CSE 1321L: Programming and Problem Solving I Lab

Lab 11

Introduction to Pygame

What students will learn:

- How to install Pygame in PyCharm
- How to initialize Pygame
- Creating and Blitting Surfaces
- Creating Rects
- Using Rects to move Surfaces

Content

Overview

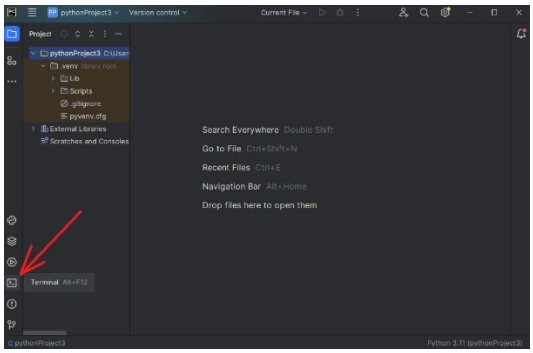
Over the next few labs, we will be learning about Pygame, which is a group of modules that make it easier to create your own games. Pygame is built on top of the Simple DirectMedia Layer (SDL) library, abstracting away many of its functionalities into readable Python code. While convenient, Pygame only provides the base building blocks, such as drawing things to the screen, moving them around, and accessing files on your computer: We'll still have to do most of the legwork when it comes to designing and implementing the rules of our game, which we'll discuss at a later lab.

If you are already comfortable with Surfaces, Rects, and rect.move(), skip to the last page.

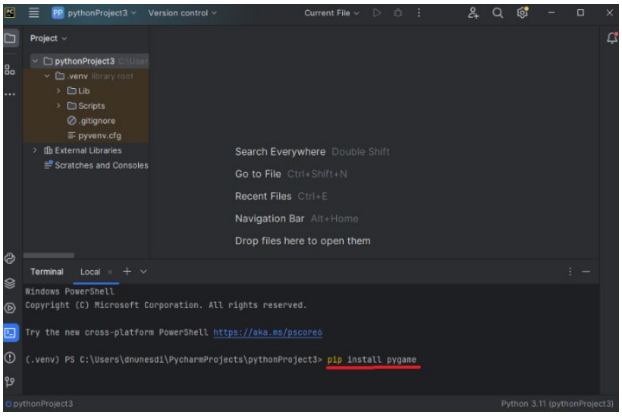
The following pages will discuss how to install Pygame, how to run it, how to draw things to the screen, and how to move them around.

Installing Pygame:

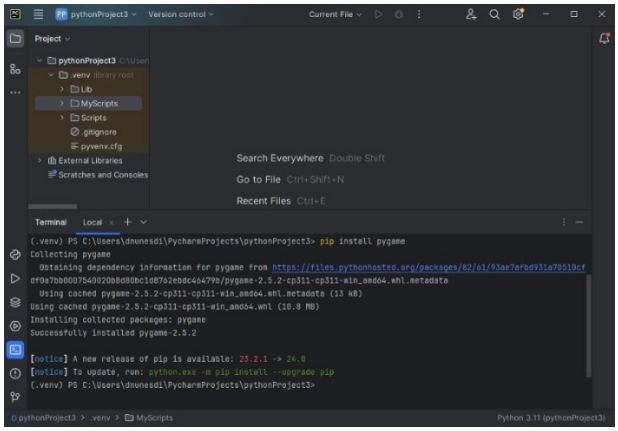
To Install Pygame, first create a project in PyCharm as normal. Once your project is up, click on the "Terminal" button at the bottom left of your PyCharm window:



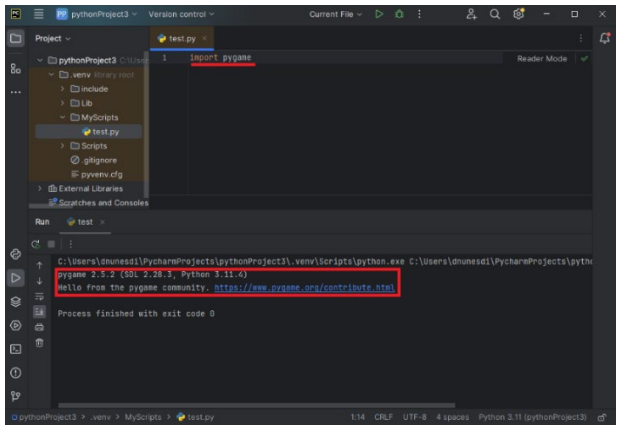
In the black window that appears, type in "`pip install pygame`":



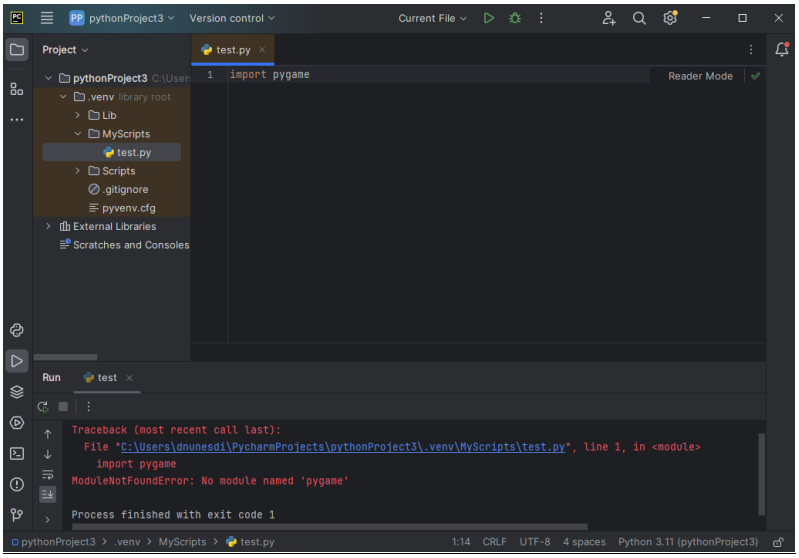
After a brief installation, Pygame should be installed in your PyCharm project. **Notice that this installation is only valid for your current project**. If you create a new project, you'll have to run the **pip command again**.



You can now test if Pygame is installed by creating a new Python file and trying to **import Pygame**. If your program runs without errors, Pygame has been installed, and you are good to go:



If you get an error message, something went wrong. Either try to run the pip command again and note the error message (if it gives you one) or reach out to your GTA for help.



Below, you can find the Pygame template we've been discussing in lecture. This template simply creates a black screen that is 500x500, which will go away if we press ESCAPE or if we click the close button at the top right. We'll be expanding on that template in this lab with what we've been learning in lecture.

```
import pygame, sys
from pygame.locals import *

# Initializes Pygame
pygame.init()

# Manage any external resources, such as files or networking
# currently empty
# Manage any special functions or classes our game needs
# currently empty
# Initializes the Display's Surface and saves it
resolution = (500, 500)
screen = pygame.display.set_mode(resolution)

# Creates the clock, so we can slow the game down
# to a manageable speed
clock = pygame.time.Clock()

# Main gameplay loop
while True:
    # Check which keys have been pressed
    keys = pygame.key.get_pressed()

    # Checks what events happened, and act on them.
    for event in pygame.event.get():
        if event.type == QUIT:
```

```
            sys.exit(0)
        if keys[pygame.K_ESCAPE]:
            sys.exit(0)

    # Paint the whole screen back
    screen.fill(color=(0,0,0))

    # Update the Display with the contents of the Display's surface
    pygame.display.flip()

    # Slow the game to update 60 times per second
    clock.tick(60)
```

Pygame Surfaces:

Pygame has something called a Surface, which is an object that stores pixel information. The idea is to create a Surface with the images that you want, and then draw those Surfaces to the screen, which is itself also a Surface. Creating a Surface is relatively straightforward: create a variable like any other using the Surface constructor. This constructor takes in a pair of numbers, which represent the width and the height of the Surface. The Surface below is 50x50 pixels:

```
surf1 = pygame.Surface((50,50)) # notice the double-parenthesis
```

