Chapter 11: Getting Started with Pygame

Contents

Time required: 90 minutes

DRY

Don't **R**epeat **Y**ourself

Code Shape

Please group program code as follows.

- Declare constants and variables
- Get input
- Calculate
- Display

Visualize and Debug Programs

The website www.pythontutor.com helps you create visualizations for the code in all the listings and step through those programs one instruction at a time. As you do that, pay attention to the movements of the red and green arrows on the left, the diagram on the right, and the printed material at the bottom. This will help you to better understand the behavior of the programs we are working on.

This is a great way to debug your code. You can see the variables change as you step through the program.

www.pythontutor.com

What is Pygame?

The Pygame library is the most well-known python library for making games. It's not the most advanced or high-level library. It is simple and easy to learn (comparatively). Pygame serves as a great entry point into the world of graphics and game development.

Pygame is installed with Python. Pygame is a framework that includes several modules with functions for drawing graphics, playing sounds, handling mouse input, and other game like things. Pygame provides functions for creating programs with a **graphical user interface,** or **GUI** (pronounced, "gooey").

Install and Update Pygame with pip

pip is a package manager for **Python**. It's a tool that allows you to install and manage additional libraries and dependencies that are not distributed as part of the standard library.

Install Pygame:

```
pip install pygame
```

If you are using Pygame on a regular basis, you will want to Update Pygame.

```
pip install pygame -U
```

You are now ready to embark on your successful and prosperous game development career in Python!

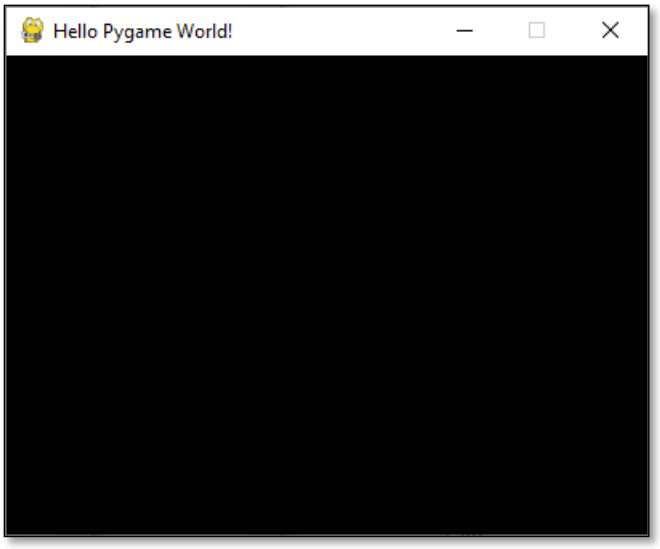
Tutorial 1: Hello Pygame

Yet another Hello, World program. This time we do it in Pygame.

Type in the following code and save the file as **hello_pygame.py**.

```
"""
    Name: hello_pygame.py
    Author:
    Date:
    Purpose: Create a blank Pygame window
"""
# Import pygame libraries
import pygame
from pygame.locals import *
# Import sys.exit to cleanly exit program
import sys

#--------------- INITIALIZE PYGAME ---------------#
# Initialize the pygame module
pygame.init()
# Create display window
SCREEN = pygame.display.set_mode((400, 300))
# Set caption for window
pygame.display.set_caption("Hello Pygame World!")

#--------------- INFINITE GAME LOOP --------------#
while True:
    # Listen for program events
    for event in pygame.event.get():

        # Closing the program window
        # causes the QUIT event to be fired
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

    # Redraw the screen
    pygame.display.update()
```

Example run:



Yay! You've just made the world's most boring video game! It's just a blank window with Hello World! at the top of the window in the title bar.

Creating a window is the first step to making graphical games.

Pong

Imagine the year is 1973. All the talk is about a new arcade game which has been released by Atari. That game is called Pong.

It might not seem much of a game by today's standards. It was a massive hit in its day.

Don't be deceived, although a simple game, Pong covers a wide range of aspects of computer game programming. There is movement, control, collision detection, scoring, artificial intelligence. It's all in there!

Being able to program Pong is a doorway to being able to program a lot of other games.

Watch out! Once you start playing Pong you might find less time to program, as it is quite addictive!

We aren't going to do a complete Pong tutorial. We will get started with drawing and bouncing a ball around.

