CSci 1133
Lab Exercise 14

Pygame — Coding a Breakout Game Clone

One of the primary reasons Python has become such a popular programming language is the availability of an extensive collection of "packages" developed by 3rd parties and offered for low/no cost via download from Internet web sites. These packages include extended mathematical libraries such as **Numpy** (a Matlab-like environment), scientific libraries such as **Scipy**, plotting and visualization libraries like **Matplotlib** and hundreds of others. Many of these packages are translated into machine code, so they execute very quickly and efficiently, providing a "best of all worlds" scenario in which you can employ pre-compiled classes and methods from the package for the parts of your program that need to execute rapidly, and regular (interpreted) Python code for the bulk of your program that does not require blindingly-fast execution.

In this week's lab, we will be using a fast, powerful graphics package called **Pygame**. It's a relatively simple system that supports highly abstract graphical interfaces in Python and allows programmers (you!) to quickly create simple video games. The goals of this lab are to: a) use all of your newly acquired knowledge to code something tangible that you can show-off, i.e., your own version of a Breakout video game clone:

https://en.wikipedia.org/wiki/Breakout_clone

b) fully grasp the power of black box systems and method abstraction, and c) hopefully inspire you to continue to use Python, even if you do not continue taking coding classes. However, more than anything else, this lab is for you to *have fun*. As always, if you have any questions or need help, do not hesitate to ask any of the TAs.

You will find that the Pygame documentation is incredibly helpful (http://www.pygame.org/docs/). It includes both explanations of the various methods Pygame includes as well as example code. Good luck!

Warm-up

Before you can get to coding the actual game, it is important to first familiarize yourself with Pygame's general interface. The goal of this warm-up is to create a simple program that opens up a window and changes the screen color every time a "player" presses the spacebar.

1. First, import the `pygame` and `sys` modules. You need to import pygame in order to use any of its methods, and sys will be used as a handy way to quit the program.

2. Next, create a `main` function (the remainder of the Warm-up will be implemented in this function). Make sure you instantiate the following within the body of the `main` function:
    a. Separate *width* and *height* variables for the screen size (600 by 600 is recommended)
    b. A *list* of at least three color-tuples. A color-tuple is a Python tuple containing 3 integer values representing the RGB (red, green, blue) components of the color. For example, black is (0,0,0), Red is (255,0,0), and white is (255,255,255).
    c. The animation-speed in frames-per-second (fps), which allows your game to run at a consistent speed on various computers. Make this 60 to begin with, although you can play around with it later to see how it affects the speed of the game.
    d. You will need to create and save a *display object* that will be used for updating the screen itself. This can be done by calling `pygame.display.set_mode((int,int))`, where the only parameter is a tuple in which the first value is the width of the screen and the second value is the height of the screen. Although not exactly the same, you can imagine this display object to be similar to the turtle screen.

e. You will also need to instantiate a clock object that will be used in coordination with the animation speed to update the screen. The pygame class for the clock object is `pygame.time.Clock()` (bind the object to a variable for future reference).

3. Call the method `pygame.display.set_caption(str)` to set the name of the window (shown on the top bar).

   At this point, you might notice that nothing happens if you run the code. That's to be expected! Since our screen isn't actually updating, nothing should happen yet. However, your code should also not bring up any errors. Check to make sure this is true before moving on.

4. The rest of this warm-up takes place in an (intentionally!) *infinite* loop, e.g., `while True:` We want our screen to continue updating forever unless the game ends or the user quits. The next thing you need to do is *update* the screen so that you can actually see what's going on. This will be done in three steps (in the body of your while loop):
   a. First, color the display using the `.fill(`*color-tuple*`)` method of the display object created in step 2d. For now, just use the first element in your color list defined above.
   b. Next, you need to update the display to represent what has been changed since the last update. To do this, call the method: `pygame.display.flip()`
   c. Finally, you need to keep track of how often to update your screen. Using the clock object created in step 2e, call the method `.tick(int)`, where the integer is the animation speed.

