LAB 14

TEC 284

LOGICAL PROGRAM DESIGN

OBJECTIVES:

* Create a "simple” game using Python and the PyGame library
* Use principles of game design to develop a logical program flow

SUPPLIES:

1x Raspberry Pi, or any computer with Python installed

INTRODUCTION:

*In this lab activity, you will be going through the process of creating a simple, side-scrolling game that will be ran locally on the Pi. That’s right – we are once again playing video games in this class! The PyGame library documentation can be found* [*here*](https://www.pygame.org/docs/) *if you need to reference it.*

*This is a long lab activity! Make sure you and your partner take turns doing the programming work!*

DOWNLOADING PYGAME

1. Open a new Terminal window on your Pi. Type in the command:

pip install pygame --break-system-packages

More than likely, this will already be installed on your Pi.

A BASIC PYGAME EXAMPLE

1. Open Thonny and copy and paste the following program in a new project:

# Simple pygame example

# Import and initialize the pygame library

import pygame

pygame.init()

# Import random for random numbers  
import random

# Set up the drawing window

screen = pygame.display.set\_mode([500, 500])

# Run until the user asks to quit

running = True

while running:

# Did the user click the window close button?

for event in pygame.event.get():

if event.type == pygame.QUIT:

running = False

# Fill the background with white

screen.fill((255, 255, 255))

# Draw a solid blue circle in the center

pygame.draw.circle(screen, (0, 0, 255), (250, 250), 75)

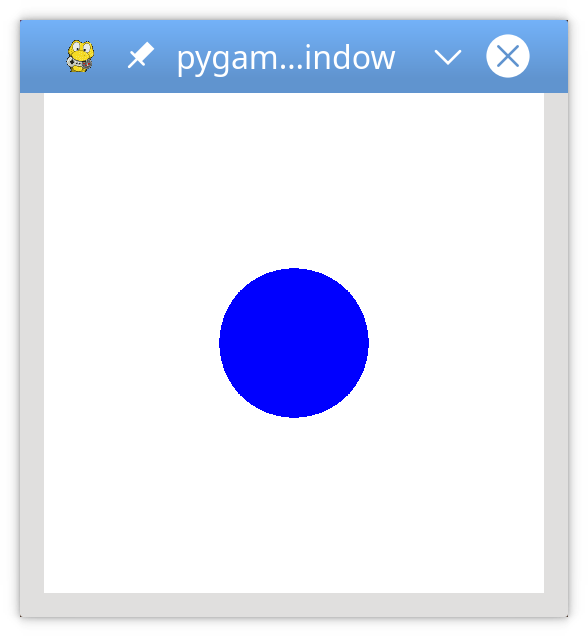
# Update the contents of the display

pygame.display.flip()

# When done, quit the program

pygame.quit()

1. When you run this code, you should see a window pop up with a white background and a solid blue circle inside, as shown:



1. Before you start writing any code, it’s always a good idea to have some design in place. The gameplay and mechanics will obviously be simple:
   1. The goal of the game is to avoid incoming obstacles:
      1. The player starts on the left side of the screen.
      2. The obstacles enter randomly from the right and move left in a straight line.
   2. The player can move left, right, up, or down to avoid the obstacles.
   3. The player cannot move off the screen.
   4. The game ends either when the player is hit by an obstacle or when the user closes the window.

YOUR FIRST GAME

1. Open a fresh Thonny program. Copy and paste the following starting code:

# Import the pygame module

import pygame

# Import pygame.locals for easier access to key coordinates

# Updated to conform to flake8 and black standards

from pygame.locals import (

RLEACCEL,

K\_UP,

K\_DOWN,

K\_LEFT,

K\_RIGHT,

K\_ESCAPE,

KEYDOWN,

QUIT,

)

# Initialize pygame

pygame.init()

1. What this does is import the abstractions needed to use your keyboard’s UP, DOWN, LEFT, RIGHT, and ESC key. It also imports the mouse. Let’s add some more gaming necessities:

# Define constants for the screen width and height

SCREEN\_WIDTH = 800

SCREEN\_HEIGHT = 600

# Create the screen object

# The size is determined by the constant SCREEN\_WIDTH and SCREEN\_HEIGHT

screen = pygame.display.set\_mode((SCREEN\_WIDTH, SCREEN\_HEIGHT))

1. Notice our object orientation here. You can almost read it backwards to see what’s going on. On the second to last line, we’re using the SCREEN\_HEIGHT and SCREEN\_WIDTH constants to set the mode of the display for PyGame.