Tutorial 2: Bouncing Ball 1

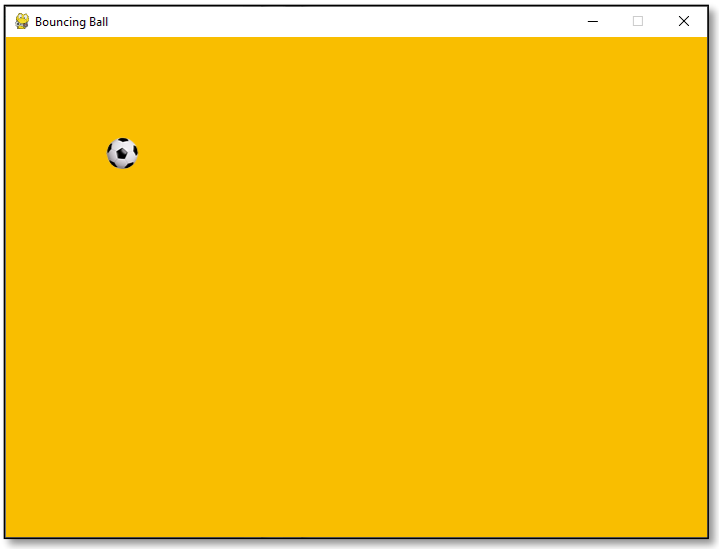
Download the **soccer_ball.png** file attached to this assignment to the folder you are creating this program in.

Create a Python file named **bouncing_ball_1.py** Add the following code.

```
"""
    Name: bouncing_ball_1.py
    Author:
    Date:
    Purpose: Draw a stationary ball
"""

# Import pygame libraries
import pygame

COUGAR_GOLD = (249, 190, 0)
#------------------- INITIALIZE PYGAME ----------------------#
# Initialize pygame module for action
pygame.init()

# Create a game surface
# width x and height y are set as a tuple
SURFACE = pygame.display.set_mode((700, 500))

# Load the ball image from the file system into a variable
ball = pygame.image.load("./soccer_ball.png")

# Create a rectangle the same size as the ball
# rect is used to set the location of the ball
ball_rect = ball.get_rect()

# Set initial postion of the ball rectangle
# x/left = 100, y/top = 100
ball_rect.left = 100
ball_rect.top = 100

# Set caption for window
pygame.display.set_caption("Bouncing Ball")
```

```
#--------------------- INFINITE GAME LOOP -------------------#
while True:
    """
        Infinite game loop
    """
    #------------- Draw on the surface/backbuffer -----------#
    # Fill the display surface
    # Clears the previous screen
    SURFACE.fill(COUGAR_GOLD)

    # Draw the ball on the surface
    SURFACE.blit(
        ball,       # Image to draw
        ball_rect   # Location to draw the image
    )

    #-------- Copy the backbuffer into video memory ---------#
    pygame.display.flip()
```

Example run:



Pygame Core Concepts

Let's start by explaining several core concepts related to the Pygame library and about creating games in general. Many of these concepts are transferable skills. If you switch to a more advanced game engine, many of these concepts will still hold true.

```
import pygame
```

The above code imports **pygame** and its modules into the program.

```
pygame.init()
```

This line is necessary whenever you want to use the Pygame library. Think of it as starting the pygame engine. It must be added before any other **pygame** function, or else an initialization error may occur.

```
# Create a display surface referenced by the surface reference variable
SURFACE = pygame.display.set_mode((700, 500))
```

This is a call to the **pygame.display.set_mode()** function, which returns the **pygame.Surface** object for the window. (Surface objects are described later.) Notice that we pass a tuple value of two integers to the function: **(700, 500)**. This tuple tells the **set_mode()** function how wide (x) and how high (y) to make the window in pixels. **(700, 500)** will make a window with a width of 700 pixels and height of 500 pixels.

An important aspect in games is individually accessing coordinates. To show a set of co-ordinates, you place both the X and Y values in a tuple, where the first integer is X and second integer is Y.

Both values start from the top-left hand side. X values increase from left to right, and Y values increase from top to bottom.

The coordinate values must be integers. A pixel is meant to represent the smallest possible area on a screen. There is no such thing as "half a pixel".

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

```
# Sets the title bar caption
pygame.display.set_caption("Bouncing Ball")
```

This sets the caption text that will appear at the top of the window by calling the **pygame.display.set_caption()** function. The string value 'Bouncing Ball' is passed in this function call to make that text appear as the caption of the window.

Game Loops and Game States

Every game that has what is called a "Game Loop". The game loop runs forever until the game is closed. All games have some form of a game loop.

The Game Loop is where all the game events occur, update, and get drawn to the screen. Once the initial setup and initialization of variables is out of the way, the Game Loop begins where the program keeps looping over and over until an event of type **QUIT** occurs.

Changes in the game are not implemented until the **pygame.display.update()** has been called. Since games are constantly changing values, the update function is in the game loop, constantly updating.

```
#Game loop runs forever
while True:
    # Code
    # Code

    pygame.display.update()
```

This 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).

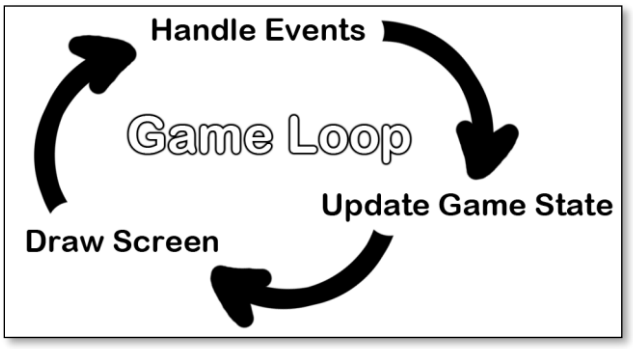
A game loop (also called a main loop) is a loop where the code does three things:

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 save, 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.