5. Before this code will actually work, you need to provide *event handling*. Just as in earlier labs where we used Turtle methods to process mouse clicks, Pygame has means to interpret various actions a user may take (you may be glad to know that it's also generally simpler than Turtle!). The following code loops through all the events that Pygame is detecting (such as mouse clicks and keyboard input) and checks to see if any of them is of the type `QUIT` e.g., the user clicked the little x on the top left of the screen. If so, we call `sys.exit()` to close the display and end the program. Copy this code to the *beginning* of the "infinite" while loop:

```
for event in pygame.event.get():
    if event.type == pygame.QUIT:
        sys.exit()
```

6. Finally, edit this code to determine if the user pressed the spacebar. Note that pressing a key counts as an event in Pygame, but specific keys do not create distinct events. Instead, you need to first check if the event type is `pygame.KEYDOWN` and, if so, further check if the specific *event key* (`event.key`) is the spacebar (`pygame.K_SPACE`). If the spacebar has been pressed, change the screen color. It's up to you how you do this. You can iterate through the color list you created in step 2 or select a new random color each time. If you have coded everything correctly, you should be able to run the script and see the screen color change every time you press space.

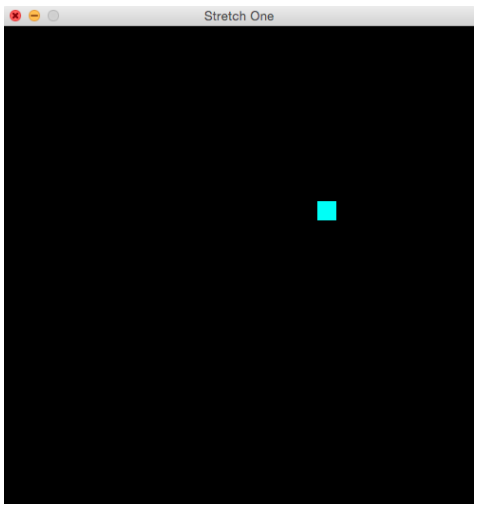
Make sure you understand everything so far before moving on! If anything does not make sense, ask a TA or a fellow student. The rest of this lab will be difficult without understanding of the material thus far.

Stretch

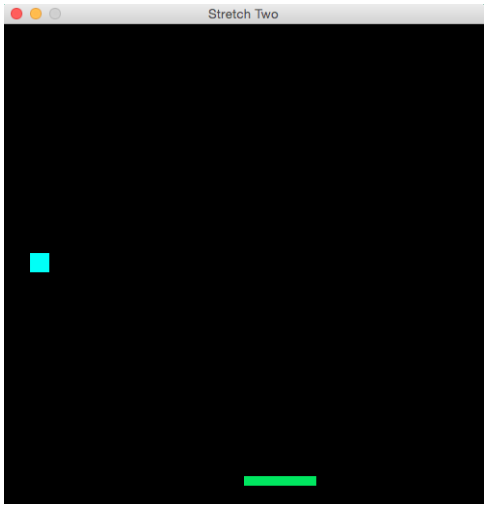
Now that you have some understanding of Pygame, you can move on to handling graphics animation and control. This stretch exercise is split into two parts. In the first part, you will extend the warm-up code you've written to make a small "block" bounce around the screen. In the second part, you will add a "paddle" that can bounce the block back and have the game quit if the paddle doesn't "catch" it. You can think of this as a breakout clone game without any blocks to hit (or an unwinnable form of Pong).

Both parts of the Stretch and the Workout will require you to use *images*—you can find images that we have already created in a folder on the class page. Download these and put them in the same folder as your Python source file. Although you can technically use whatever images you want, I would recommend coding with these ones first (they have already been correctly been sized).

Here are some example images of what your code may produce at the end of Part 1 and Part 2 of the Stretch:



Display at End of Part 1



Display at end of Part 2

1). Bouncing Block

One of the most useful things about Pygame is that it can create a rectangle object for any image you use. Rectangle objects include variables specifying their x,y coordinates and methods such as movement and collision detection with other rectangle objects. *Note: unlike Turtle, Pygame defines the top left of the window as (0,0) and everything below and to the right as larger.*

Steps 1 – 3 should happen before the infinite while loop in `main`.

1. First, you will need to upload the image for the block into Pygame. You can do this with the `pygame.image.load(str)` function, in which `str` is the name of the image. `.load()` will return an *image object*, make sure to bind this to a variable—Pygame will need to use it to actually display the image later.

2. Next, obtain a *rectangle* object for the image using the image `.get_rect()` method. The initial rectangle location will default to the top left of the screen, but you can move it by directly accessing its coordinates using its `top` and `left` public class variables.

