left"`, `"right"`, `"middle"`, `"scroll_up"`, or `"scroll_down"`.

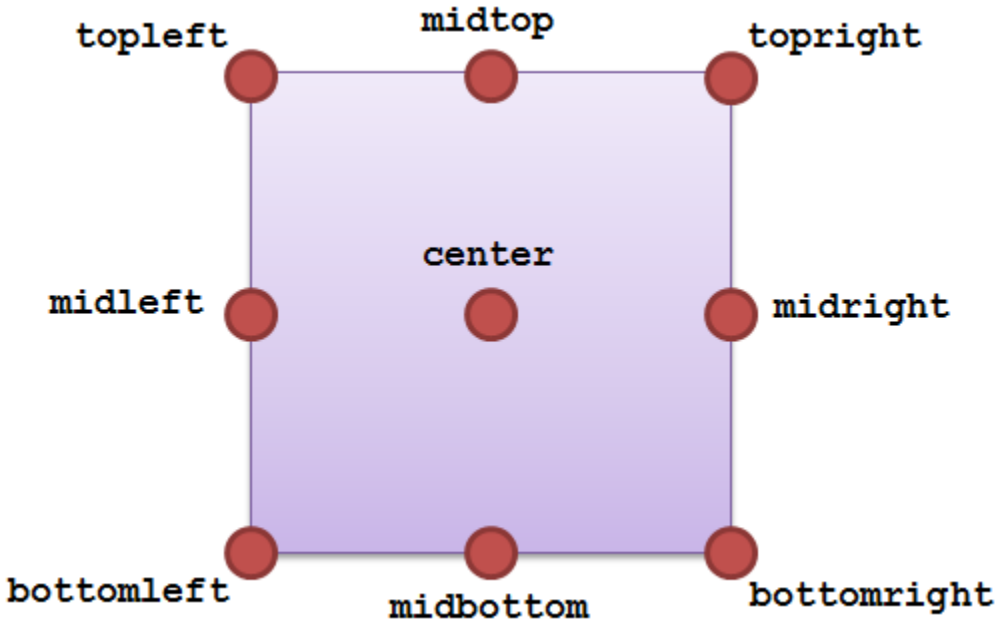
Triggered by Either any mouse button being pressed, or a specific mouse button being pressed

`"input.mouse.up[.left | .right | .middle | .scroll_up | .scroll_down]" : Event( os, buttor`
: **Parameters**
- **pos** (*2-tuple*) – The location of the mouse cursor
- **button** (*str*) – Either `"left"`, `"right"`, `"middle"`, `"scroll_up"`, or `"scroll_down"`.

Triggered by Either any mouse button being released, or a specific mouse button being re-

3.2.2 Anchors

There are several anchor points available, each given by a different string. The image below shows their locations on an image.



