pyGame Lecture – Game Example

Sushi Game

This will be a second example of a game which features the following:

1. Use of Classes
2. Multiple Files
3. Multiple Screens
4. Leaderboard

We will also inherit from pygame.sprite.Sprite.

I. Main Menu Screen

This is set up similar to the Fruit Game, where the user is given a screen with three options:

1. Play Game
2. Display the Leaderboard
3. Quit

This is controlled from the file sushigame.py, which has a single menu function that gets called. This function displays a screen, has three buttons with text and detects mouseclicks on each of the buttons.

This function will have two nested loops. One will run until the user chooses to quit, while the loop nested inside of it will execute the user's options (between the first two). On the main screen, we draw just the buttons and text, so that portion looks like this. For now, the code inside of the pygame event loop is omitted. Also, before the menu function, a few global variables are declared:

```
SCREEN_W = 1000
SCREEN_H = 600

# Initialize all imported pygame modules.
pygame.init()
# Create a custom caption that will be displayed on the top of your window of
your game.
pygame.display.set_caption('Sushi Game!')

# Create a Font object from the system fonts. I chose size 100.
font = pygame.font.SysFont(None, 45)
bigfont = pygame.font.SysFont(None, 60)

# Create an object to help track time
clock = pygame.time.Clock()

def menu():

    # Creates a screen with width and height.
    screen = pygame.display.set_mode((SCREEN_W, SCREEN_H))
```

```
# Created buttons. X, Y, Width, Height.
button_1 = pygame.Rect(300, 100, 400, 100)
button_2 = pygame.Rect(300, 250, 400, 100)
button_3 = pygame.Rect(300, 400, 400, 100)

# While true, so infinite loop.
while True:

    # Fill surface with a solid color, I chose black.
    screen.fill((0, 0, 0))

    # Get mouse position x and y.
    mx, my = pygame.mouse.get_pos()

    # Refers to the draw_text function I made above.
    # Requires text, font, color, surface, x, y.
    # This is how we draw text to screen. IMPORTANT!!
    sushifunctions.draw('Menu', bigfont, (255, 255, 255), screen, 450, 30)

    # Draws buttons onto screen
    pygame.draw.rect(screen, (192,192,192), button_1)
    pygame.draw.rect(screen, (147,112,219), button_2)
    pygame.draw.rect(screen, (0,200,200), button_3)

    # Creates and draws text on buttons.
    sushifunctions.draw('Play Game',font,(255, 255, 255), screen, 420, 135)
    sushifunctions.draw('Disp Leaders',font,(255,255,255),screen,350, 285)
    sushifunctions.draw('Quit', font, (255, 255, 255), screen, 470, 435)

    # Get events from the queue
    for event in pygame.event.get():

        # Event handling here.

    # Update portions of the screen for software displays
    pygame.display.update()

    # Update the clock in milliseconds. Should be called once per frame.
    clock.tick(60)
```

Recall that the draw function simply allows us to easily draw some text on the pyGame screen where we specify the text, font, color, screen and top left position.

Thus, this is basically just the visual display.

In the event loop, we'll be looking for mouse clicks in one of three places. In this code the mouse click position was obtained in the main while True loop, but it could have ALSO been obtained inside of the mousebutton down portion of the code. (This latter design idea would be slightly more efficient.)

On the next page we include the event code step by step:

```
            # If you close the game.
             if event.type == QUIT:
                 pygame.quit()
                 sys.exit()

             # If you press down a key.
             if event.type == KEYDOWN:
                 # If that key was the escape button do this.
                 if event.key == K_ESCAPE:
                     pygame.quit()
                     sys.exit()
```

We've chosen to give the user two ways to exit the game, either the GUI exit button or the escape key.

