CHAPTER 2 – PYGAME BASICS

Setting Up a Pygame Program

The first few lines of code in the Hello World program are lines that will begin almost every program you write that uses Pygame.

```
1. import pygame, sys
```

Line 1 is a simple import statement that imports the pygame and sys modules so that our program can use the functions in them. All of the Pygame functions dealing with graphics, sound, and other features that Pygame provides are in the pygame module.

Note that when you import the pygame module you automatically import all the modules that are in the pygame module as well, such as pygame.images and pygame.mixer.music. There's no need to import these modules-inside-modules with additional import statements.

```
2. from pygame.locals import *
```

Line 2 is also an import statement. However, instead of the import modulename format, it uses the from modulename import * format. Normally if you want to call a function that is in a module, you must use the modulename.functionname() format after importing the module. However, with from modulename import *, you can skip the modulename. portion and simply use functionname() (just like Python's built-in functions).

The reason we use this form of import statement for pygame.locals is because pygame.locals contains several constant variables that are easy to identify as being in the pygame.locals module without pygame.locals. in front of them. For all other modules, you generally want to use the regular import modulename format. (There is more information about why you want to do this at http://invpy.com/namespaces.)

```
4. pygame.init()
```

Line 4 is the pygame.init() function call, which always needs to be called after importing the pygame module and before calling any other Pygame function. You don't need to know what this function does, you just need to know that it needs to be called first in order for many Pygame functions to work. If you ever see an error message like pygame.error: font not initialized, check to see if you forgot to call pygame.init() at the start of your program.

```
5. DISPLAYSURF = pygame.display.set_mode((400, 300))
```

Line 5 is a call to the pygame.display.set_mode() function, which returns the pygame.Surface object for the window. (Surface objects are described later in this chapter.) Notice that we pass a tuple value of two integers to the function: (400, 300). This tuple tells the set_mode() function how wide

and how high to make the window in pixels. (400, 300) will make a window with a width of 400 pixels and height of 300 pixels.

Remember to pass a tuple of two integers to set_mode(), not just two integers themselves. The correct way to call the function is like this: pygame.display.set_mode((400, 300)). A function call like pygame.display.set_mode(400, 300) will cause an error that looks like this: TypeError: argument 1 must be 2-item sequence, not int.

The pygame.Surface object (we will just call them Surface objects for short) returned is stored in a variable named DISPLAYSURF.

```
6. pygame.display.set_caption('Hello World!')
```

Line 6 sets the caption text that will appear at the top of the window by calling the pygame.display.set_caption() function. The string value 'Hello World!' is passed in this function call to make that text appear as the caption:

Game Loops and Game States

```
7. while True: # main game loop
```

```
8.     for event in pygame.event.get():
```

Line 7 is a while loop that has a condition of simply the value True. This means that it never exits due to its condition evaluating to False. The only way the program execution will ever exit the loop is if a break statement is executed (which moves execution to the first line after the loop) or sys.exit() (which terminates the program). If a loop like this was inside a function, a return statement will also move execution out of the loop (as well as the function too).

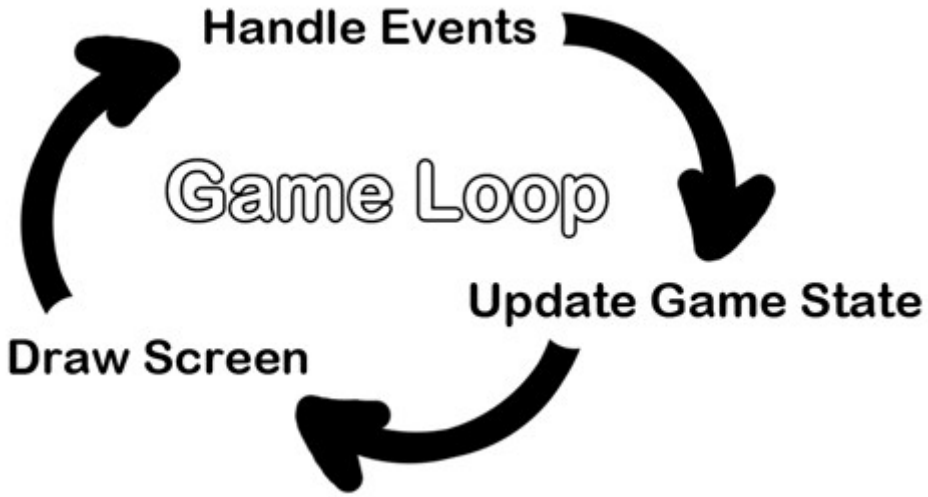
The games in this book all have these while True loops in them along with a comment calling it the “main game loop”. A game loop (also called a main loop) is a loop where the code does three things:

1. Handles events.
2. Updates the game state.
3. Draws the game state to the screen.

The game state is simply a way of referring to a set of values for all the variables in a game program. In many games, the game state includes the values in the variables that tracks the player’s health and position, the health and position of any enemies, which marks have been made on a board, the score, or whose turn it is. Whenever something happens like the player taking damage (which lowers their health value), or an enemy moves somewhere, or something happens in the game world we say that the game state has changed.

