Kyle Forker
Dr. Wang
February 23, 2022
CS-489

Term 2 Summary

Starting off term 2 the creation of the first game was started. After learning the basic functionalities of PyGame through the game Snake, the second game, Flappy Bird, was started. While this game already exists it shows application of the research and understanding of PyGame. Since this project is limited to just over two months the construction from a unique game from the ground up is not very reasonable therefore, two, maybe three, games will be produced. These games will be replicas of current games, but coded off the knowledge and understanding of the research so far.

The game, Flappy Bird, has been finished and is currently working. The only thing left to fix is some finishing touches on the game. Overall, the game has shown the depth of the research done in term one and term two. This summary will go over the code that was used in the Flappy Bird game and have a walk-through of the main portions of it.

```
1   import pygame
2   import random
3
4   pygame.init()
5   displayWidth = 700
6   displayHeight = 900
7
8   display = pygame.display.set_mode((displayWidth, displayHeight))
9   pygame.display.set_caption("Flappy Bird")
10  clock = pygame.time.Clock()
```

This is the standard way to begin a PyGame project as discussed in the term one summary. The developer must import PyGame and then initialize it. Then the developer must set up the game window with a height and width.

For anyone who has ever played flappy bird they would know that the game appears as if the player were to be moving through pipes and progressing to the right. This is not the case and in fact the background and the obstacles move instead of the player. To make things organized in python and since PyGame can make use of object-oriented programming the developer can set up classes, often not used in python, to keep everything clean and readable. Below is a screenshot to view the class for the moving background element of the game. For more information and videos of the game working there will be links provided at the end of the summary as well as copious amounts of screenshots and video on the website as well.

```
29  ##########################################################################################
30  #<---------- Background Class ---------->
31  ##########################################################################################
32  class background:
33      def __init__(self):
34          self.bgImage = pygame.image.load('city_bg.png')
35          self.rectBg = self.bgImage.get_rect()
36
37          self.bgY = 0
38          self.bgX = 0
39          self.bgY2 = 0
40          self.bgX2 = self.rectBg.width
41
42          self.movingSpeed = 5
43
44      def moveBG(self):
45          self.bgX -= self.movingSpeed
46          self.bgX2 -= self.movingSpeed
47
48          if self.bgX <= -self.rectBg.width:
49              self.bgX = self.rectBg.width
50
51          if self.bgX2 <= -self.rectBg.width:
52              self.bgX2 = self.rectBg.width
53
54      def displayBG(self):
55          display.blit(self.bgImage, (self.bgX, self.bgY))
56          display.blit(self.bgImage, (self.bgX2, self.bgY2))
```

In this screenshot, the class is created. Inside this class holds all the functions that can execute when the class is called as well as on their own. The first function is the initialization function in which main variables go into. This sets up the foundation for the rest of the class. The use of "self" here is to be able to access all variables amongst all functions inside the class. This means that every function has access to all the variables that were initialized. The "moveBG" function allows for the background to scroll across the game window. This uses if statements to check the position of the background and move it to the left at a constant speed. Once it has reached the end of the screen, denoted by "-self.rectBG.width" the background will then redraw on the right side of the screen to simulate an infinite scroll in the game. The final function draws the assets to the screen when called.

The second class used inside the game is the "pipes" class. This controls the obstacles that the player must maneuver through to score points. These also move in the same fashion as the background except there has to be more of them than just one on the screen at once.

```
################################################################################
#<---------- Pipes Object Class ---------->
################################################################################
class pipes():
    def __init__(self):
        self.bottomPipeImage = pygame.image.load('pipe.png').convert_alpha()
        self.bottomPipeImage = pygame.transform.smoothscale(self.bottomPipeImage, (130, 700))
        self.topPipeImage = pygame.transform.flip(self.bottomPipeImage, False, True)
        self.rectTopPipe = self.topPipeImage.get_rect()
        self.rectBottomPipe = self.bottomPipeImage.get_rect()

        self.scoreBoxImage = pygame.image.load('score.png').convert_alpha()
        self.rectScoreBox = self.scoreBoxImage.get_rect()

        self.pScore = 0

        #<------ Top Pipe X Coords ------>
        self.topPipeX = 0
        self.topPipeX2 = displayWidth + 50

        #<------ Bottom Pipe X Coords ------>
        self.bottomPipeX = 0
        self.bottomPipeX2 = displayWidth + 50

        self.disApart = 150
        self.movingSpeed = 5

        #<------ Generate Y-Values for Pipes ------>
        self.height = random.randint(350,800)
        self.height2 = (displayHeight - self.height) + 100
```