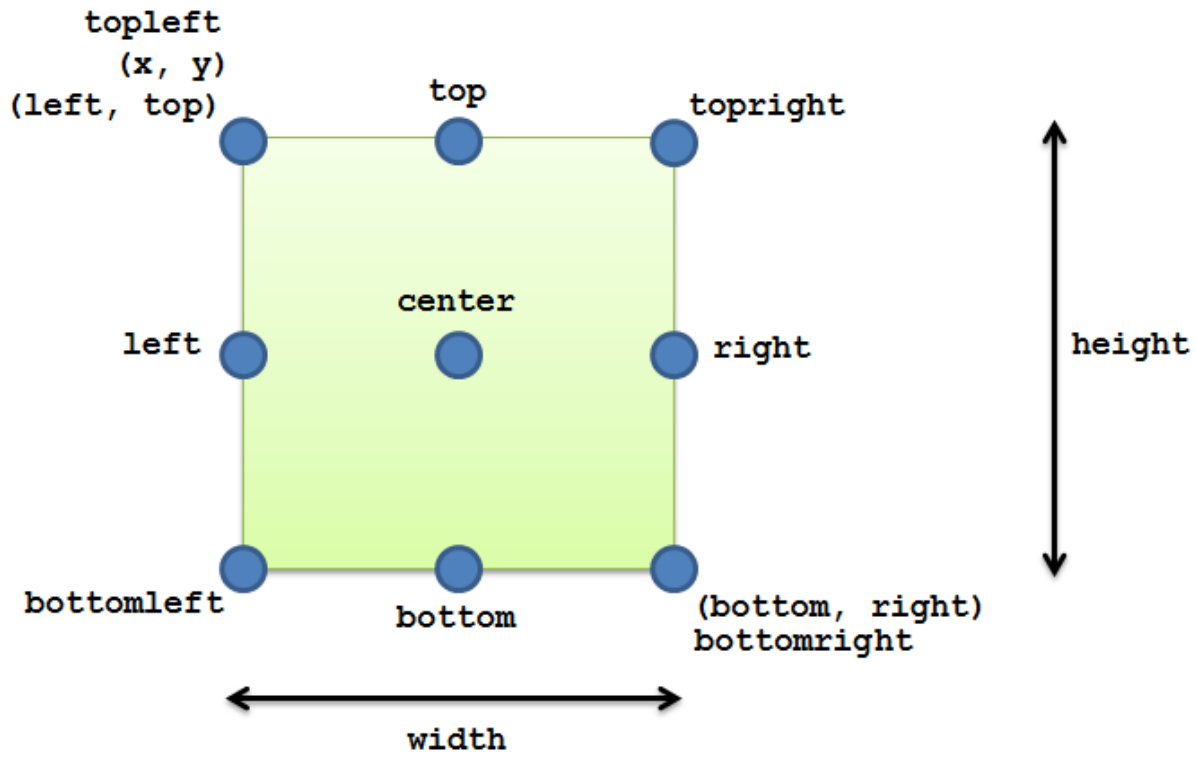
3.2.3 Keyboard Keys

3.2.4 Keyboard Modifiers

3.2.5 Layering

3.2.6 Rects

`Rects` can be accessed in a number of ways. The diagram below demonstrates the possibel properties.



3.2.7 Styleable properties

3.2.8 Valid Image Formats

- JPG
- PNG
- GIF (non animated)
- BMP
- PCX
- TGA (uncompressed)
- TIF
- LBM (and PBM)
- PBM (and PGM, PPM)
- XPM

3.3 Spyral API Cheat Sheets

The following are convenient listings of the attributes and methods of some of the more common classes and modules.

3.3.1 Sprites

Attribute	Type	Description
pos	Vec2D	Sprite's position within the parent
x	int	Sprite's horizontal position within the parent
y	int	Sprite's vertical position within the parent
anchor	str (*anchor point*)	Where to offset this sprite
image	Image	The image to display
visible	bool	Whether or not to render this sprite
layer	str (*layering*)	Which layer of the parent to place this sprite in
width	float	Width of the sprite after all transforms
height	float	Height of the sprite after all transforms
size	Vec2D	Size of the sprite after all transforms
rect	Rect	A rect describing size and position
scale	Vec2D (can be set to float)	Scale factor for resizing the image
scale_x	float	Scale factor for horizontally resizing the image
scale_y	float	Scale factor for vertically resizing the image
flip_x	bool	Whether the image should be flipped horizontally
flip_y	bool	Whether the image should be flipped vertically
angle	float	How much to rotate the image
mask	Rect or None	Alternate size of collision box
parent*	View or Scene	The immediate parent View or Scene
scene*	Scene	The top-most parent Scene

*Read-only

3.3.2 Scenes

Attribute	Type	Description
background	Image	The image to display as the static background
layers**	list of str (*layering*)	The layers for this scene
width*	int	Width of this scene internally (not the window).
height*	int	Height of this scene internally (not the window).
size*	Vec2D	Size of this scene internally (not the window).
rect	Rect	A rect stretching from (0, 0) to the size of the window.
parent*	Scene	This Scene
scene*	Scene	This Scene

*Read-only

** Can only be set once

3.3.3 Views

Attribute	Type	Description
pos	Vec2D	View's position within the parent
x	int	View's horizontal position within the parent
y	int	View's vertical position within the parent
width	float	Internal width of the view
height	float	Internal height of the view
size	Vec2D	Internal size of the view
rect	Rect	A rect describing size and position
anchor	str (*anchor point*)	Where to offset this view
visible	bool	Whether or not to render this view
layer	str (*layering*)	Which layer of the parent to place this view in
layers**	list of str (*layering*)	The layers for this view
scale	Vec2D (can be set to float)	Scale factor for resizing the view
scale_x	float	Scale factor for horizontally resizing the view
scale_y	float	Scale factor for vertically resizing the view
output_width	float	Width of the view after all transforms
output_height	float	Height of the view after all transforms
output_size	Vec2D	Size of the sprite after all transforms
crop	bool	Whether this View should be cropped
crop_width	float	Horizontal amount to keep uncropped
crop_height	float	Vertical amount to keep uncropped
crop_size	Vec2D	Size of the uncropped region within the View
mask	Rect or None	Alternate size of collision box
parent*	View or Scene	The immediate parent View or Scene
scene*	Scene	The top-most parent Scene

*Read-only

** Can only be set once

CHAPTER 4

Further References

4.1 Game Development Resources

4.1.1 General

Amit's Game Programming Information A mix of links to other content, and some great original content, make this a great resource when working on games. The sections on AI and Path Finding have been useful to many game developers.

4.1.2 Art and Design

2D Game Art For Programmers A great series on creating 2D artwork for games. If you enjoy the first post, be sure to read the blog, it contains many more articles than are on Gamasutra.

OpenGameArt A fantastic, on-going work to collect open source, public domain

4.2 Educational Game Design

I do a lot of research on Educational Game Design, so I'll be posting interesting papers along with summaries. Please read and comment!