Quitting the Game Loop

Every game loop must have an end point, or some action that triggers the end point (such as clicking the quit button), or else your game will run indefinitely.

```
while True:
    pygame.display.update()
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()
```

We call both **pygame.quit()** and **sys.exit()** to close the pygame window and the python script respectively.

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.

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..

We have no if statements that run code for other types of Event objects. There isn't any 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 is done handling all the Event objects that have been returned by **pygame.event.get()**, the program execution continues to these final lines. We will add more event handling as we grow our program.

```
# Redraw the surface object
pygame.display.update()
```

This line 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 **window** 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 image is redrawn to the screen each time **pygame.display.update()** is called.

That is the entire program. After this line is done, the infinite while loop starts again from the beginning. This program does nothing besides make a window appear on the screen, constantly check for a **QUIT** event, and then redraws the unchanged window to the screen over and over and over again.

Event Objects

Any time the user does one of several actions 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.

```
# Watch for any program events
for event in pygame.event.get():

    # Closing the program by clicking the X
    # causes the QUIT event to be fired
    if event.type == pygame.QUIT:

        # Quit Pygame
        pygame.quit()

        # Stop Python script
        sys.exit()
```

This 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.

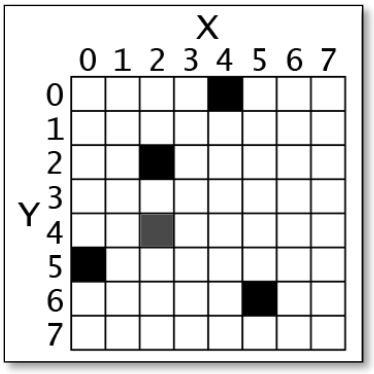
An "Event" occurs when the user performs a specific action, such as clicking their mouse or pressing a keyboard button. Pygame records every event that occurs.

One of the many attributes (or properties) held by event objects is type. The type attribute tells us what kind of event the object represents.

If you take a look at the example shown earlier, you'll see we used **event.type == QUIT** to determine whether the game is to be closed or not.

Pixel Coordinates

The window that the "Bouncing Ball" program creates is composed of little square dots on the screen called pixels. Each pixel starts off as black but can be set to a different color. Imagine that instead of a Surface object that is 300 pixels wide and 300 pixels tall, we just had a Surface object that was 8 pixels by 8 pixels. If that tiny 8x8 Surface was enlarged so that each pixel looks like a square in a grid, and we added numbers for the X and Y axis, then a good representation of it could look something like this:



We can refer to a specific pixel by using a Cartesian Coordinate system. Each column of the X-axis and each row of the Y-axis will have an "address" that is an integer from 0 to 7 so that we can locate any pixel by specifying the X and Y axis integers.

For example, in the above 8x8 image, we can see that the pixels at the XY coordinates (4, 0), (2, 2), (0, 5), and (5, 6) have been painted black, the pixel at (2, 4) has been painted gray, while all the other pixels are painted white. XY coordinates are also called points. If you've taken a math class and learned about Cartesian Coordinates, you might notice that the Y-axis starts at 0 at the top and then increases going down, rather than increasing as it goes up. This is just how Cartesian Coordinates work in Pygame (and almost every programming language).

The Pygame framework often represents Cartesian Coordinates as a tuple of two integers, such as (4, 0) or (2, 2). The first integer is the X coordinate and the second is the Y coordinate.

Colors

Colors are a big part of any game development framework or engine.

Pygame uses the RGB system of colors. This stand for Red, Green and Blue respectively. These three colors combined (in varying ratios) are used to create all the colors you see on computers, or any device that has a screen.

Some of these colors are predefined in Pygame. See the **pygame_color_picker.py** program attached to this assignment.

There are three primary colors of light: red, green and blue. (Red, blue, and yellow are the primary colors for paints and pigments, but the computer monitor uses light, not paint.) By combining different amounts of these three colors you can form any other color.

In Pygame, we represent colors with tuples of three integers. The first value in the tuple is how much red is in the color. An integer value of 0 means there is no red in this color, and a value of 255 means there is the maximum amount of red in the color. The second value is for green and the third value is for blue. These tuples of three integers used to represent a color are often called RGB values.

Because you can use any combination of 0 to 255 for each of the three primary colors, this means Pygame can draw 16,777,216 different colors (that is, 256 x 256 x 256 colors).