In this screenshot there is the initialization function for the class once again. Here the variables used throughout the class are defined once again as well. This includes setting up the images and hitboxes for the obstacles. Since the obstacles, or pipes, in flappy bird have random heights there needs to be a random number generator to generate the heights of each of the pipes that are created. Next, the first height can be used for the bottom obstacle and the second one must use an equation to keep a consistent space between the top and bottom pipes for each set of obstacles.

```
    def movePipes(self):
        #<------ moving top pipe ------>
        self.topPipeX -= self.movingSpeed
        self.topPipeX2 -= self.movingSpeed

        if self.topPipeX <= -self.rectBottomPipe.width:
            self.topPipeX = displayWidth

        if self.topPipeX2 <= -self.rectBottomPipe.width:
            self.topPipeX2 = displayWidth

        #<------ moving bottom pipe ------>
        self.bottomPipeX -= self.movingSpeed
        self.bottomPipeX2 -= self.movingSpeed

        if self.bottomPipeX <= -self.rectBottomPipe.width:
            self.bottomPipeX = displayWidth

        if self.bottomPipeX2 <= -self.rectBottomPipe.width:
            self.bottomPipeX2 = displayWidth
```

Continuing with this screenshot, here is the function in which controls the movement of the pies across the screen. This is the same code that was used to move the background except it is implemented differently later in the program.

```
def spawnPipes(self):
    #<------ Render Pipes ------>
    display.blit(self.topPipeImage, (self.topPipeX2, -self.height2))
    display.blit(self.bottomPipeImage, (self.bottomPipeX2, self.height))

    #<------ Update Collision Boxes for Pipes ------>
    self.rectTopPipe = self.topPipeImage.get_rect(topleft=(self.topPipeX2, -self.height2))
    self.rectBottomPipe = self.bottomPipeImage.get_rect(topleft=(self.bottomPipeX2, self.height))

    #<------ Enable to Render HitBoxes ------>
    #pygame.draw.rect(display, (255,255,0), self.rectTopPipe)
    #pygame.draw.rect(display, (255,255,0), self.rectBottomPipe)

    #<------ Score Counter ------>
     self.rectScoreBox = pygame.Rect((self.rectTopPipe[0] + self.rectTopPipe.width) + 30, self.rectTopPipe[1] + 670, 1, 350)
    self.rectScoreBox = self.scoreBoxImage.get_rect(topleft=((self.rectTopPipe[0] + self.rectTopPipe.width + 30), self.rectTopPipe[1] + 670))
    #pygame.draw.rect(display, (0,255,0), self.rectScoreBox)
```

Next, not only do the pipes have to spawn but the hitboxes for the pipes need to be tied to their image counterparts. Included in this there is also a hitbox for the player to pass through to be able to count up the score. This is the last line of code that is not commented in this

screenshot. The commented lines of code allow the developer to view the hitboxes of the models in the game rather than just their images.

The final class to walk through is the player class.

```
##########################################################################################
#<---------- Player Class ---------->
##########################################################################################
class bird(pipes):

    def __init__(self):
        self.playerImage = pygame.image.load('bird.png').convert_alpha()
        self.playerImage = pygame.transform.smoothscale(self.playerImage, (70,45))
        self.playerRect = self.playerImage.get_rect(center=(100,400))
        self.pScore = 0
        self.loss = False

    def moveBird(self, val):
        self.playerRect[1] += val
        #print(self.playerRect.center)

    def spawnBird(self):
        display.blit(self.playerImage, self.playerRect)

        #<------ Enable to Render Player HitBox ------>
        pygame.draw.rect(display, (255,0,0), self.playerRect)
```

In this screenshot the class bird needs to inherit the pipes class. This allows them to communicate information between the classes. Once again, the initialization function comes first. The second function is the function that controls the players y coordinate in game. Since the player never actually moves left or right the developer only needs to work with the y coordinate. Later on in the game function the value of “val” is passed into this function telling the player to go up or down. The spawnBird function allows the bird to be drawn on screen alongside adding a hitbox to the player as well.

```
    def collisions(self, coords):
        topCoords = coords[0]
        bottomCoords = coords[1]
        topLoss = self.playerRect.colliderect(topCoords)
        bottomLoss = self.playerRect.colliderect(bottomCoords)

        #print(self.playerRect)

        if self.playerRect[1] < 0 or self.playerRect[1] > 900:
            topLoss = True

        if topLoss or bottomLoss:
            #message("You Lost Press Q to quit or R to replay.", lossFont, 0, 500, (255,0,0))
            self.loss = True

    def returnLoss(self):
        return self.loss

    def birdLoc(self):
        return self.playerRect

    def score(self, coords):
        scoreCoords = coords[2]
        scoreAdd = self.playerRect.colliderect(scoreCoords)
        if scoreAdd:
            self.pScore += 1

    def currScore(self):
        return self.pScore
```

The final parts of the bird class includes a function to calculate if the player has collided with an obstacle. This also checks if the player attempts to go out of bounds as well. The returnLoss and birdLoc functions allow for returning essential data for the game function to use later on in the code. The score function allows to detect the player colliding with the score hitbox. This then raises the score if the conditions are met. The final function allows for return data about the current score in the game. These return functions allow the classes to communicate directly with other parts of the python script whether that be functions outside the class or just variables in general.