3. Rectangle objects have a `move([int,int])` method that takes a *list* of coordinates and moves the rectangle by the first `int` in the x direction (right is positive, left is negative), and by the second `int` in the y direction (down is positive, up is negative). You should maintain the double `int` *list*

(I recommend using [2,2] to begin with) as a variable that you can later use to change the direction whenever the block "hits" a wall. In addition, you should call the `move` method in the main while loop so that the block will keep moving continuously. Make sure you rebind the return value to the rectangle object (or else it won't actually do anything—it is not a mutable data type)

4. In order to see the block appear on the screen, we need to explicitly *draw* it onto the screen. If your display object from the warm-up was called *display*, your image was saved to *image*, and your rectangle object was saved to the variable *imageRect*, the code to do this would be `display.blit(image, imageRect)`. Note: the second parameter in `blit` is actually an integer tuple (x,y) that tells the screen where to put the image (*imageRect* will get processed this way). Call this method before the `flip` method from before (otherwise it will never be processed in time to draw) and after the `fill` method from before (otherwise the screen will always just draw over the rectangle). If you run this code right now, you should get a block that moves in one direction until running off the screen.

   Do you see your square moving across the screen, but leaving a trail behind it? Do not fear, this is normal! Pygame does graphics by drawing on top of itself (just like Turtle!), so if you are not redrawing your display with fill, everything will leave behind a trail. Grab a TA or fix this yourself before moving on.

5. The only thing left to do is implement edge detection code that will change the direction the block is moving whenever the block runs into a wall. Create a separate method that takes in the rectangle object for the block, its speed list, and the dimensions of the screen. This method will look at the different coordinates of the sides of the rectangle object (if it was called rect, this would be `rect.top, rect.bottom, rect.left, and rect.right`) and see whether or not they are at the edge of the screen. If the block is at the edge, modify the movement xy list to "reflect" the rectangle. For example, if the block hits the right of the screen, the second integer in the block speed variable should become the opposite value (ex: 2 to -2) of what it was before. Do this in the body of your while loop to make sure the block never leaves the screen.

2). Paddle Defense!

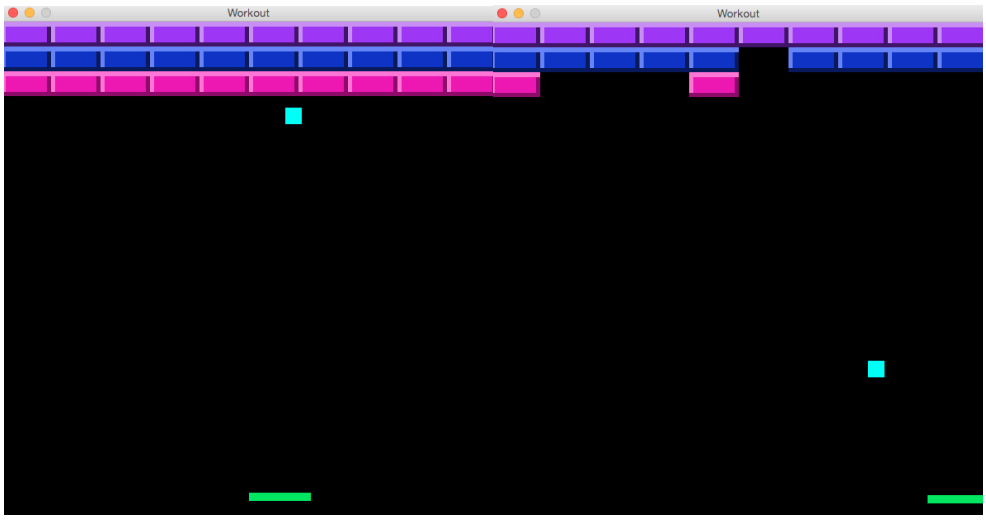
Using everything you just learned from Part One of the stretch and the warm-up, you should be able to create a paddle that moves left and right on the bottom of the screen, reversing directions based on left and right arrow key presses. If the block hits the paddle, bounce it back up. If the ball moves past the bottom of the screen, the user "loses" and the game should quit. In addition, you should stop the paddle from moving if it hits either edge of the screen (until the user turns it back). The only additional method you may need for this part is `rectObjectOne.colliderect(rectObjectTwo)`. As implied, `colliderect` is a method that belongs to a rectangle object that takes a second rectangle object as a parameter. It returns a boolean value representing whether or not the two rectangle objects are overlapping.

Note: for simplicity, you may assume that if the paddle and the block are touching, the block is on top of the rectangle. This can cause glitches if the block hits the side of the paddle, but the chances of such are small and you can wait to fix that until later in lab (if you have extra time).