The rest of the events we look for will be mouse button clicks, which if they happened, occurred at the position (mx, my).

```
          # If you click on screen.
            if event.type == MOUSEBUTTONDOWN:

                # 1 - left click
                if event.button == 1:

                    # Activate game.
                    if button_1.collidepoint((mx, my)):

                        name = sushifunctions.getname()
                        myScore = sushifunctions.playgame(name)

sushifunctions.updateScores("highscores.txt",[myScore,name])

                    # Show leaderboard.
                    if button_2.collidepoint((mx, my)):
                        sushifunctions.showhighscores("highscores.txt")

                    # Get out.
                    if button_3.collidepoint((mx, my)):
                        pygame.quit()
                        sys.exit()
```

All of the actions detected are through the left button click. We can use pyGame's built in collidepoint code to check if the mouse button click occured in button_1, button_2 or button_3 (all rectangles).

If the user selects button_1, then we get the user's na,e via the getname() function in sushifunctions, call the playgame function and then store the score returned by it in the variable myScore. Then, we update our list of high scores based on this game played via the updateScores function.

If the user selects button_2, we just run the showhighscores function, passing in the name of the file with the scores.

Finally, if the user presses the Quit button, we end the program.

II. Use of Classes

This game will have objects falling vertically from the sky and a cat at the bottom (the player). The objects falling from the sky are sushi and Detos. The player earns 1 point for each piece of sushi eaten and loses a life for each collision with a Detos. The player gets three lives total. This, in terms of objects, we have sushi pieces, Detos and the Cat. There will be two classes used from which all of the objects for the game will be created. Both of the classes created will directly inherit from pygame.sprite.Sprite.

First, the Obj class. Both the sushi and Detos objects will be of this type:

```
# Inherit Sprite, use this class for sushi and detos.
class Obj(pygame.sprite.Sprite):

    # constructor (and we can use super instead of sprite.Sprite
    def __init__(self, img, pos, speedX, speedY):
        pygame.sprite.Sprite.__init__(self)
        self.image = img
        self.rect = self.image.get_rect()
        self.rect.x = pos[0]
        self.rect.y = pos[1]
        self.speedX = speedX
        self.speedY = speedY

    def draw(self, surface):
        surface.blit(self.image, (self.rect.x, self.rect.y))

    # update would do nothing if blank, but let's make it move
    def update(self):
        self.rect.x += self.speedX
        self.rect.y += self.speedY
```

Very little is special here. The image and rectangle are required and beyond that we just add speedX and speedY. Both the draw and update methods are as one would expect them to be.

The player has limited movement (can only move left and right on the bottom of the screen.) Also, the player can wrap around the bottom and the player has a top maximum speed in either direction. The extra code in this class is largely to enforce these limitations.

First, let's look at just the first three corresponding methods in the player class as the ones above in Obj:

```
# Inherit Sprite, use this class for the player.
class Player(pygame.sprite.Sprite):

    # constructor (and we can use super instead of sprite.Sprite
    def __init__(self, img, pos):
        pygame.sprite.Sprite.__init__(self)
        self.image = img
        self.rect = self.image.get_rect()
        self.rect.x = pos[0]
        self.rect.y = pos[1]
        self.speedX = 0

    def draw(self, surface):
        surface.blit(self.image, (self.rect.x, self.rect.y))

    # update would do nothing if blank, but let's make it move
    def update(self, SCREEN_W):
        self.rect.x = (self.rect.x + self.speedX + SCREEN_W)%SCREEN_W
```

As can be seen, the first two are near identical, but the update doesn't mention y at all and also provides code to do the wrap around screen implementation via mod.

Now, here's the code that limits how the speed of the cat can change:

```
    # updates the current speed by dx
    def changeSpeed(self, dx):

        # Speed update.
        self.speedX += dx

        # Lets me change directions faster, to go left.
        if dx < 0 and self.speedX > 0:
            self.speedX = 0

        # Same for right.
        if dx > 0 and self.speedX < 0:
            self.speedX = 0

        # This is my max speed right.
        if self.speedX > 9:
            self.speedX = 9

        # And max speed left.
        if self.speedX < -9:
            self.speedX = -9
```

Basically, we start with the usual update, but then we allo for a few things: (1) we allow the player to quickly change direction, no matter how fast he is going via the first two if statements, and (2) we limit the maximum speed per frame the player is moving to 9.