4.2.1 Research

Digital Math Activity: Diagnostic Feedback

Title: "Embedding diagnostic mechanisms in a digital game for learning mathematics" (Huang, 2014)

Summary: This is a study where they made a multiple-choice activity with diagnostic feedback, and then did an assessment. They used the ARCS model of motivational design , which is pretty helpful from a design point of view:

- Gain the student's **attention**
- Establish **relevance**
- Increase student's **confidence**
- Give the student **satisfaction**

As I previously said, this is not a game: it's a multiple-choice activity. It isn't meant to *teach*, it's meant to be a low-stakes assessment with **diagnostic feedback**. They put some good work into identifying and diagnosing the mistakes that a student could make when doing addition. When they choose the wrong answer, they give feedback targeted to that student. Think of it like really good compiler errors - it can make or break your understanding of the error when you're just starting out!

Results

They assessed 56-2nd graders (45% female) (Southern Taiwan). One group had the diagnostic element, the other was basically just answering multiple choice questions. Their results for their measures:

- **Learning achievement questions from accredited textbooks** : students given the diagnostic feedback performed better than the students who didn't, both in comprehension and application type of questions.
- **Mathematical anxiety** : diagnostic feedback students were a little less anxious
- **Learning motivation based on ARCS** : Feedback was positive, except some learners felt that the material was too simple.
- **Qualitative interviews** : Results confirm ARCS data

My Thoughts

Weaknesses :

- Relatively small sample size.
- Although this activity is more fun than a quiz, it is still not as much fun as a regular game - multiple-choice is a pretty boring
- Only intended for assessment rather than actual learning.

The biggest take-away : Feedback is *key* in designing both software and instruction!

Formal Abstract

Mathematics is closely related to daily life, but it is also one of the lessons which often cause anxiety to primary school students. Digital game-based learning (DGBL) has been regarded as a sound learning strategy in raising learner willingness and interest in many disciplines. Thus, ways of designing a DGBL system to mitigate anxiety are well worth studying. This study adopts an Input–Process–Outcome DGBL model to develop a DGBL system

with a diagnostic mechanism strategy for a primary school mathematics course. In addition to exploring the impact of different learning methods on learning performance, this study further analyzes the learning methods in terms of learner anxiety about mathematics, learning motivation and learning satisfaction from the perspective of Attention, Relevance, Confidence-building, and Satisfaction (ARCS) motivation theory. The diagnostic mechanism strategy demonstrates the advantages of the DGBL system for mathematics learning. During the learning process, the ARCS questionnaire revealed that students who engage in learning through the DGBL method are positively motivated. The findings of this study suggest that centering on the daily life experiences of learners, integrating a proper game model into mathematics learning and providing a diagnostic mechanism prompt can effectively enhance interest in learning mathematics and reduce anxiety. When anxiety is mitigated, both learning motivation and learning performance are enhanced.

References

1. Huang, Y. M., Huang, S. H., & Wu, T. T. (2013). Embedding diagnostic mechanisms in a digital game for learning mathematics. *Educational Technology Research and Development*, 1-21.

Gender in Middle School Educational Games

Title : Gender differences in game activity preferences of middle school children: implications for educational game design (Kinzie, 2008)

Summary : This is one of my favorite papers. It describes a survey of 42 middle schoolers on their interest in game mechanics and elements. Demographic details:

- **Average age**: 12-years old
- **Grade**: 60% in sixth, 14% in seventh, and 21% in eighth
- **Ethncity**: 64% white, 12% asian american, 10% african american, 2% hispanic, 12% other
- Most regularly played video games

Results

- **Gender of their character** :
  - 7% students wanted a “genderless alien”
  - The rest wanted their own gender.
- **Age of their character**:
  - 64% wanted a slightly older character (13-20 years old)
  - 30% wanted a 20-30 year old
  - 8% wanted less than 13 years old
  - No one wanted older than 30 years
- **Build of male characters**:
  - 61% wanted a “muscular” build
  - 34% wanted a “fit” build
  - 5% wanted a “slight” build
- **Build of female characters**:

  - 61% wanted a "fit" build
  - 24% wanted a "slight" build
  - 14% wanted a "muscular" build
  - Girls were significantly more likely than boys to want a "slight" build
- **Ethnicity**:
  - 81% of White students wanted a white character
  - Insufficient data to draw conclusions from other ethnicity, but they seemed to have more variability
- **Choice of opponent**:
  - "Evil Overlord": 50% of the students overall (66% of the boys)
  - "Rival group of kids": 29% of the students overall (50% of the girls)
  - "Powerful government", "Neighborhood Bully": Appealed to no one
- **Who would you want to save?**:
  - "People your own age": 43% overall (46% of boys)
  - "Young children": 33% overall (56% of boys)
  - "Adults": 14% overall
  - "Senior Citizens": 10% overall
- **What do you want to save?**:
  - "All living things on the planet": 51% overall (63% of boys)
  - "Individual animals", "Individual people", "All the people in a city": otherwise evenly distributed
- **Where do you want the game to take place?**:
  - "Street Scene": 48%
  - "Sports playing field": 21%
  - "Shopping Mall": 17%
  - "Large meadow with pond": 14%
- **What do you do when you're stuck?**:
  - "Methodically try different ways to solve the problem": 50%
  - "Hints from a guide": 21%
  - "Discovering the answer through trial and error": 14%
  - "Being given the answer": 14%

Beyond these simple questions, they also asked students about *activity modes* (e.g., Explorative, Social, Creative, etc.,). They were looking at students general preferences between the modes and specific attitudes towards individual modes.