For example, we will create the tuple (0, 0, 0) and store it in a variable named BLACK. With no amount of red, green, or blue, the resulting color is completely black. The color black is the absence of any color. The tuple (255, 255, 255) for a maximum amount of red, green, and blue to result in white. The color white is the full combination of red, green, and blue. The tuple (255, 0, 0) represents the maximum amount of red but no amount of green and blue, so the resulting color is red. Similarly, (0, 255, 0) is green and (0, 0, 255) is blue.

The values for each color range from 0 – 255, a total of 256 values. You can find the total number of possible color combinations by evaluating 256 x 256 x 256, which results in a value well over 16 million.

You can mix the amount of red, green, and blue to form other colors. Here are the RGB values for a few common colors:

Color	RGB Values
Aqua	(0, 255, 255)
Black	(0, 0, 0)
Blue	(0, 0, 255)
Fuchsia	(255, 0, 255)
Gray	(128, 128, 128)
Green	(0, 128, 0)
Lime	(0, 255, 0)
Maroon	(128, 0, 0)
Navy Blue	(0, 0, 128)
Olive	(128, 128, 0)
Purple	(128, 0, 128)
Red	(255, 0, 0)
Silver	(192, 192, 192)
Teal	(0, 128, 128)
White	(255, 255, 255)
Yellow	(255, 255, 0)

To use colors in Pygame, we first create Color objects using RGB values. RGB values must be in a tuple format, with three values, each corresponding to a respective color. Here are some examples below showing some color objects in pygame.

```
BLACK = pygame.Color(0, 0, 0)          # Black
WHITE = pygame.Color(255, 255, 255)    # White
GREY = pygame.Color(128, 128, 128)     # Grey
RED = pygame.Color(255, 0, 0)          # Red
HUSKER_RED = (227, 25, 55)
HUSKER_CREAM = (253, 242, 217)
COUGAR_BLUE = (0, 58, 112)
COUGAR_GOLD = (249, 190, 0)
```

The **fill(color)** method is used to fill in objects. For instance, assigning a rectangle the color green will only turn the borders green. If you use the **fill()** method and pass a green color object, the rectangle will become completely green.

FPS (Frames per Second)

Computers are extremely fast and can complete millions of loop cycles in under a second. Obviously, this is a bit fast for us humans. As reference, movies are run at 24 frames per second. Anything less than that will have obvious stutter to it, and values over 100 may cause the things to move too fast for us to see.

The frame rate or refresh rate is the number of pictures that the program draws per second. It is measured in FPS or frames per second. (On computer monitors, the common name for FPS is hertz. Many monitors have a frame rate of 60 hertz, or 60 frames per second.) A low frame rate in video games can make the game look choppy or jumpy. If the program has too much code to run to draw to the screen frequently enough, then the FPS goes down.

If we do not create a limitation, the computer will execute the game loop as many times as in can within a second. To limit it we use the **tick(fps)** method where **fps** is an integer. The **tick()** method belongs to the **pygame.time.Clock** class and must be used with an object of this class.

```
CLOCK = pygame.time.Clock()
CLOCK.tick(60)
```

The frames per second can vary from game to game, depending on how it was designed. A good value to aim for is between 30 – 60. Keep in mind, that if you create a rather complex and heavy game the computer might not be able to run it well at higher frames.

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.

```
CLOCK = 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.)

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()`.

```
CLOCK.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.

Screen double-buffering

The image you see on your screen is stored in memory as a pixel buffer. This memory area, called the video memory, stores the color of every pixel on the screen.

We don't draw directly to the video memory. That is because the drawing process involves clearing the memory, then drawing the individual sprites, one by one. This would cause flickering if we did it in the video memory.

Instead, we use a second memory buffer that is just ordinary memory (it doesn't affect the display). This is sometimes called the back buffer. We draw our sprites into that buffer. When we have finished drawing and every pixel is set to its final color, we call flip, which copies that memory buffer into the actual video memory. This is called double-buffering.

You don't need to worry about this, Pygame sorts most of it out. You just need to remember:

1. Draw all your sprites using the screen object (this represents the back buffer).
2. When all the drawing is complete, call pygame.display.flip() to copy the complete buffer into video memory.