The last method in the class calculates the total number of intersections with the player and any item in the sushiGroup and removes these items from the sushiGroup:

```
    # Returns how many items in sushiList intersect with me and removes them
    # from sushiList.
    def numIntersect(self, sushiGroup):

        res = 0

        # Go through list backwards.
        for item in sushiGroup:

            # Here is a collision, add 1 to answer and remove from list.
            if self.rect.colliderect(item.rect):
                res += 1
                sushiGroup.remove(item)

        # Here is our answer.
        return res
```

III. Rest of the Functions

The removeUseless and draw functions are the same as in previous games. The getname function is new and allows the user to type their name and for the letters to show up in a pyGame screen. This function is a bit more arduous via pyGame than regular programming. When a pyGame window is open, we have to have the event loop running and capture each keystroke in the event loop. We have to also process the backspace key in addition to all of the letters. Since the Ascii values of letters are contiguous, that allows us to avoid an if statement with 26 branches. We can do string manipulation as needed as the user types letters, types the backspace key and types the enter key (to signify the name is completed). We'll be redrawing the current state of this string buffer onto the screen name times a second. This function will draw a new screen and have this screen disappear when the user is done. Here is the setup of the function:

```
def getname():
    pygame.init()
    clock = pygame.time.Clock()
    bigfont = pygame.font.SysFont(None, 60)
    DISPLAYSURF = pygame.display.set_mode((SCREEN_W, SCREEN_H))
    pygame.display.set_caption("Get Name Screen")
    buffer = ""
    textbox = pygame.Rect(550, 100, 300, 50)

    while True:
        DISPLAYSURF.fill((0, 0, 0))
        draw('Enter Your Name',bigfont, (255, 255, 255), DISPLAYSURF, 200, 100)
        pygame.draw.rect(DISPLAYSURF, (192,192,192), textbox)
        draw(buffer, bigfont, (255, 255, 255), DISPLAYSURF, 550, 100)

        # Get events from the queue
        for event in pygame.event.get():
            # Events processed here.

        pygame.display.update()
        clock.tick(60)
```

This is fairly similar to the main screen set up, but with fewer buttons and boxes.

The event loop is critical though, since we have to detect and process the pressing of 54 different keys (26 lowercase letters, 26 uppercase letters, backspace and enter). If we wanted to allow spaces we could, but this implementation does not. As previously mentioned, we can exploit that character Ascii values are contiguous to shorten our code:

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

    # If you close the game.
    if event.type == QUIT:
        pygame.quit()
        sys.exit()

    # If you press down a key.
    if event.type == KEYDOWN:

        # If that key was the escape button do this.
        if event.key == K_ESCAPE:
            pygame.quit()
            sys.exit()

        # Appends a lowercase letter.
        if event.unicode >= 'a' and event.unicode <= 'z' and len(buffer) < 15:
            buffer = buffer + event.unicode

        # Appends an uppercase letter.
        if event.unicode >= 'A' and event.unicode <= 'Z' and len(buffer) < 15:
            buffer = buffer + event.unicode

        # Removes the last letter.
        if event.key == K_BACKSPACE and len(buffer) > 0:
            buffer = buffer[:len(buffer)-1]

        # Done with the name return it.
        if event.key == K_RETURN:
            return buffer
```

For the event, the unicode variable represents the Ascii value of the key pressed. In the code above, we only append a letter to the current buffer if it's an lowercase or uppercase letter and that the length of the string is no more than 15 characters long. This limits any of the names entered to 15 letters or less. The 16th letter will simply never get added to the buffer. If the user presses the backspace button, ONLY if there's at least one letter in the string is that letter (the last letter) removed. Without the length check, the string slice operation would crash the program. Finally, when the enter key is recorded, the getname method stops in its tracks and returns the current state of the buffer.

The rest of the code has three long functions:

1. playgame
2. showhighscores
3. updateScores

Both #2 and #3 are near identical to the corresponding functions shown in the fruit game, so these notes will only break down the playgame function in detail.

IV. playgame function

The beginning of the playgame function sets up many variables with their initial values and loads all the necessary pictures (there are three). These variables set up levels with more detos as the user's score increases. The levels go from 0 to 4. Once the user hits level four, the frequency of the detos and sushi stay fixed. As the levels increase, we get more detos and more sushi. Here is that set up:

```
def playgame(player):

    # Basic set up.
    pygame.init()
    DISPLAYSURF = pygame.display.set_mode((SCREEN_W, SCREEN_H))
    pygame.display.set_caption("Sushi Game!")
    clock = pygame.time.Clock()

    # Store sushi here....
    sushi = []

    # Store Deetos here.
    detos = []

    # Initialize variables.
    myscore = 0
    lives = 3
    DY = 5
    DX = 0
    DETOSRATE = [100, 50, 30, 15,  5]
    SUSHIRATE = [ 50, 40, 30, 20, 10]
    level = 0

    # This is the one image we will use.
    sushiPic = pygame.image.load("dangoo.jpg")
    sushiPic = pygame.transform.scale(sushiPic,(100,50))

    # This is what we're avoiding.
    detosPic = pygame.image.load("detosarenogood.png")
    detosPic = pygame.transform.scale(detosPic,(50,50))

    myImage = pygame.image.load("spottedoof.jpg")
    myImage = pygame.transform.scale(myImage,(80,80))
    me = Player(myImage, (SCREEN_W/2, SCREEN_H-80))

    # So we know how many frames we've run.
    step = 0

    curT = time.time()
```

By default, all the objects have a DY of 5. The DETOSRATE and SUSHIRATE is once in how many frames a new object of that type is added. Each image is also loaded and scaled. The player starts with 3 lives and is at level 0.

In the game loop we update our velocity if the user presses the left or right arrow keys. After that, we add images based on the level using mod to make sure an image gets added periodically based on the rates set up previously. Each image (sushi or detos) starts at the top of the screen at a randomly chosen x coordinate. Here is that code:

```
while True:

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

          if event.type == KEYDOWN:
               if event.key == K_LEFT:
                   me.changeSpeed(-3)
               if event.key == K_RIGHT:
                   me.changeSpeed(3)

      # These images are big, so I only add a drop once every few frames.
      if step%SUSHIRATE[level] == 0:
          x = random.randint(1, SCREEN_W)
          item = Obj(sushiPic, (x,0), DX, DY)
          sushi.append(item)

      # These will be created half as often as the sushi.
      if step%DETOSRATE[level] == 0:
          x = random.randint(1, SCREEN_W)
          item = Obj(detosPic, (x, 0), DX, DY)
          detos.append(item)
```

So our delta X changes by 3 moving right or left for the appropriate key presses, and then if the step variable is divisible by the sushirate for the current level, a sushi object is added, and the same for the detos list.

Then we start drawing items on the screen. The background, followed by all of the sushi and detos, and text stating the player's score and number of lives left. We move all of the objects for the next screen:

```
        # Start drawing.
        DISPLAYSURF.fill(WHITE)

        # blit allows us to draw a surface onto another surface.
        for x in sushi:
            x.draw(DISPLAYSURF)
        for x in detos:
            x.draw(DISPLAYSURF)
        me.draw(DISPLAYSURF)

        # Update score, lives.
        myscore += me.numIntersect(sushi)
        lives -= me.numIntersect(detos)
        font = pygame.font.SysFont("Arial", 24)

        # Draw score, lives.
        draw("Score: "+str(myscore), font, BLUE, DISPLAYSURF, 20, 20)
        draw("Lives: "+str(lives), font, RED, DISPLAYSURF, 20, 120)

        # Move all objects.
        for x in sushi:
            x.update()
        for x in detos:
            x.update()
        me.update(SCREEN_W)
```

Finally, we process a loss (if lives equal 0), and then we update the rest of our variables. Notice that items also fall faster as the level increases:

```
        # We have died.
        if lives == 0:

            # Display final score, wait a bit and return.
            draw(player+", you Died. Final Score: "+str(myscore), font, BLUE,
DISPLAYSURF, 20, 220)
            pygame.display.update()
            pygame.time.wait(2000)

            # We return this to the main screen.
            return myscore

        # Update the display and process lists.
        level = min(4, myscore//10)
        DY = 5 + 2*level
        pygame.display.update()
        removeUseless(sushi)
        removeUseless(detos)
        clock.tick(30)
        step += 1
```