- To appeal to *both genders*: emphasize **Explorative** and **Problem-Solving** play
- To appeal more to *girls*: emphasize **Creative** play
- To appeal more to *boys*: emphasize **Active** and **Strategic** play

My Thoughts

Weaknesses :

- Relatively small population size
- Relied entirely on self-report
- This kind of survey is highly susceptible to cultural normalization

The biggest take-away : Boys and girls have some things in common, but they definitely don't look at the world the same. Ask your opposite-gendered friends if your game idea appeals to them. Of course, keep in mind that appealing to existing social norms might also reinforce negative ones; you might try pushing student's expectations a little.

Formal Abstract

Educators and learning theorists suggest that play is one of the most important venues for learning, and games a useful educational tool. This study considers game activity preferences of middle school-aged children, so that educational games might be made more appealing to them. Based on children's activity modes identified in our prior research, we developed the Educational Game Preferences Survey, which collects information on children's preferences for play activity modes, their attitudes about each activity mode, and their preferences for game characters, settings, and forms of help. Survey results suggest the appeal of the Explorative mode of play for all children, especially girls. Gender differences in children's preferences and attitudes for Active, Strategic, and Creative play modes were also found. We close with recommendations for game design to appeal to both boys and girls, as well as for boys and girls individually, to build engagement and hopefully lead to learning.

References

1. Kinzie, M. B., & Joseph, D. R. (2008). Gender differences in game activity preferences of middle school children: implications for educational game design. *Educational Technology Research and Development*, 56(5-6), 643-663.

Commercial Games in Schools

Title : Integrating Commercial Off-the-Shelf Video Games into School Curriculums (Charsky & Mims, 2008)

Summary : This one was just too much fun not to write about. They take off-the-shelf games (e.g., Civilization III, SimCity) and bring them into real classrooms. Note that this isn't really a research study (although off-the-shelf games have been studied in this context), it's just describing how to do this effectively.

Results

The big contribution that they make is three guidelines for introducing games. These guidelines act as a sequence.

1. **Learn the Game**: The reverse of gamification. For a given game, you analyze and report on the components. For instance, in a history game you create a timeline, or in a simulation game like Rollercoaster Tycoon, you make a budget. This gives students the chance to get used to the game, while still retaining the fun context.
2. **Cross-over**: Use the games as a comparison tool to teach misconceptions by explicitly pointing out where the game falls short. Their example is that in the game SimCity, your mayor avatar has absolute control to raise and lower taxes. Obviously, real mayors don't have this ability, but it's an opportunity to discuss the problems that mayors *do* have.

3. **Game as a Theory of the Content**: This goes beyond teaching misconceptions by asking students to propose changes that make the game more realistic.

My Thoughts

Weaknesses :

- Not a rigorous study, just a series of guidelines.
- They chose games that were easier to educationalize; how could other games be used for educational purposes, e.g., Final Fantasy or Call of Duty?

The biggest take-away : Just because a game wasn't built with education in mind, doesn't mean it won't have potential to teach!

References

1. Charsky, D., & Mims, C. (2008). Integrating commercial off-the-shelf video games into school curriculums. *TechTrends*, 52(5), 38-44.

Endogenous vs. Exogenous Games

Title : From Content to Context: Videogames as Designed Experience (Squire, 2006)

Summary : Kurt Squire is a big name in educational video games, and a paper by him is always worth reading. This paper is about games as a designed experience [1] , but I wanted to focus on his discussion of Endogenous vs. Exogenous games. If you have the time, though, you should skim the entire article!

Results

In Exogenous games, the learning context is external to the gameplay, as opposed to internalized like in Endogenous games. The article breaks this down further by contrasting the two terms:

- **Learner is...**
  - **Exogenous**: An empty receptacle. An example is Math Blaster, where the learner is "motivated" to learn a prescribed set of skills and facts.
  - **Endogenous**: An active, sense-making, social organism. An example is Grand Theft Auto, where the learner brings existing identities and experiences that color interpretations of the game experience.
- **Knowledge is...**
  - **Exogenous**: Knowledge of discrete facts. The facts are "true" by authority (generally the authority of the game designer).
  - **Endogenous**: Tool set used to solve problems. The right answer in Civilization is that which is efficacious for solving problems in the game world.
- **Learning is...**

[1] Designed Experiences are semi-controlled experiences for students that fits with the "learning by doing" metaphor - that is, people learn when they actively are doing something. The text gives a great example of this phenomenon when teaching students history via Civilization III. The students were playing as land-starved European countries that needed more resources in order to fight their wars; the natural solution was to develop colonies in the Africas and Americas and exploiting those resources. These gameplay decisions mimic the same route that history took, and launched some neat class discussions on the motivations of colonization. Designed Experiences are a very cool way to teach, so this article might get a follow-up blog post to really explore the concept!