If you've ever played a game that let you saved, the "save state" is the game state at the point that you've saved it. In most games, pausing the game will prevent the game state from changing.

Since the game state is usually updated in response to events (such as mouse clicks or keyboard presses) or the passage of time, the game loop is constantly checking and re-checking many times a second for any new events that have happened. Inside the main loop is code that looks at which events have been created (with Pygame, this is done by calling the pygame.event.get() function). The main loop also has code that updates the game state based on which events have been created. This is usually called event handling.



pygame.event.Event Objects

Any time the user does one of several actions (they are listed later in this chapter) such as pressing a keyboard key or moving the mouse on the program's window, a pygame.event.Event object is created by the Pygame library to record this "event". (This is a type of object called Event that exists in the event module, which itself is in the pygame module.) We can find out which events have happened by calling the pygame.event.get() function, which returns a list of pygame.event.Event objects (which we will just call Event objects for short).

The list of Event objects will be for each event that has happened since the last time the pygame.event.get() function was called. (Or, if pygame.event.get() has never been called, the events that have happened since the start of the program.)

```
7. while True: # main game loop
```

```
8.     for event in pygame.event.get():
```

Line 8 is a for loop that will iterate over the list of Event objects that was returned by pygame.event.get(). On each iteration through the for loop, a variable named event will be assigned the value of the next event object in this list. The list of Event objects returned from pygame.event.get() will be in the order that the events happened. If the user clicked the mouse and then pressed a keyboard key, the Event object for the mouse click would be the first item in the list and the Event object for the keyboard press would be second. If no events have happened, then pygame.event.get() will return a blank list.

The QUIT Event and pygame.quit() Function

```
9.      if event.type == QUIT:
```

```
10.         pygame.quit()
```

```
11.         sys.exit()
```

Event objects have a member variable (also called attributes or properties) named type which tells us what kind of event the object represents. Pygame has a constant variable for each of possible types in the pygame.locals modules. Line 9 checks if the Event object's type is equal to the constant QUIT. Remember that since we used the from pygame.locals import * form of the import statement, we only have to type QUIT instead of pygame.locals.QUIT.

If the Event object is a quit event, then the pygame.quit() and sys.exit() functions are called. The pygame.quit() function is sort of the opposite of the pygame.init() function: it runs code that deactivates the Pygame library. Your programs should always call pygame.quit() before they call sys.exit() to terminate the program. Normally it doesn't really matter since Python closes it when the program exits anyway. But there is a bug in IDLE that causes IDLE to hang if a Pygame program terminates before pygame.quit() is called.

Since we have no if statements that run code for other types of Event object, there is no event-handling code for when the user clicks the mouse, presses keyboard keys, or causes any other type of Event objects to be created. The user can do things to create these Event objects but it doesn't change anything in the program because the program does not have any event-handling code for these types of Event objects. After the for loop on line 8 is done handling all the Event objects that have been returned by pygame.event.get(), the program execution continues to line 12.

```
12.     pygame.display.update()
```

Line 12 calls the pygame.display.update() function, which draws the Surface object returned by pygame.display.set_mode() to the screen (remember we stored this object in the DISPLAYSURF variable). Since the Surface object hasn't changed (for example, by some of the drawing functions that are explained later in this chapter), the same black image is redrawn to the screen each time pygame.display.update() is called.

That is the entire program. After line 12 is done, the infinite while loop starts again from the beginning. This program does nothing besides make a black window appear on the screen, constantly check for a QUIT event, and then redraws the unchanged black window to the screen over and over again. Let's learn how to make interesting things appear on this window instead of just blackness by learning about pixels, Surface objects, Color objects, Rect objects, and the Pygame drawing functions.

Surface Objects and The Window

Surface objects are objects that represent a rectangular 2D image. The pixels of the Surface object can be changed by calling the Pygame drawing functions (described later in this chapter) and then displayed on the screen. The window border, title bar, and buttons are not part of the display Surface object.

In particular, the Surface object returned by pygame.display.set_mode() is called the display Surface. Anything that is drawn on the display Surface object will be displayed on the window when the pygame.display.update() function is called. It is a lot faster to draw on a Surface object (which only exists in the computer's memory) than it is to draw a Surface object to the computer screen. Computer memory is much faster to change than pixels on a monitor.