There are 4 functions that are not inside of classes in this script. The first two deal with displaying messages on screen.

```
def message(msg, mType, x, y, color):
    msg = mType.render(msg, True, color)
    display.blit(msg, (x, y))
    pygame.display.update()

def dropShadowMessage(msg, mType, x, y, color):
    mesg = mType.render(msg, True, (0,0,0))
    display.blit(mesg, (x + 2, y + 2))
    mesg = mType.render(msg, True, color)
    display.blit(mesg, (x, y))
    pygame.display.update()
```

The first function allows to display messages on the screen and the second function allows for displaying text with a drop shadow effect on the screen.

The third function outside of the main classes is the scoreCount function.

```
def scoreCount(score):
    dropShadowMessage(str(score // 14), scoreFont, 320, 200, (255,255,255))
```

This takes in the score that is read from the previous return functions in the player class.

The following screenshots of code will display the entire function that is the game itself. This will show all the code that controls the interface of the game.

```
#############################################################################
#<---------- Game Function ---------->
#############################################################################
def game():
    pipesList = []
    count = 0
    BACKGROUND = background()
    BIRD = bird()
    PIPES = pipes()
    close = False
    start = True
    go = True
    while go:

        while close:
            message("You Lost Press Q to quit or R to replay.", lossFont, 0, 500, (255,0,0))

            for event in pygame.event.get():
                if event.type == pygame.KEYDOWN:
                    if event.key == pygame.K_q:
                        go = False
                        close = False
                    if event.key == pygame.K_r:
                        game()
```

In the first part there is the initialization of variables that will be used in the function. The use of the first while loop is for the basics of PyGame. PyGame runs inside a while loop. The second while loop makes sure that the game does not just close when the player loses, but displays a screen to allow the player to play again or choose the option to quit. Checking for keys being pressed are also done inside of for loops in PyGame. The code looks for an event that triggers an action and what type of even that may be; mouse click or keyboard press.

```
    while start:
        display.fill((0,0,0))
        message("Flappy Bird", generalFont, 150, 300, (255,255,255))
        message("Press Space To Start", lossFont, 200, 400, (255,255,255))
        for event in pygame.event.get():
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_SPACE:
                    start = False


    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            go = False
            pygame.quit()

        if event.type == pygame.KEYDOWN and (event.key == pygame.K_SPACE):
            BIRD.moveBird(-80)

    BACKGROUND.moveBG()
    BACKGROUND.displayBG()

    count += 1
    BIRD.spawnBird()
    BIRD.moveBird(6)
    #BIRD.returnLoss()
```

This third while loop controls the start of the game. This allows the user to not be instantly kicked into the game when it is started or after the player loses and chooses retry. Once again there is the use of the for loop to check for inputs. In the final if statement the code checks if the player has used the space bar. This then moves the player up -80 pixels if the space bar is pressed. This then feeds into the player class and is translated to the “val” variable. The functions within the background class are called to create the background. The moveBird function has 6 passed into it so the player will continuously fall if the player is not tapping the space bar.

```
        if BIRD.returnLoss():
            close = True

        if count > 58:
            new_pipes = pipes()
            pipesList.append(new_pipes)
            count = 0
            print("len =",len(pipesList))

        for pipe in pipesList:
            pipe.movePipes()
            pipe.spawnPipes()
            returnInfo = pipe.pCollide()
            BIRD.collisions(returnInfo)
            #pipe.scoreCollide(BIRD.birdLoc())
            #scoreCount(pipe.returnScore())
            BIRD.score(returnInfo)

        if len(pipesList) > 3:
            print("deleting",pipesList[:1])
            del pipesList[:1]

        #scoreCount(PIPES.returnScore())
        scoreCount(BIRD.currScore())

        pygame.display.update()
        clock.tick(32)

game()
```

Finally, if the bird loses the game will go to the close while loop. The for loop deals with creating obstacles and storing them in a list. This list then clears the obstacle that is off screen. To end, clock.tick is how fast the game operates and the display.update() allows PyGame to refresh the game window to draw all objects on the screen. The final call of the game function allows the script to run the function that holds all this code.

Overall the research that was learned and used in term 2 allowed for the creation of this game. This research allowed the ability to learn the use of functions and classes within PyGame

scripts. It also allowed for the knowledge of debugging and fixing bugs in the code for a PyGame game as well. This research has been very important for the project overall and has deepened the knowledge not only PyGame, but python as well.