If there is not much time left in lab or you do not feel fully confident in the code thus far (or you simply prefer this option), we recommend replacing the following Workout tasks with creating the game Pong. (Or... of course you can do Pong *and* the Workout! :-)

Workout

For the workout, you will be adding the “bricks” to the game. By the end of the workout, your display should look something like this:



Although this may seem like a hard task at first, it's not actually too different than what you've done thus far. The workout will not be as guided as the previous parts of this lab, but here are three generic helping tips we recommend you do (but not everything you will need):

1. Add a brick class that takes an image and a rectangle object of that image as constructor variables. In addition, there should at least be an accessor method for each variable.

2. Before the while loop in `main`, create a two dimensional list. Each list in the outer list will represent one row of bricks, and each element in the inner list should be a brick object. Tip: you can use the index of the list to find the x and y positions of the display. If you would like, you can also use a list of image strings to choose different colored bricks (the pictures above use the row index to choose which image to display).

3. Either edit an earlier method or create a new one that will compare the block to each element in the brick list. If they collide, move the block accordingly and delete the brick. This will be similar to the remove method in Lab 13—however, it is not necessary to redraw the bricks in this method. The bricks will need to be re-blitzed in the main while loop, so drawing that here as well would be excessive.

You are not required (or even encouraged!) to make your game look identical to the one above—have fun with it! The challenge also has some additional recommendations for ways you can personalize your game, so take a look once you finish the workout!

Challenge

Congratulations! If you've made it this far, you have already coded a basic functioning game… think about how jealous your non-coding friends will be. :) Feel free to go back to your code and fix some bugs if you have them (such as hitting the paddle at the side) or see what happens when you change certain variables (how much does the animation-speed really matter?). However, if you've already done that or have decided that you want to do something a little different, here are some suggestions and tips:

- First and foremost, you likely noticed something peculiar about this game: no matter what you do (unless you glitch it), the block always moves in the same path. In reality, the block path is predetermined from the beginning. Because the only thing it does it turn around, there's not much possibility for variability in it. If you would like to create a more realistic, or at least less repetitive, game, here are some options:
    - Change it so that the block reflects differently if the paddle is moving in the same direction as it, opposite direction as it, or no direction at all.
    - Create it so that the movement changes based on some random number each time (this wouldn't be very realistic, but it could spice things up!)
    - Use actual math to figure out how the ball should move with respect to its own speed and the paddle's speed. Especially if you decide to make it so that the paddle and/or ball may change speed somehow, this is the best way to go.

- Use a write method: Pygame actually has a way to write directly on the screen, though it requires a bit more work on your part than Turtle. You have to find the font and save it to a variable using the method `pygame.font.Font(str)`, in which str is a downloaded font. Next, you must render that font and save it to a different variable using the function `____.render(str,bool,(int,int,int))`, where `____` is whatever variable you used just above. The first parameter is what you wish to print, the second is whether or not you want smooth edges, and the third is an RGB color string. Every time you wish to change the string, you must reset that variable. The `render` function actually takes a string and a font and creates an image of it, so you will display it using `blit` like before. Here are some ideas of what you could do with this method:
    - create "lives"—instead of ending the game when the ball falls, subtract one life and try again
    - have a score count—every time a block breaks, add one!
    - display a game over screen—instead of ending when you lose, display a game instead (perhaps it shows the score or asks if the user wants to play again)

- In addition, you can add some more generic updates to what you already have:
    - You can create some fun and creative brick structures. Make a python! A narwal! A gopher! Whatever you want!
    - You can add different levels to the game—if the user clears all the bricks, then they get more bricks!
    - Give bricks "lives" of a sort. If you add a variable to the brick class that represents its remaining lives (as well as accessor and mutator methods for this variable), you can decrement the life of a brick by one every time it gets hit. This way, the brick only gets deleted when its life count hits zero. Tip: if you do it this way, I would recommend changing it so different images represent different lives. This will make it easier to debug (and more fun!).

- Another option is to add power-ups… after all, who doesn't like feeling super? How you implement these and what they do is up to you! Perhaps you have a powerup that spawns a second ball? Maybe it speeds up your paddle? The possibilities are limitless!

- You can also add music to your game—it could play as long as the game is open, or maybe it makes a sound every time the ball hits something. The documentation for music in Pygame is here: https://www.pygame.org/docs/ref/music.html.

- Using your code from the stretch, it is pretty simple to implement a two-player game of Pong. If you want, you can do just that! With some extra key-binding detection and collision managing, this should now be relatively easy.

Of course, you don't have to do any of this! If you have another brilliant idea, go for it! As a reminder, the link to the Pygame documentation is here: http://www.pygame.org/docs/.