Often your program will draw several different things to a Surface object. Once you are done drawing everything on the display Surface object for this iteration of the game loop (called a frame, just like a still image on a paused DVD is called) on a Surface object, it can be drawn to the screen. The computer can draw frames very quickly, and our programs will often run around 30 frames per second (that is, 30 FPS). This is called the "frame rate" and is explained later in this chapter.

Drawing on Surface objects will be covered in the "Primitive Drawing Functions" and "Drawing Images" sections later this chapter.

Rect Objects

Pygame has two ways to represent rectangular areas (just like there are two ways to represent colors). The first is a tuple of four integers:

1. The X coordinate of the top left corner.
2. The Y coordinate of the top left corner.
3. The width (in pixels) of the rectangle.
4. Then height (in pixels) of the rectangle.

The second way is as a pygame.Rect object, which we will call Rect objects for short. For example, the code below creates a Rect object with a top left corner at (10, 20) that is 200 pixels wide and 300 pixels tall:

```
>>> import pygame
```

```
>>> spamRect = pygame.Rect(10, 20, 200, 300)
```

```
>>> spamRect == (10, 20, 200, 300)
```

```
True
```

The handy thing about this is that the Rect object automatically calculates the coordinates for other features of the rectangle. For example, if you need to know the X coordinate of the right edge of the pygame.Rect object we stored in the spamRect variable, you can just access the Rect object's right attribute:

```
>>> spamRect.right
```

```
210
```

The Pygame code for the Rect object automatically calculated that if the left edge is at the X coordinate 10 and the rectangle is 200 pixels wide, then the right edge must be at the X coordinate 210. If you reassign the right attribute, all the other attributes are automatically recalculated:

```
>>> spamRect.right = 350
```

```
>>> spamRect.left
```

```
150
```

Here's a list of all the attributes that pygame.Rect objects provide (in our example, the variable where the Rect object is stored in a variable named spamRect):

Attribute Name	Description
myRect.left	The int value of the X-coordinate of the left edge of the rectangle.
myRect.right	The int value of the X-coordinate of the right edge of the rectangle.
myRect.top	The int value of the Y-coordinate of the top edge of the rectangle.
myRect.bottom	The int value of the Y-coordinate of the bottom edge.
myRect.centerx	The int value of the X-coordinate of the center of the rectangle.
myRect.centery	The int value of the Y-coordinate of the center of the rectangle.
myRect.width	The int value of the width of the rectangle.
myRect.height	The int value of the height of the rectangle.
myRect.size	A tuple of two ints: (width, height)
myRect.topleft	A tuple of two ints: (left, top)
myRect.topright	A tuple of two ints: (right, top)
myRect.bottomleft	A tuple of two ints: (left, bottom)
myRect.bottomright	A tuple of two ints: (right, bottom)
myRect.midleft	A tuple of two ints: (left, centery)

myRect.midright	A tuple of two ints: (right, centery)
myRect.midtop	A tuple of two ints: (centerx, top)
myRect.midbottom	A tuple of two ints: (centerx, bottom)

Primitive Drawing Functions

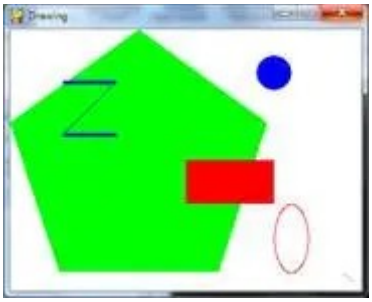
Pygame provides several different functions for drawing different shapes onto a surface object. These shapes such as rectangles, circles, ellipses, lines, or individual pixels are often called drawing primitives. Open IDLE's file editor and type in the following program, and save it as *drawing.py*.

```
1. import pygame, sys

2. from pygame.locals import *

3.

4. pygame.init()

5.

6. # set up the window

7. DISPLAYSURF = pygame.display.set_mode((500, 400), 0, 32)

8. pygame.display.set_caption('Drawing')

9.

10. # set up the colors

11. BLACK = (  0,   0,   0)

12. WHITE = (255, 255, 255)

13. RED = (255,   0,   0)
```

```
14. GREEN = (  0, 255,   0)

15. BLUE = (  0,   0, 255)

16.

17. # draw on the surface object

18. DISPLAYSURF.fill(WHITE)

19. pygame.draw.polygon(DISPLAYSURF, GREEN, ((146, 0), (291, 106), (236, 277), (56, 277),
(0, 106)))

20. pygame.draw.line(DISPLAYSURF, BLUE, (60, 60), (120, 60), 4)

21. pygame.draw.line(DISPLAYSURF, BLUE, (120, 60), (60, 120))

22. pygame.draw.line(DISPLAYSURF, BLUE, (60, 120), (120, 120), 4)

23. pygame.draw.circle(DISPLAYSURF, BLUE, (300, 50), 20, 0)

24. pygame.draw.ellipse(DISPLAYSURF, RED, (300, 250, 40, 80), 1)

25. pygame.draw.rect(DISPLAYSURF, RED, (200, 150, 100, 50))

26.

27. pixObj = pygame.PixelArray(DISPLAYSURF)

28. pixObj[480][380] = BLACK

29. pixObj[482][382] = BLACK

30. pixObj[484][384] = BLACK
```

```
31. pixObj[486][386] = BLACK
32. pixObj[488][388] = BLACK
33. del pixObj
34.
35. # run the game loop
36. while True:
37.     for event in pygame.event.get():
38.         if event.type == QUIT:
39.             pygame.quit()
40.             sys.exit()
41.     pygame.display.update()
```