  - **Exogenous**: Memorizing. Learners reproduce a set of prescribed facts, such as mathematics tables.
  - **Endogenous**: Doing, experimenting, discovering for the purposes of action in the world. Players learn in role-playing games for the purposes of acting within an identity.
- **Instruction is...**
  - **Exogenous**: Transmission. The goal of a drill and practice game is to transmit information effectively and to "train" a set of desired responses.
  - **Endogenous**: Making meaning/construction, discovery, social negotiation process. Instruction in Supercharged! involves creating a set of well designed experiences that elicit identities and encourage learners to confront existing beliefs, perform skills in context, and reflect on their understandings.
- **Social model is...**
  - **Exogenous**: "Claustrophobic." Players are expected to solve problems alone; using outside resources is generally "cheating."
  - **Endogenous**: Fundamentally group oriented. Games are designed to be played collectively, in affinity groups, and distributed across multiple media. They are designed with complexity to spawn affinity groups and communities that support game play.
- **Pre-knowledge is...**
  - **Exogenous**: Set of facts, knowledge, and skills to be assessed for proper pacing. In Math Blaster, players' self-efficacy in mathematics is not addressed.
  - **Endogenous**: Knowledge to be leveraged, played upon. Pre-knowledge is expected to color perception, ideas, and strategies. In Environmental Detectives, challenges are structured so that players become increasingly competent and learn to see the value of mathematics.
- **Identity is...**
  - **Exogenous**: Something to be cajoled. If players are not "motivated" to do math, the game developer's job is to create an "exciting" context for the learner.
  - **Endogenous**: Something to be recruited, managed, built over time. In Environmental Detectives, learners develop identities as scientists.
- **Context is...**
  - **Exogenous**: A motivational wrapper. The context in Math Blaster is something to make learning more palatable
  - **Endogenous**: The "content" of the experience. In Civilization, the geographical-materialist game model is the argument that situates activity and drives learning.

My Thoughts

The biggest take-away : If you're making a game, seriously ask yourself if it's Endo- or Exo-. Any programmer can make an Exogenous game; a real Educational Game Developer makes Endogenous games!

Formal Abstract

Interactive immersive entertainment, or videogame playing, has emerged as a major entertainment and educational medium. As research and development initiatives proliferate, educational researchers might benefit by developing more grounded theories about them. This article argues for framing game play as a designed experience. Players' understandings are developed through cycles of performance within the gameworlds, which instantiate particular theories of the world (ideological worlds). Players develop new identities both through game play and through the gaming

communities in which these identities are enacted. Thus research that examines game-based learning needs to account for both kinds of interactions within the game-world and in broader social contexts. Examples from curriculum developed for Civilization III and Supercharged! show how games can communicate powerful ideas and open new identity trajectories for learners.

References

1. Squire, K. (2006). From content to context: Videogames as designed experience. *Educational researcher*, 35(8), 19-29.

4.2.2 Queued Research Papers

1. Evidence-Centered Design
2. Acquisition vs. Participation Metaphors for Knowledge
3. Game Development Design Principles (Gelderblom's Masters thesis)
4. TPACK: Technological, Content, and Pedagogical Knowledge
5. What is Constructivism?
6. What is Situated Learning?
7. Motivation & Engagement: MUSIC model
8. Hanging Out, Messing Around, and Geeking Out
9. Next-generation Learning
10. Simulations vs. Games
11. Use - Modify - Create
12. Socio-Cognitive Learning
13. Informal vs. Formal Learning Environments
14. Connected Learning
15. How People Learn

CHAPTER 5

The Platipy Project

5.1 OLPC Contributor Application

5.1.1 Project Title & Shipment Detail

Name of Project :

Platipy (http://platipy.org)

Shipping Address You've Verified :

REDACTED

Number of Laptops (or other hardware) You Request to Borrow :

3 XO laptops

- 1 XO 1.0
- 1 XO 1.5
- 1 XO 1.75

Loan Length:

1 year

5.1.2 Team Participants

Name(s) & Contact Info:

- Austin Cory Bart (acbart@vt.edu, REDACTED)
- Robert Deaton (rdeaton@udel.edu, REDACTED)
- Eric McGinnis (ericmcg@udel.edu, REDACTED)

Employer and/or School:

- Austin Cory Bart: Graduate student at Virginia Tech
- Robert Deaton: Software Engineer at Quizlet.com
- Eric McGinnis: Graduate student at University of Delaware

Past Experience/Qualifications:

- **Austin Cory Bart (http://www.acbart.com)**:

  - Honors Bachelor of Science Degree in Computer Science with Distinction from the University of Delaware
  - PhD Student in Computer Science at Virginia Tech (*in-progress*)
  - Graduate Certificate in Learning Sciences at Virginia Tech (*in-progress*)
  - Two semesters of CISC-374 (Educational Game Design)
  - Author of Broadway and Wacky Writer .
  - Secondary developer of Spyral
- **Robert Deaton**:
  - Bachelor of Science Mathematics, Computer Science, University of Delaware
  - Teaching Assistant for CISC-374 (Educational Game Design)
  - Primary developer of Spyral
- **Eric McGinnis**:
  - Bachelor of Science Degree in Computer Science from the University of Delaware
  - Teaching Assistant for CISC-374 (Educational Game Design)
  - Experience teaching Scratch