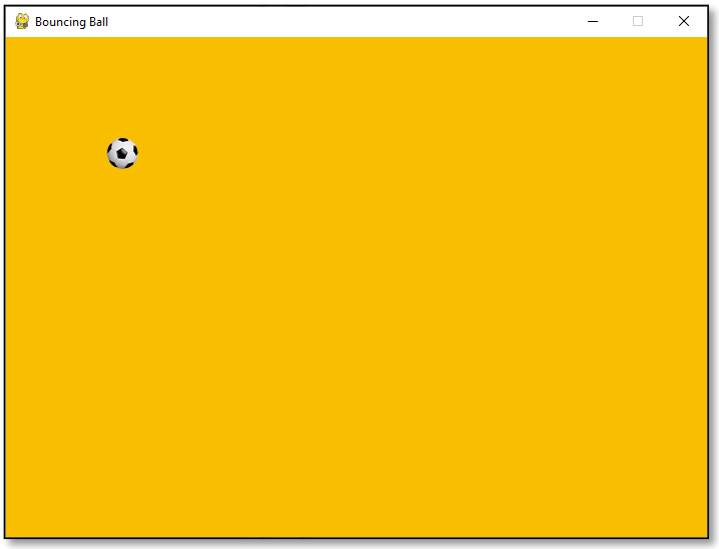
Tutorial 3: Bouncing Ball 2

Time to move our ball. Open **bouncing_ball_1.py** Save as **bouncing_ball_2.py** Add the following code. There are only a few changes.

```
"""
    Name: bouncing_ball_2.py
    Author:
    Date:
    Purpose: The ball moves across the screen
"""

# Import pygame library
import pygame
# Import sys.exit to cleanly exit program
from sys import exit

COUGAR_GOLD = (249, 190, 0)
#-------------------- INITIALIZE PYGAME ----------------------#
# Initialize pygame for action
pygame.init()

# Set screen width x and height y as a tuple
SURFACE = pygame.display.set_mode((700, 500))

# Load the ball image from the file system
ball = pygame.image.load("./soccer_ball.png")

# Create a rectangle the same size as the ball
# rect is used to set the location of the ball
ball_rect = ball.get_rect()

# Set initial postion of the ball rectangle
# x/left = 100, y/top = 100
ball_rect.left = 100
ball_rect.top = 100

# Set caption for window
pygame.display.set_caption("Bouncing Ball")

# The CLOCK object manages how fast the game runs
CLOCK = pygame.time.Clock()
```

```
#--------------------- INFINITE GAME LOOP -------------------#
while True:
    """
        Infinite game loop
    """
    # Listen for program events
    for event in pygame.event.get():

        # Closing the program window
        # causes the QUIT event to be fired
        if event.type == pygame.QUIT:
            # Quit Pygame
            pygame.quit()
            # Exit Python
            exit()

    #------------- Draw on the surface/backbuffer -----------#
    # Fill the display surface
    # Clears the previous screen
    # Comment out this line to see why is is necessary
    SURFACE.fill(COUGAR_GOLD)

    # Draw the ball on the surface
    SURFACE.blit(
        ball,       # Image to draw
        ball_rect   # Location to draw the image
    )

    # Move ball down and to the right 2 pixels at a time
    # x = 2, y = 2
    ball_rect = ball_rect.move(2, 2)

    #-------- Copy the backbuffer into video memory ---------#
    # Redraw the surface object
    pygame.display.flip()

    # Regulates how often the game loop executes
    # 60 FPS (frames per second)
    CLOCK.tick(60)
```

When this program is run, the following window is displayed until the user closes the window. The ball keeps moving until it disappears off the screen.



You did it again! You created the world's second most boring game. You moved a ball across and out of the screen.

You can't tell by the screenshot. Our ball is moving down and to the right 2 pixels a frame. Our Frames per Second is 60; the ball is moving 120 pixels a second.

Notice that we are using **surface.fill("goldenrod1")** to clear the screen each time through the game loop. Comment out that line to see why.

Rect Object

Instead of drawing the ball directly to the screen, we create a Rect object that we draw the ball in.

The code below creates a Rect object that is the same size as the ball..

```
ball = pygame.image.load("ball_sports.png")
ballrect = ball.get_rect()
```

The handy thing about this is that the Rect object automatically calculates the coordinates for other features of the rectangle. Rect makes moving other objects easier to find and move around the game screen.

If you need to know the X coordinate of the right edge of the pygame.Rect object we stored in the spamRect variable, you access the Rect object's right attribute:

```
ballrect.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. If we used drawing objects, we would have to calculate this manually.