THE GAME LOOP

1. A game loop defines the core control mechanic of the game. Every game, as it is designed, starts with a [game loop](https://gamedesignskills.com/game-design/core-loops-in-gameplay/). Our game loop will look something like this:
   1. Process user input
   2. Update the state of all game objects
   3. Update the display and audio output
   4. Maintain the speed of the game

There are two events that can end our loop:

1. You close the game (obviously)
2. You collide with an obstacle

Let’s start with that second thing. Any input from the user results in an *event* being generated. Every *event* has a *type* associated with it.

1. Add the following to your code. This will be the basis of our game loop:

# Variable to keep the main loop running

running = True

# Main loop

while running:

# Look at every event in the queue

for event in pygame.event.get():

# Did the user hit a key?

if event.type == KEYDOWN:

# Was it the Escape key? If so, stop the loop.

if event.key == K\_ESCAPE:

running = False

# Did the user click the window close button? If so, stop the loop.

elif event.type == QUIT:

running = False

1. Our new variable, *running*, will basically set up the equivalent of an Arduino loop() function. Except this time, we can set our *running* variable to “False” to end the loop entirely. To test your program so far, try running it. You should get a black screen with nothing in it. Press the ESC key or click the little “X” to close your window.
   1. Note: You may need to add pygame.quit() to close the window as well.

DRAWING ON THE SCREEN

1. In PyGame, a *surface* is a rectangular object that you can draw on. The background *screen* is a *surface*, but we can also make our own *surface* object that is separate from our screen.
2. Add this code to your program:

# Fill the screen with white

screen.fill((255, 255, 255))

# Create a surface and pass its length and width

surf = pygame.Surface((50, 50))

# Give the surface a color to separate it from the background

surf.fill((0, 0, 0))

rect = surf.get\_rect()

# Put the center of surf at the center of the display

surf\_center = (

(SCREEN\_WIDTH-surf.get\_width())/2,

(SCREEN\_HEIGHT-surf.get\_height())/2

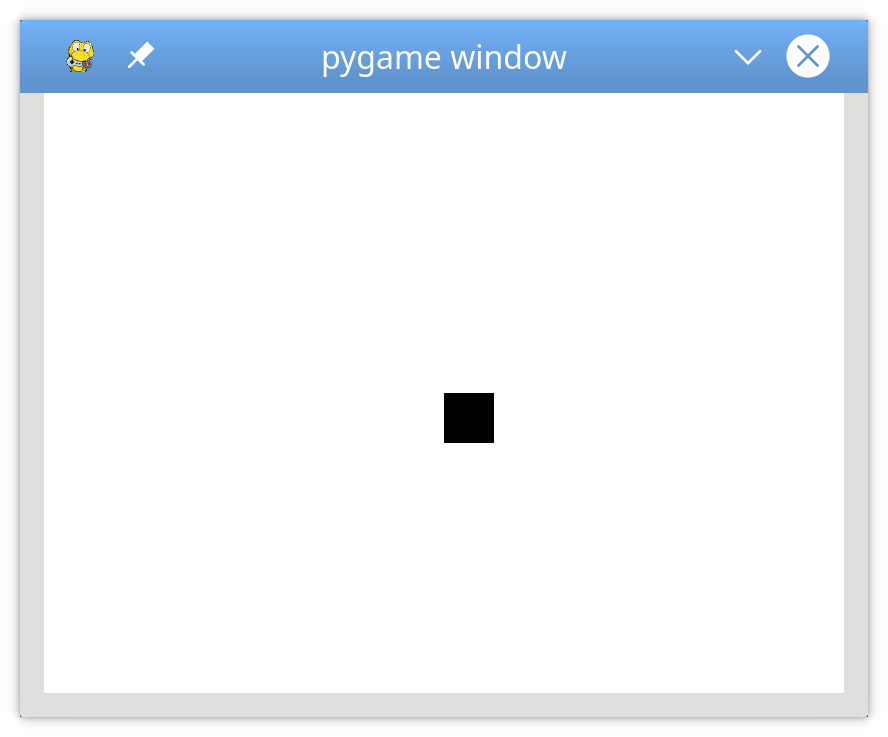
)