5.1.3 Objectives

Project Objectives:

The overarching objective of the Platipy project is to provide tools and guides to develop educational games for the XO laptop. Our team wants to provide not just an educational experience for programmers, but also spur developers to create new, useful activities for children. Specifically, our goals for this project are to develop and/or create the following tools:

- **Spyral**: A pure-Pygame game development library. Offers a number of very powerful features, including automatic dirty rendering, actor-oriented programming (using greenlets), and a convenient API. Sits on top of Pygame completely, insulating the user from its intricacies.

  Status: Most of the interface and core functionality is established, although some smaller details need to be fleshed out, such as extending the Style system and more Widgets. Some additional work needs to be done to make it a full Pygame replacement, e.g. replacing the audio modules. Progress can be tracked on Spyral's Github.
- **Example.Activity**: A template activity that greatly simplifies XO activity development. Contains powerful features for developing Spyral-based activities off the XO, including handling translations, game scaling for non-XO resolutions, and packaging the game for the XO and other platforms such as Windows and Mac.

  Status: Most core functionality is complete, including profiling and scaling. However, non-XO packaging still needs work.
- **Platipy Docs**: A complete guide for Python, XO activity, and Spyral development. Will also include a curated gallery of completed Spyral-based XO activities.

  Status: The first version of the Python guide is roughly 90%, but needs editing. The XO Activity development is mostly finished. Spyral documentation is out of date. There is currently no gallery.
- **Conspyre**: A networking server framework and client library for school-based XO distributions that provides data persistence (in case of students not being allowed to own XOs) and student/teacher

communication (especially with bad network connections) with minimal developer work. Designed to work directly with Spyral.

Status: The current version needs to be rewritten, and the client-side front-end needs work. Consideration for network outages needs to be added to its core functionality. Lot's of work needs to be done with this.

5.1.4 Plan of Action

Plan and Procedure for Achieving the Stated Objectives:

Our plan of action is to iteratively develop our tools and deploy them in classroom settings and to the greater Python and OLPC communities. Using feedback from users we'll continually improve our systems with greater stability and innovative features.

5.1.5 Needs

Why is this project needed?

Presently, XO game development can be a harrowing task for beginners. Although true novices have Scratch, when they want to move onto more complicated programming their options are limited. PyGTK is more suitable for complicated applications, and neither Flash or Java has gained traction on the platform. HTML/Javascript game development is progressing, but suffers from speed and internet connectivity issues. Efforts to create a new language (KAGE) seem to have stalled, and the utility of teaching children a completely artificial game development language could be considered controversial. Pygame has historically been a favorite choice for game development.

However, Pygame is still an unnecessarily complicated system; for instance, do beginners need to comprehend the difference between the six different kinds of Groups that are available in Pygame? And the software engineering principles engendered by Pygame are very weak, with most games completely breaking from Model-View-Controller. Finally, most Pygame games suffer from being highly unoptimized, due to the high learning curve associated with understanding how best to optimize a Pygame game. XO activity development itself is also a difficult prospect, with XO files having a very precise and unforgiving structure. This can be a discouraging barrier for novice programmers from contributing to the program.

Locally?

Locally, the University of Delaware works with the Chester Community Charter School by teaching a course to undergraduate Computer Science majors about educational game development. Early iterations of the class suffered from spending an inordinate amount of time teaching how to program in Pygame, severely detracting from the quality of the games produced. After several iterations of the class, Robert Deaton created Spyral to simplify many of the common difficulties encountered and to provide a number of optimizations to the platform (e.g. automatic dirty sprite updating). At present, Spyral is used extensively in the class as a full Pygame replacement. Although still not in it's complete form, Spyral has already had an improvement on the games produced in the classroom, as evidenced by a small research study we've conducted where we compared games created pre- and post- Spyral in the Delaware class (http://platipy.org/publications/CHEP_2013.pdf). The games produced post-spyral are also available on our website.

As Spyral will be used for the foreseeable future in this class, it is very important that it continues to be developed along with its associated tools Conspyre, Platipy Docs, and Spur.

In the greater OLPC/Sugar community?

The tools being generated as a result of this project have great potential to be used by the broader Sugar community to develop games; they are open-source, free, powerful, and flexible tools for game development and thus can be used by anyone to make any kind of game.

Outside the community?

Spyral and its associated tools have great potential to be used outside of the project. In fact, Spyral is compatible with any system that provides Pygame, including the Raspberry Pi and Android (using the Pygame for Android Subset).

Why can't this project be done in emulation using non-XO machines?

Ultimately, rigorous testing is required in order to gauge the performance of our systems. Developing on a modern desktop computer will not give realistic information about the speed, reliability, etc. of a program on the XO. For that reason, we need XO laptops to develop on and test our examples and conduct unit/integration tests on.