```
ballrect.right = 350
ballrect.left
150
```

Here's a list of all the attributes that pygame.Rect objects provide:

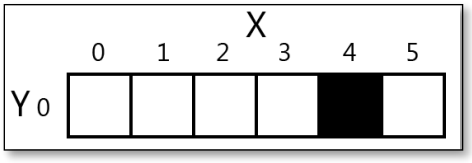
Attribute Name	**Description**
myRect.left	The int value of the X-coordinate of the left side of the rectangle.
myRect.right	The int value of the X-coordinate of the right side of the rectangle.
myRect.top	The int value of the Y-coordinate of the top side of the rectangle.
myRect.bottom	The int value of the Y-coordinate of the bottom side.
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)

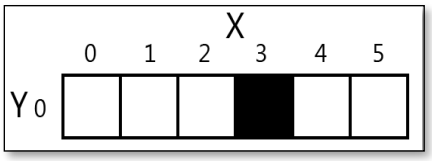
Animation

Now that we know how to get the Pygame framework to draw to the screen, let's learn how to make animated pictures. A game with only still, unmoving images is dull. (Sales of the game "Look At This Rock" have been disappointing.)

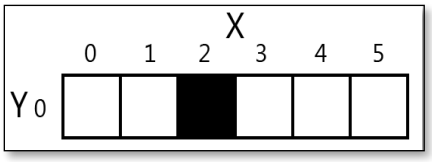
Animated images are the result of drawing an image on the screen, then a split second later drawing a slightly different image on the screen. Imagine the program's window was 6 pixels wide and 1 pixel tall, with all the pixels white except for a black pixel at 4, 0. It would look like this:



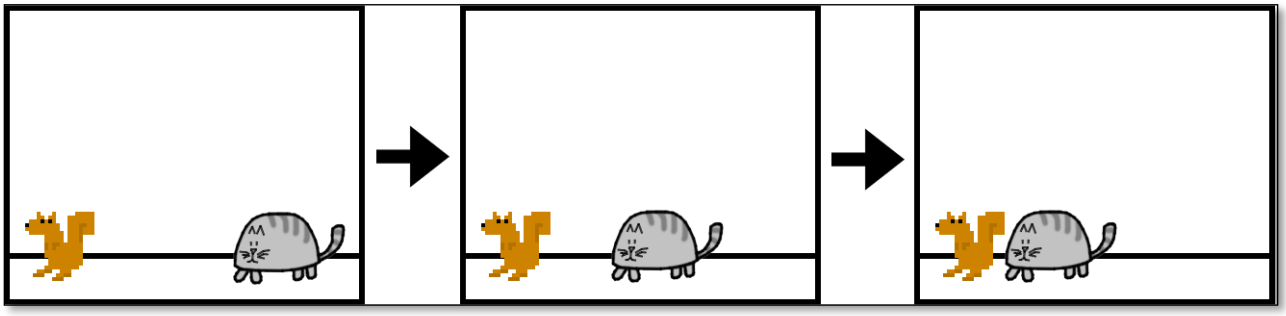
If you changed the image so that 3, 0 was black and 4, 0 was white, it would look like this:



To the user, it looks like the black pixel has "moved" over to the left. If you redrew the window to have the black pixel at 2, 0, it would continue to look like the black pixel is moving left:



It may look like the black pixel is moving, but this is just an illusion. To the computer, it is just showing three different images that each just happen to have one black pixel. Consider if the three following images were rapidly shown on the screen:



To the user, it would look like the cat is moving towards the squirrel. But to the computer, they're just a bunch of pixels. The trick to making believable looking animation is to have your program draw a picture to the window, wait a fraction of a second, and then draw a slightly different picture.

Tutorial 4: Bouncing Ball 3

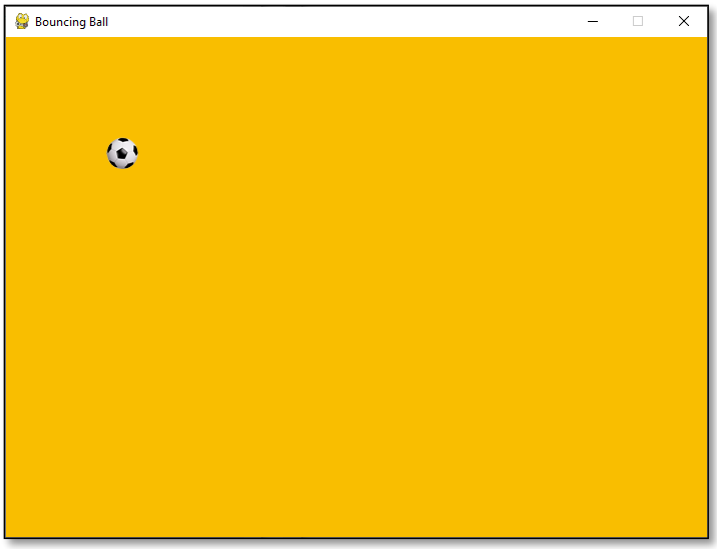
The grand finale!!! Yes, we are finally going to make something move. By changing the (x, y) values each time through the game loop, we animate our ball.

Open **bouncing_ball_2.py** Save as **bouncing_ball_3.py** Change the code to the following:

```
"""
    Name: bouncing_ball_3.py
    Author:
    Date:
    Purpose: Bounce a ball around the screen
"""

# Import pygame library
import pygame
# Import sys.exit to cleanly exit program
from sys import exit

COUGAR_GOLD = (249, 190, 0)
#------------------- INITIALIZE PYGAME ----------------------#
# Set screensize constants
WIDTH = 700
HEIGHT = 500

# Initialize pygame for action
pygame.init()

# Create a game surface
# width x and height y are set as a tuple
SURFACE = pygame.display.set_mode((WIDTH, HEIGHT))

# Load the ball image from the file system
ball = pygame.image.load("./soccer_ball.png")

# Create a rectangle the same size as the ball
# rect is used to set the location of the ball
ball_rect = ball.get_rect()