The Surface we created contains 2500 pixels (50x50). Later, we'll learn how to add an image to our Surfaces but, for now, let's just paint it all red:

```
surf1.fill((255,0,0)) # red-colored Surface
```

The fill() method, when supplied with a trio of numbers, will paint the Surface that color. The numbers represent Red, Green, and Blue, and they must be between 0 and 255 inclusive:

```
surf1.fill((0,255,0)) # green-colored Surface
surf1.fill((0,0,255)) # blue-colored Surface
```

Combining the three numbers gives you different colors:

```
surf1.fill((128,64,0)) # brown-colored Surface
```

The Display (i.e.: the game window) also has a Surface, and anything that gets drawn to this Surface will show up on your screen. We can retrieve the Display's Surface by setting its resolution or by using get_surface().

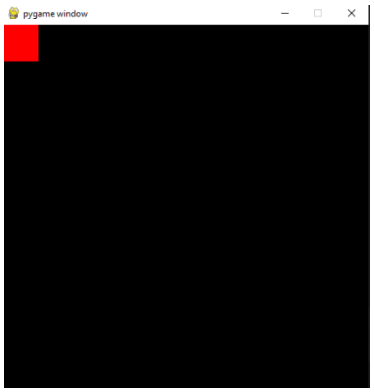
```
screen = pygame.display.set_mode((500,500))
screen = pygame.dispaly.get_surface()
```

Finally, we can draw our Surface to the display using the blit() method. The blit() method needs at least two parameters, which are the Surface being drawn and the coordinates where that surface should be drawn to:

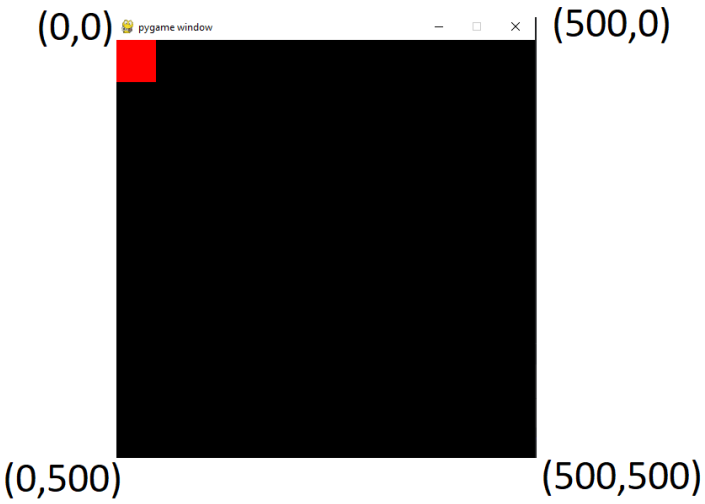
```
screen.blit(surf1, (0,0))
pygame.display.flip() # the display must be updated after the blit
```

Keep in mind that, because we are painting the screen black every time our gameplay loop runs, we need to draw the red Surface onto the Display's Surface **after** we fill the Display's Surface with black. Otherwise, we'll draw the red square and then paint over it with black!

If you did everything correctly, you should get the following:



A final note on coordinates: In computer graphics, **the origin is at the top left**, with the x-axis increasing to the right and the y-axis increasing down:

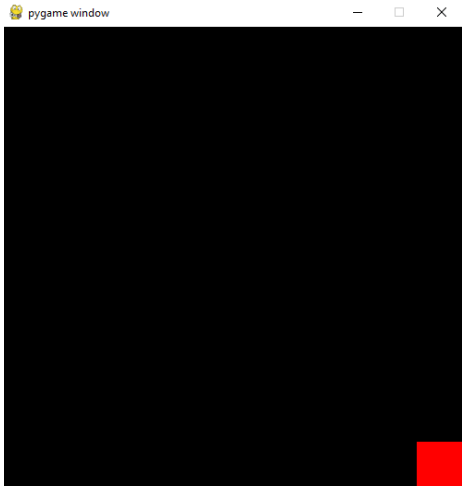


Thus, if we wanted to put the red square at the bottom right of the screen, we'd have to blit it at coordinates (450,450):

```
screen.blit(surf1, (450,450))
```