Why are you requesting the number of machines you are asking for?

Although one of our members (Eric McGinnis) has direct access to the University of Delaware's XO Laptop library, our other two members do not. For their sake, we need XO laptops on which to test and develop Spyral.

Will you consider (1) salvaged/rebuilt or (2) damaged XO Laptops?

We can consider them, but damaged XOs might affect the results of our performance tests. Using them would be potentially suboptimal.

5.1.6 Sharing Deliverables

Project URL where you'll Blog specific ongoing progress:

http://platipy.org

how will you convey tentative ideas & results back to the OLPC/Sugar community, prior to completion?

Our primary form of communication will be through the official Platipy blog. However, as a natural consequence of our development process, we'll be keeping all of our respective githubs up-to-date. Additionally, we will contact the OLPC listservs, news outlets, and relevant online communities at important milestones.

How will the final fruits of your labor be distributed to children or community members worldwide?

All resources generated by this project will be available on public facing websites. Additionally, we will update official resources such as the Laptop Wiki with links and information pertaining to using our tools. Finally, we will notify the relevant blogs and news sources after each important release.

Will your work have any possible application or use outside our community?

Yes, our work will have extensive application outside of the OLPC community as previously described. We will use similar means to reach out to external communities, including contacting news sources, posting on sites like r/python, etc.

Have you investigated working with nearby XO Lending Libraries or Project Groups?

We will be working with the Project Group at the University of Delaware and the XO distribution at Chester Community Charter School. Austin will be investigating establishing a Project Group at Virginia Tech, as there are no groups local to that area.

5.1.7 Quality/Mentoring

Would your Project benefit from Support, Documentation and/or Testing people?

Yes. Software should always be tested, and we can benefit from having external eyes.

Teachers' input into Usability?

Minimally. Most of our work is oriented towards developers, not teachers.

How will you promote your work?

Through an official blog, online technology communities such as Hacker News, r/python, etc., and the official OLPC listservs.

Can we help you with an experienced mentor from the OLPC/Sugar community?

Yes, an experienced mentor could be useful, who would be knowledgeable about the ways that our project could be fit into the OLPC/Sugar community.

5.1.8 Timeline (Start to Finish)

Please include a Proposed timeline for your Project life-cycle. (this can be in the form of Month 1, Month 2, etc rather than specific dates)

Month	Goal
1	Finished version 1.0 of Spyral
2	Finished version 1.0 of Platipy Docs
3	Finished version 1.0 of Example.Activity
4	Finished version 1.0 of Conspyre
5-12	Iteratively improve the products

Specify how you prefer to communicate your ongoing progress and obstacles:

Through our official site (http://platipy.org)

5.2 Games Gallery

5.2.1 Educational

Coming soon!

5.2.2 General

Coming soon!

5.3 Open Tasks

Although our github is the canonical source for new issues, we wanted a nicely curated list of the "Big Issues" still present in our tools. If you are interested in contributing to an open-source project to support education of disadvantaged youth, please get involved!

5.3.1 Spyral

- Advanced physics through Box2D
- Audio API (perhaps even just exposing Pygame's)
- More widgets (e.g., radio button, drop-down menus, multi-line textbox)
- Unit tests
- Absolute positioning: Sprites currently report their position within parents, not an absolute position on the screen
- Static sprites: sprites that will always be static, emulating part of the background
- Hex-grids: Currently, only square regions are intelligently handled by Spyral, it'd be nice to have Hex
- Widget disabling
- Finalize actors
- Creating and destroying widgets on the fly
- Clipboard functionality
- Caption and Icon API

5.3.2 Example.Activity

- Live/Recorded Event handling integration
- PyqVer checking
- Application freezing: Exporting runnable games to windows, mac

5.3.3 Platipy

- Tutorial on layering
- Tutorial on forms
- Tutorial on events
- Tutorial on Actors
- Tutorial on Animations
- Tutorial on Views

Advanced tutorials on

5.3.4 Conspyre

CHAPTER 6

Release Information

6.1 Latest Versions

The latest version of Example.activity is v0.3. The latest version of spyral is v0.9.6

6.1.1 Downloads

To update spyral to a newer version than the one included with your launcher, remove libraries/spyral and replace with the downloaded version.