# Set initial postion of the ball rectangle
# x/left = 100, y/top = 100
ball_rect.left = 100
ball_rect.top = 100

# Speed as an x, y list
speed = [2, 2]

# Set caption for window
pygame.display.set_caption("Bouncing Ball")

# CLOCK object manages how fast the game runs
CLOCK = pygame.time.Clock()
```

```
#--------------------- INFINITE GAME LOOP -------------------#
while True:
    """
        Infinite game loop
    """
    # Listen for program events
    for event in pygame.event.get():

        # Closing the program window
        # causes the QUIT event to be fired
        if event.type == pygame.QUIT:
            # Quit Pygame
            pygame.quit()
            # Exit Python
            exit()

    #------------- Draw on the surface/backbuffer -----------#
    # Fill the display surface
    # Clears the previous screen
    # Comment out this line to see why is is necessary
    SURFACE.fill(COUGAR_GOLD)

    # Draw the ball on the backbuffer
    SURFACE.blit(
        ball,       # Image to draw
        ball_rect   # Location to draw the image
    )

    # Check for collision with left or right wall
    if ball_rect.left < 0 or ball_rect.right > WIDTH:
        # Reverse x direction
        speed[0] = -speed[0]

    # Check for collision with top or bottom wall
    if ball_rect.top < 0 or ball_rect.bottom > HEIGHT:
        # Reverse y direction
        speed[1] = -speed[1]

    # Move ball at speed list
    # x = speed[0], y = speed[1]
    ball_rect = ball_rect.move(speed)

    #-------- Copy the backbuffer into video memory ---------#
    # Copy the backbuffer into video memory
    pygame.display.flip()

    # Regulates how often the game loop executes
    # 60 FPS (frames per second)
    CLOCK.tick(60)
```

Example run:

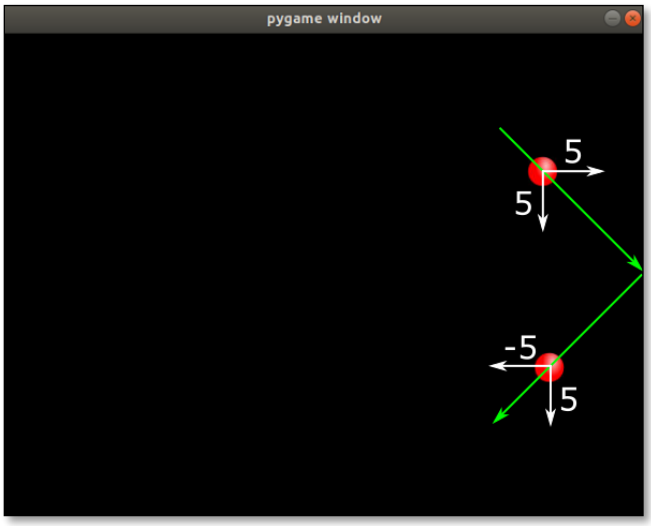


Yes, the screenshot looks the same. It is bouncing off the edges of the screen. Stare at the image long enough, you will see it start to move.

```
# Check for collision with left or right wall
if ballrect.left < 0 or ballrect.right > WIDTH:
    # Reverse x direction
    speed[0] = -speed[0]
```

If the ball's left edge moves past the left edge of the screen, the direction reverses by changing the speed to the negative of what it currently is. If the current speed is +2, when it is changed to a negative, it reverses direction. This works the same with each edge of the screen.

The image below shows how the x direction will reverse.



Tutorial 5: Mow the Lawn

As the grass gets longer, our soccer ball gets harder and harder for the players to find. Let's get our tractor out of the shed and mow the lawn.

```
"""
    Name: bouncing_ball_4.py
    Author:
    Date:
    Purpose: Move a tractor around the screen
    Turn the tractor in the direction of the movement
"""

# Import pygame libraries
import pygame
# Import sys.exit to cleanly exit program
from sys import exit

COUGAR_GOLD = (249, 190, 0)
#------------------- INITIALIZE PYGAME ----------------------#
# Set screensize constants
WIDTH = 700
HEIGHT = 500

# Initialize pygame for action
pygame.init()

# Create a game surface
# width x and height y are set as a tuple
SURFACE = pygame.display.set_mode((WIDTH, HEIGHT))

# Load the images from the file system
ball = pygame.image.load("./soccer_ball.png").convert_alpha()
tractor = pygame.image.load("./green_tractor.png").convert_alpha()

# Pre rotate 4 images for turning the tractor when the cursor keys are pressed
tractor_up = tractor
tractor_down = pygame.transform.rotate(tractor, 180)
tractor_right = pygame.transform.rotate(tractor, -90)
tractor_left = pygame.transform.rotate(tractor, 90)

# Create a rectangle the same size as the image
# rect is used to set the location of the image
ball_rect = ball.get_rect()
tractor_rect = tractor.get_rect()

# Set initial postion of the ball rectangle
# x/left = 100, y/top = 100
ball_rect.left = 100
ball_rect.top = 100
```

```
# Speed as an x, y list
speed = [2, 2]