- The square's origin is at its top left, so wherever we draw it on the screen, it will start being drawn at the top left and then expand right up to its width and down up to its height.
- If we drew the square at (500,500), which is the edge of the screen, the square would be drawn outside the screen.

- We need to consider the square's width and height when determining where to draw it. And since its height and width are 50x50, we subtract those from the screen's dimensions to get the coordinates where we need to paint the red square (500 – 50 = 450).



Pygame Rects:

Rects, short for Rectangles, are a very useful class in Pygame to keep track of spatial information. Specifically, Rects are used to keep track of where a rectangular area is on the screen. We can create one as such:

```
rect1 = pygame.Rect(0,0, 50,50)
```

Rects need 4 parameters, their **x** coordinates, their **y** coordinates, their **width**, and **height**. Thus, our Rect above would be 50x50, and would be at the top left of our screen. Notice, however, that simply creating a Rect does not draw it to the Display. Indeed, a Rect cannot be blitted to the Display; **only Surfaces can be blitted onto the Display.**

Rects should, instead, be used as a convenience to keep track of where a Surface should be blitted.

```
rect1 = pygame.Rect(0,0,50,50)
surf1 = pygame.Surface((rect1.width, rect1.height))
screen = pygame.display.set_mode((500,500))
screen.blit(surf1, (rect1.x, rect1.y))
pygame.display.flip()
```

Doing the above (creating a Rect, creating a Surface based on a Rect, then blitting the Surface onto the Display using the Rect's coordinates) might seem like a lot of extra work for now, but it will considerably simplify moving our Surfaces later on.

Rects feature many fields to simplify getting their coordinates and measurements:

```
x,y
top, left, bottom, right
topleft, bottomleft, topright, bottomright
midtop, midleft, midbottom, midright
center, centerx, century
```

```
size, width, height
w,h
```

If you need to figure out where the bottom-right corner of a Rect is, rather than doing:

```
rect_coordinates = (rect1.x + width, rect1.y + height)
```

We can instead use the appropriate field:

```
rect_coordinates = rect1.bottomright
```

All fields are updated automatically for you whenever you edit one of the fields:

```
rect1 = pygame.Rect(0,0,50,50)
print(rect1.topright) # prints (50,0)
rect1.x = 10 # changes the x coordinate
rect1.y = 5 # changes the y coordinate
print(rect1.topright) # prints(60, 5)
rect1.w = 100 # changes the width
print(rect1.topright) # prints(110, 5)
```

Prefer using Rects going forward, whenever you want to store information on a rectangular area. Also, notice that some of the Rect fields are pairs of numbers (e.g.: size), while others are single numbers (e.g.: width).

Rect Movement:

To achieve the illusion of movement, we need to update a Rect's position on the screen, and then blit the Surface associated with it to the new position. We can update the Rect's position by using its move() method, which takes in two numbers as its parameters. These numbers represent where you wish to move the Rect, with the numbers representing the x-axis and the y-axis, respectively. **Passing in negative numbers updates the Rect's position in the opposite direction**.