When this program is run, the following window is displayed until the user closes the window:



Notice how we make constant variables for each of the colors. Doing this makes our code more readable, because seeing GREEN in the source code is much easier to understand as representing the color green than (0, 255, 0) is.

The drawing functions are named after the shapes they draw. The parameters you pass these functions tell them which Surface object to draw on, where to draw the shape (and what size), in what color, and how wide to make the lines. You can see how these functions are called in the *drawing.py* program, but here is a short description of each function:

· **fill(color)** – The fill() method is not a function but a method of pygame.Surface objects. It will completely fill in the entire Surface object with whatever color value you pass as for the color parameter.

· **pygame.draw.polygon(surface, color, pointlist, width)** – A polygon is shape made up of only flat sides. The surface and color parameters tell the function on what surface to draw the polygon, and what color to make it.

The pointlist parameter is a tuple or list of points (that is, tuple or list of two-integer tuples for XY coordinates). The polygon is drawn by drawing lines between each point and the point that comes after it in the tuple. Then a line is drawn from the last point to the first point. You can also pass a list of points instead of a tuple of points.

The width parameter is optional. If you leave it out, the polygon that is drawn will be filled in, just like our green polygon on the screen is filled in with color. If you do pass an integer value for the width parameter, only the outline of the polygon will be drawn. The integer represents how many pixels width the polygon's outline will be. Passing 1 for the width parameter will make a skinny polygon, while passing 4 or 10 or 20 will make thicker polygons. If you pass the integer 0 for the width parameter, the polygon will be filled in (just like if you left the width parameter out entirely).

All of the pygame.draw drawing functions have optional width parameters at the end, and they work the same way as pygame.draw.polygon()'s width parameter. Probably a better name for the width parameter would have been thickness, since that parameter controls how thick the lines you draw are.

· **pygame.draw.line(surface, color, start_point, end_point, width)** – This function draws a line between the start_point and end_point parameters.

· **pygame.draw.lines(surface, color, closed, pointlist, width)** – This function draws a series of lines from one point to the next, much like pygame.draw.polygon(). The only difference is that if you pass False for the closed parameter, there will not be a line from the last point in the pointlist parameter to the first point. If you pass True, then it will draw a line from the last point to the first.

· **pygame.draw.circle(surface, color, center_point, radius, width)** – This function draws a circle. The center of the circle is at the center_point parameter. The integer passed for the radius parameter sets the size of the circle.

The radius of a circle is the distance from the center to the edge. (The radius of a circle is always half of the diameter.) Passing 20 for the radius parameter will draw a circle that has a radius of 20 pixels.

· **pygame.draw.ellipse(surface, color, bounding_rectangle, width)** – This function draws an ellipse (which is like a squashed or stretched circle). This function has all the usual parameters, but in order to tell the function how large and where to draw the ellipse, you must specify the bounding rectangle of the ellipse. A bounding rectangle is the smallest rectangle that can be drawn around a shape. Here's an example of an ellipse and its bounding rectangle:

The bounding_rectangle parameter can be a pygame.Rect object or a tuple of four integers. Note that you do not specify the center point for the ellipse like you do for the pygame.draw.circle() function.

· **pygame.draw.rect(surface, color, rectangle_tuple, width)** – This function draws a rectangle. The rectangle_tuple is either a tuple of four integers (for the XY coordinates of the top left corner, and the width and height) or a pygame.Rect object can be passed instead. If the rectangle_tuple has the same size for the width and height, a square will be drawn.

pygame.PixelArray Objects

Unfortunately, there isn't a single function you can call that will set a single pixel to a color (unless you call pygame.draw.line() with the same start and end point). The Pygame framework needs to run some code behind the scenes before and after drawing to a Surface object. If it had to do this for every single pixel you wanted to set, your program would run much slower. (By my quick testing, drawing pixels this way is two or three times slower.)