# Set caption for window
pygame.display.set_caption("Mow the Lawn")

# CLOCK object manages how fast the game runs
CLOCK = pygame.time.Clock()

#------------------------- INFINITE GAME LOOP ----------------------#
while True:
    """
        Infinite game loop
    """
    # Listen for program events
    for event in pygame.event.get():

        # Closing the program window
        # causes the QUIT event to be fired
        if event.type == pygame.QUIT:
            # Quit Pygame
            pygame.quit()
            # Exit Python
            exit()

    #--------------------------- ROTATE TRACTOR ---------------------#
        # Turn the tractor the direction the cursor keys are pressed
        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_UP:
                x, y = tractor_rect.center
                tractor = tractor_up
                tractor_rect = tractor.get_rect(center=(x, y))
            elif event.key == pygame.K_DOWN:
                x, y = tractor_rect.center
                tractor = tractor_down
                tractor_rect = tractor.get_rect(center=(x, y))
            elif event.key == pygame.K_RIGHT:
                x, y = tractor_rect.center
                tractor = tractor_right
                tractor_rect = tractor.get_rect(center=(x, y))
            elif event.key == pygame.K_LEFT:
                x, y = tractor_rect.center
                tractor = tractor_left
                tractor_rect = tractor.get_rect(center=(x, y))

    #------------------ CAPTURE KEYPRESSED EVENTS -----------------#
    # Capture key pressed events into a list
    pressed_keys = pygame.key.get_pressed()
    # Move the tractor in the direction of the cursor keys
    # Decide which way to go, move_ip(x, y)
    if pressed_keys[pygame.K_UP]:
        tractor_rect.move_ip(0, -3)
    if pressed_keys[pygame.K_DOWN]:
        tractor_rect.move_ip(0, 3)
    if pressed_keys[pygame.K_LEFT]:
        tractor_rect.move_ip(-3, 0)
    if pressed_keys[pygame.K_RIGHT]:
        tractor_rect.move_ip(3, 0)
```

```
106     #------------------ DRAW ON THE BACKBUFFER ----------------------#
107     # Fill the display surface
108     # Clears the previous screen
109     # Comment out this line to see why is is necessary
110     SURFACE.fill(COUGAR_GOLD)
111
112     # Draw the ball on the backbuffer
113     SURFACE.blit(
114         ball,           # Image to draw
115         ball_rect       # Location to draw the image
116     )
117
118     # Draw the tractor on the backbuffer
119     SURFACE.blit(
120         tractor,        # Image to draw
121         tractor_rect    # Location to draw the image
122     )
123
124     # Check for collision with left or right wall
125     if ball_rect.left < 0 or ball_rect.right > WIDTH:
126         # Reverse x direction
127         speed[0] = -speed[0]
128
129     # Check for collision with top or bottom wall
130     if ball_rect.top < 0 or ball_rect.bottom > HEIGHT:
131         # Reverse y direction
132         speed[1] = -speed[1]
133
134     # Move ball at speed list
135     # x = speed[0], y = speed[1]
136     ball_rect = ball_rect.move(speed)
137
138     #--------- COPY THE BACKBUFFER INTO VIDEO MEMORY ------------------#
139     # Copy the backbuffer into video memory
140     pygame.display.flip()
141
142     # Regulates how often the game loop executes
143     # 60 FPS (frames per second)
144     CLOCK.tick(60)
```

Example run:

Using your cursor keys, you can drive the tractor around and mow the lawn.

Tutorial 6: Keep the Tractor on the Soccer Field

The tractor driver tends to drove off the field when they are mowing. Let's fence them in.

This is the only part that changed. The **tractor_rect** properties keep track of where the tractor is. If they move outside the surface boundary, the keys won't do anything.

```
65      #--------------- Capture key pressed events -------------#
66      pressed_keys = pygame.key.get_pressed()
67      # Is the tractor on the screen?
68      # These if statements prevent the tractor
69      # from going off of the screen
70      # Decide which way to go, move_ip(x, y)
71      if tractor_rect.top > 0:
72          if pressed_keys[pygame.K_UP]:
73              tractor_rect.move_ip(0, -3)
74      if tractor_rect.bottom < HEIGHT:
75          if pressed_keys[pygame.K_DOWN]:
76              tractor_rect.move_ip(0, 3)
77      if tractor_rect.left > 0:
78          if pressed_keys[pygame.K_LEFT]:
79              tractor_rect.move_ip(-3, 0)
80      if tractor_rect.right < WIDTH:
81          if pressed_keys[pygame.K_RIGHT]:
82              tractor_rect.move_ip(3, 0)
```

The Bouncing Ball game, though it is very simple, contains many of the mechanics and physics of more complex games. Time for you to see what you can create. Find some tutorials, start your game development career!

Assignment Submission

Attach all program files to the assignment in BlackBoard.