Notice that move() returns a new Rect. **You need to save this new Rect, as it is this new Rect that is in the new position**.

```
rect1 = pygame.Rect(0,0,50,50)
rect1 = rect1.move(5,5) # creates a new Rect at coordinates (5,5)
rect1 = rect1.move(5,5) # creates a new Rect at coordinates (10,10)
```

We then need to blit the Surface associated with this Rect to its new position on the screen.

```
surf1 = pygame.Surface((50,50))
surf1.fill((255,0,0))
screen.blit(surf1, rect1.topleft)
pygame.display.flip()
```

The code above will simply paint the Surface at (10,10) of the Display, but it will not move around. If we want to achieve the illusion of movement, **we must update the Rect's position every time the main gameplay loop runs**:

```
import pygame, sys
from pygame.locals import *

pygame.init()

resolution = (500, 500)

screen = pygame.display.set_mode(resolution)

surf1 = pygame.Surface((50,50))
surf1.fill((255,0,0))
rect1 = pygame.Rect(0,0,50,50)

clock = pygame.time.Clock()

# Main gameplay loop
while True:
    keys = pygame.key.get_pressed()
      for event in pygame.event.get():
            if event.type == QUIT:
                  sys.exit(0)
            if keys[pygame.K_ESCAPE]:
                  sys.exit(0)

      screen.fill(color=(0,0,0))

      # update the Rect's position every time the loop iterates
      rect1 = rect1.move(5,5)

      # paint the Surface onto the Display's Surface at the new position
      screen.blit(surf1, rect1.topleft)

      pygame.display.flip()

      clock.tick(60)
```

Notice that we must blit the Surface onto the Display **after** we've filled the Display with the color black but **before** we update the Display's contents. Changing the order in which these operations are done (or omitting the screen.fill()) will lead to completely different results.
Notice also that our square goes off outside the Display. If we wanted to keep it in the Display, we'd need to use an IF statement to check if the square reached the end of the Display, and then stop calling move().

As with previous weeks, all labs should have the appropriate file names:

- Lab11A.py
- Lab11B.py
- Lab11C.py

Lastly, make sure you review the sample output and make sure the output of your program follows the exact same format shown in the Demo Videos.

Review the DEMO video included in the compressed zip file

Lab11A: Fade Background

Build a simple pygame app that starts as a black screen then slowly fades to white and then slowly fade back to black.

Requirements:

- The window size should be 400 x 400
- The background color should start black.
- The background color starts fading into white and then back to black again.
- The program should keep changing the background color until it is terminated.
- The program should stop when the user presses ESCAPE.
- The program should stop when the user closes the window.

Hint:

- In RGB, black is 0, 0, 0 and white is 255, 255, 255.

Lab11B: Five Squares

Build a simple pygame app that contains five red squares. The squares should be positioned on each corner plus one in the center.

Requirements:

- The window size should be 600 x 600
- The background color should be black.
- The size of each square should be 60 x 60
- The color of each square should be Red
- You must use pygame's Surfaces.
- The application should have five red squares, one of each corner and one in the center.
- The program should stop when the user presses ESCAPE.
- The program should stop when the user closes the window.

Hint:

- Remember to account for the surface's width and height when specifying the coordinate of each square

Lab11C: Moving Squares

Build a simple pygame app that contains two squares that move back-and-forth on the X axis.

Requirements:

- The window size should be 1000 x 500.
- The background color should be black.
- The application should have two squares, on the top edge and another on the bottom edge.
- The top edge square should be colored Green

- The bottom edge square should be colored Blue
- Both squares should have a size of 100 x 100
- Both squares should start on the left edge of the window.
- Both squares should move with a constant speed of 5 pixels to the right.
- Once both squares' right side reaches the right edge, the squares should move back to the left edge of the window.
- The squares should be implemented as Rects to handle movement and positioning.
- You must exclusively implement the built-in Rect `move()` function to handle the square's movement.
- The program should stop when the user presses ESCAPE.
- The program should stop when the user closes the window.

Hint:

- You probably want to keep track of what direction the Rects are moving.

Submission Instructions:

- Programs must follow the output format provided in the Demo Videos.
- Programs must be working correctly.
- Programs must be written in Python.
- Programs must be implemented exclusively using Pygame
- Programs must be submitted with the correct **.py** format.
- Programs must be saved in files with the correct file name:
    - Lab11A.py
    - Lab11B.py
    - Lab11C.py
- Programs (source code files) must be uploaded to Gradescope by the due date.