- `Download Pong.v2` (includes spyral)
- `Download Spyral.v0.9.6`

Old Versions

- `Download Spyral v0.9.6`
- `Download Spyral v0.9.5`
- `Download Spyral v0.9.4`
- `Download Spyral v0.9.3`
- `Download Spyral v0.9.2`
- `Download Spyral v0.9.1`
- Download spyral v0.2
- Download spyral v0.1.1
- Download spyral v0.1
- `Download Example.activity v0.2` (Includes spyral v0.2)
- `Download Example.activity v0.1` (Includes spyral v0.1)

6.1.2 Git Repositories

The repositories are hosted on github:

- spyral

- Example.activity

6.2 Changelogs

6.2.1 Spyral

v0.9.6

- [sprites] Fixed killing a sprite clobbering other events
- [animations] Fixed animations not firing the right events
- [core] Fixed error message being triggered when game ends ("spyral.quit cannot be found")

v0.9.5

- [collision] Fixed collision detection for forms (e.g., buttons, text input) and sprites when the dev-launcher is scaled
- [forms] Fixed using tabs to navigate forms crashing the game
- [mouse] Fixed mouse motion reporting the accurately scaled position
- [forms] Enhanced error message for declaring forms.

v0.9.4

- [mouse] Fix mouse coordinates in mouse events being incorrect with dev-launcher's scaling
- [scene] Fix background image not updating on the screen even after it's been initially set
- [events] Fix Minimizing your window and then maximizing no longer crashes with an event error

v0.9.0

- Massive, massive changes to the Spyral API
- New classes like Views, Actors, etc.
- Basicallly, it's a whole new system

v0.2.1

- [animation] Fix a logic bug in animations that would cause them to not catch an exception they should
- [camera] Fix a broken return value on camera.get_rect()
- [docs] Fix a documentation bug which used the wrong terminology
- [image] Fix image.get_size() to return a Vec2D instead of a 2-tuple
- [image] Fix image.draw_lines() not respecting the width parameter
- [image] Fix an issue where changes to images would not be rendered to the screen
- [image] Fix color channels being swapped on certain systems, in particular under scaling

- [rect] Fix a bug which didn't allow rects properly in the constructor to rects
- [rect] Fix a bug with rect.inflate and rect.inflate_ip which would cause a crash
- [sprite] Fix sprite.width and sprite.height
- [sprite] Fix sprite.visible not hiding sprites which were marked as static
- [sprite] Fix AggregateSprites only rendering some children
- [sprite] Fix AggregateSprites requiring a group be passed on their creation
- [sprite] Fix AggregateSprites not being able to have AggregateSprites as children
- [vector] Fix Vec2D's support for being tested inside containers based on hashes (i.e. sets)
- [vector] Fix Vec2D's support for being divided by a constant or another 2-tuple/Vec2D

New Features

- Add spyral.Image.draw_arc()
- spyral is now compatible with pypi and available in the cheeseshop
- Add support for object-attribute access for built-in event types
- A new rendering backend which is easier to maintain
- :above and :below modifiers for layers
- Add a more complete spyral.event.keys object
- Add official support for LiveEventHandler and ReplayEventHandler, with documentation

v0.2 - 10/02/2012

Backwards Incompatible Changes

- spyral.Sprite and spyral.Group now must have dt passed in as their first argument. (This was in the examples anyways)
- Remove sprite.blend_flags, was broken anyways. May be back in future release

New Features

- Add spyral.Vec2D, sprite.pos and sprite.scale are now Vec2D automatically
- Add spyral.Signal to spyral, as well as a number of useful signals in the docs
- Add draw_image, rotate, copy, scale, crop, flip to spyral.Image
- Add support for anchor-based positioning in spyral.Image methods
- Add sprite.scale, sprite.scale_x, sprite.scale_y, and sprite.angle, with animation support
- Add sprite.flip_x and sprite.flip_y
- Animations no longer require AnimationSprite or AnimationGroup objects, they work on standard sprites and groups
- Add spyral.Font
- Add AggregateSprite

Bug Fixes

- Fix VIDEORESIZE events crashing spyral
- Fix a bug with parallel animations not evaluating their ending condition
- Fix a bug with group.empty calling remove on sprites
- Fix a bug where sprites were being set static even when they weren't
- Fix a bug where static sprites were redrawn without clearing behind them
- Fix a frame count bug in the Iteration animator, making it more smooth
- Fix the import system, allowing the import of spyral's submodules again
- Fix a bug in rect.move_ip, previously the offsets would become the new coordinates
- Fix a limitation on the number of layers which a game could have

Miscellaneous

- Remove the legacy spyral.util module
- Remove spyral/docs in favor of documentation in platipy
- Remove sprite.blend_flags, was broken anyways. May be back in future release
- Remove the antiquated and broken examples/pong.py
- Major revisions to built-in documentation.

v0.1.1 - 09/19/2012

- Fix group.remove() to ensure sprites are no longer drawn upon removal
- Fix rect.collide_rect(), results were previously inverted.

v0.1 - 09/18/2012

- First release

6.2.2 Example.activity

v0.2 - 10/02/2012

- Fix generation of PNGs for profiling paths with spaces in them
- Fix activity.py launcher loading games before the directory was initialized
- Bump spyral to v0.2

v0.1 - 09/18/2012

- First release

6.3 Contact Developers / Submit Changes

If there is a bug in spyral or Example.activity, please do one of the following:

- open a ticket on our github,
- e-mail rdeaton@platipy.org and acbart@vt.edu to notify the authors directly, or
- send a pull request

S

spyral.director, 45
spyral.easing, 46
spyral.event, 47
spyral.image, 52
spyral.keyboard, 56
spyral.mouse, 56