Instead, you should create a pygame.PixelArray object (we'll call them PixelArray objects for short) of a Surface object and then set individual pixels. Creating a PixelArray object of a Surface object will "lock" the Surface object. While a Surface object is locked, the drawing functions can still be called on it, but it cannot have images like PNG or JPG images drawn on it with the blit() method. (The blit() method is explained later in this chapter.)

If you want to see if a Surface object is locked, the get_locked() Surface method will return True if it is locked and False if it is not.

The PixelArray object that is returned from pygame.PixelArray() can have individual pixels set by accessing them with two indexes. For example, line 28's pixObj[480][380] = BLACK will set the pixel at X coordinate 480 and Y coordinate 380 to be black (remember that the BLACK variable stores the color tuple (0, 0, 0)).

To tell Pygame that you are finished drawing individual pixels, delete the PixelArray object with a del statement. This is what line 33 does. Deleting the PixelArray object will "unlock" the Surface object so that you can once again draw images on it. If you forget to delete the PixelArray object, the next time you try to blit (that is, draw) an image to the Surface the program will raise an error that says, "pygame.error: Surfaces must not be locked during blit".

The pygame.display.update() Function

After you are done calling the drawing functions to make the display Surface object look the way you want, you must call pygame.display.update() to make the display Surface actually appear on the user's monitor.

The one thing that you must remember is that pygame.display.update() will only make the display Surface (that is, the Surface object that was returned from the call to pygame.display.set_mode()) appear on the screen. If you want the images on other Surface objects to appear on the screen, you must "blit" them (that is, copy them) to the display Surface object with the blit() method (which is explained next in the "Drawing Images" section).

Frames Per Second and pygame.time.Clock Objects

A pygame.time.Clock object can help us make sure our program runs at a certain maximum FPS. This Clock object will ensure that our game programs don't run too fast by putting in small pauses on each iteration of the game loop. If we didn't have these pauses, our game program would run as fast as the computer could run it. This is often too fast for the player, and as computers get faster they would run the game faster too. A call to the tick() method of a Clock object in the game loop can make sure the game runs at the same speed no matter how fast of a computer it runs on. The Clock object is created on line 7 of the *catanimation.py* program.

```
7. fpsClock = pygame.time.Clock()
```

The Clock object's tick() method should be called at the very end of the game loop, after the call to pygame.display.update(). The length of the pause is calculated based on how long it has been since the previous call to tick(), which would have taken place at the end of the previous iteration of the game loop. (The first time the tick() method is called, it doesn't pause at all.) In the animation program, is it run on line 47 as the last instruction in the game loop.

All you need to know is that you should call the tick() method once per iteration through the game loop at the end of the loop. Usually this is right after the call to pygame.display.update().

```
47.     fpsClock.tick(FPS)
```

Try modifying the FPS constant variable to run the same program at different frame rates. Setting it to a lower value would make the program run slower. Setting it to a higher value would make the program run faster.

Drawing Images with pygame.image.load() and blit()

The image of the cat was stored in a file named *cat.png*. To load this file's image, the string 'cat.png' is passed to the pygame.image.load() function. The pygame.image.load() function call will return a Surface object that has the image drawn on it. This Surface object will be a separate Surface object from the display Surface object, so we must blit (that is, copy) the image's Surface object to the display Surface object. Blitting is drawing the contents of one Surface onto another. It is done with the blit() Surface object method.

```
39.     DISPLAYSURF.blit(catImg, (catx, caty))
```

Line 39 of the animation program uses the blit() method to copy catImg to DISPLAYSURF. There are two parameters for blit(). The first is the source Surface object, which is what will be copied onto the DISPLAYSURF Surface object. The second parameter is a two-integer tuple for the X and Y values of the topleft corner where the image should be blitted to.

If catx and caty were set to 100 and 200 and the width of catImg was 125 and the height was 79, this blit() call would copy this image onto DISPLAYSURF so that the top left corner of the catImg was at the XY coordinate (100, 200) and the bottom right corner's XY coordinate was at (225, 279).

Note that you cannot blit to a Surface that is currently “locked” (such as when a PixelArray object has been made from it and not yet been deleted.)

The rest of the game loop is just changing the catx, caty, and direction variables so that the cat moves around the window. There is also a call to pygame.event.get() to handle the QUIT event.