# Draw surf at the new coordinates

screen.blit(surf, surf\_center)

pygame.display.flip()

1. *Blit* stands for “block transfer”, and it’s how we put one *surface* object onto another.
2. Running your code now, you should see a lovely window like this:



ADDING SPRITES

1. Fans of classic games may be familiar with the term “sprites” that are two dimensional representations of *something*. Maybe a character, an enemy, or part of the environment. We’ll eventually make our sprites look nicer, but for now, let’s just use some lovely rectangles. Underneath where you define the constants for the size of the screen, we’ll make a new class that holds your character’s sprite:

# Define a player object by extending pygame.sprite.Sprite

# The surface drawn on the screen is now an attribute of 'player'

class Player(pygame.sprite.Sprite):

def \_\_init\_\_(self):

super(Player, self).\_\_init\_\_()

self.surf = pygame.Surface((75, 25))

self.surf.fill((255, 255, 255))

self.rect = self.surf.get\_rect()

Just above your main loop, where you set running to true, paste the following code:

# Instantiate player. Right now, this is just a rectangle.

player = Player()

Finally, add the following lines right above where you update the display at the bottom of your code:

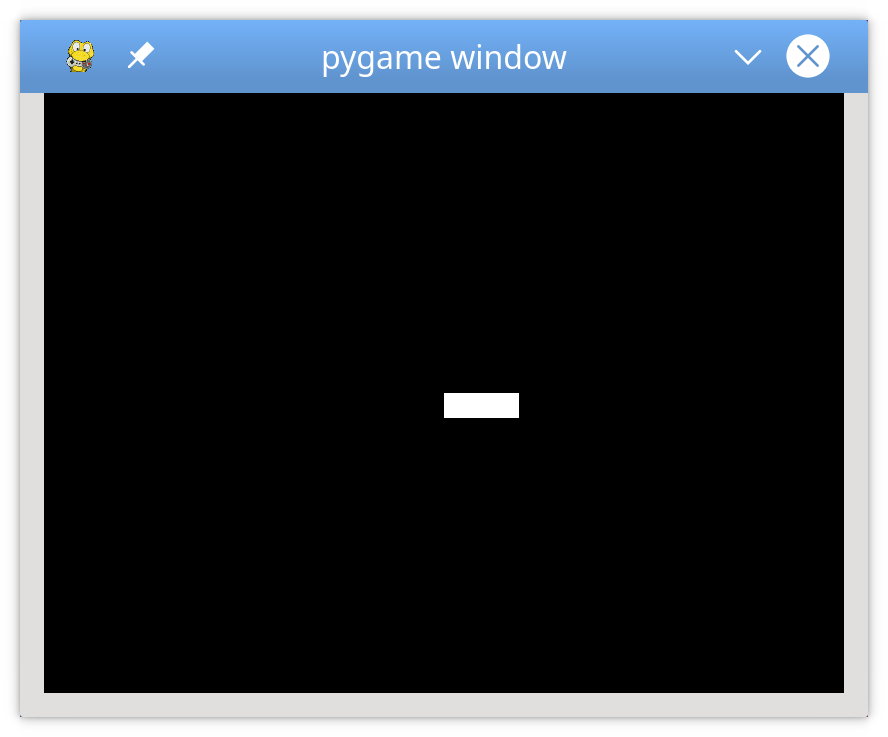
# Fill the screen with black

screen.fill((0, 0, 0))

# Draw the player on the screen

screen.blit(player.surf, player.rect)

Okay, now, when you run this code, you’ll see a very exciting rectangle on a black background. We’re getting there!



ADDING INTERACTIVTIY

1. We’ve done a lot of work to make the basic building blocks. Now, let’s add some user input. Add these lines right before where you make the screen black:

# Get the set of keys pressed and check for user input

pressed\_keys = pygame.key.get\_pressed()

# Update the player sprite based on user keypresses

player.update(pressed\_keys)

Right before you initialize PyGame, add the following function:

# Move the sprite based on user keypresses

def update(self, pressed\_keys):

if pressed\_keys[K\_UP]:

self.rect.move\_ip(0, -5)

if pressed\_keys[K\_DOWN]:

self.rect.move\_ip(0, 5)

if pressed\_keys[K\_LEFT]:

self.rect.move\_ip(-5, 0)

if pressed\_keys[K\_RIGHT]:

self.rect.move\_ip(5, 0)

# Keep player on the screen

if self.rect.left < 0:

self.rect.left = 0

if self.rect.right > SCREEN\_WIDTH:

self.rect.right = SCREEN\_WIDTH

if self.rect.top <= 0:

self.rect.top = 0

if self.rect.bottom >= SCREEN\_HEIGHT:

self.rect.bottom = SCREEN\_HEIGHT

Run your program and try to move your rectangle around! You may notice that the player’s rectangle can move very fast if a key is held down. You’ll work on that later. For now, let’s add some enemies!

ADDING SOME ENEMIES

1. Let’s add a new class of enemy sprites to our code. Paste this block of code into your main “while running” game loop:

# Define the enemy object by extending pygame.sprite.Sprite  
# The surface you draw on the screen is now an attribute of 'enemy'  
class Enemy(pygame.sprite.Sprite):  
 def \_\_init\_\_(self):  
 super(Enemy, self).\_\_init\_\_()  
 self.surf = pygame.Surface((20, 10))  
 self.surf.fill((255, 255, 255))  
 self.rect = self.surf.get\_rect(  
 center=(  
 random.randint(SCREEN\_WIDTH + 20, SCREEN\_WIDTH + 100),  
 random.randint(0, SCREEN\_HEIGHT),  
 )  
 )  
 self.speed = random.randint(5, 20)  
  
 # Move the sprite based on speed  
 # Remove the sprite when it passes the left edge of the screen  
 def update(self):  
 self.rect.move\_ip(-self.speed, 0)  
 if self.rect.right < 0:  
 self.kill()

1. In this code, we are telling the enemy to start at a random location along the right edge of the screen. The center of the rectangle is just off the screen, located at some position between 20 and 100 pixels away from the right edge, and somewhere between the top and bottom edges. We define the enemy’s speed as a random value between 5 and 20 (you can change this if you want for increased difficulty). The final few lines of code here check if the right edge of the enemy has gone past the left side of the screen. If so, we essentially delete the enemy from the game and from memory.
2. In order to make collision detection and rendering easier for ourselves, we are going to put all of our enemy sprites into a group. Right above where you initiate running = true, paste in the following code:

# Create groups to hold enemy sprites and all sprites  
# - enemies is used for collision detection and position updates  
# - all\_sprites is used for rendering  
enemies = pygame.sprite.Group()  
all\_sprites = pygame.sprite.Group()  
all\_sprites.add(player)

1. And finally, to render all of the sprites at once, we can add these lines of code at the bottom, right underneath where we set the background to be black:

# Draw all sprites  
for entity in all\_sprites:  
 screen.blit(entity.surf, entity.rect)

1. Now, anything put into *all\_sprites* will be drawn with every frame, whether it’s an enemy or the player. As you can imagine, this is very useful for us.
2. Finally, let’s add some enemies! We will need to define a custom event within PyGame to do so. Essentially, every time a new enemy spawns, we call it an *event*. So, right above where you instantiate the player, add the following code block:

# Create a custom event for adding a new enemy  
ADDENEMY = pygame.USEREVENT + 1  
pygame.time.set\_timer(ADDENEMY, 250)

1. Now, we are essentially creating a new, unique event (spawning a new enemy sprite) every 250 ms. Again, maybe this is something you change to increase or decrease the difficulty!
2. In the main game loop, right underneath the code that handles the player closing out of the game, add this block of code that handles our new event. Make sure your indentation is correct, such that it lines up with your FOR loop.

# Add a new enemy?  
 elif event.type == ADDENEMY:  
 # Create the new enemy and add it to sprite groups  
 new\_enemy = Enemy()  
 enemies.add(new\_enemy)  
 all\_sprites.add(new\_enemy)

1. And then, right underneath where we get the set of keys that player has pressed, we can add this code:

# Update enemy position  
 enemies.update()

1. Cool! Now, when you run your code, you should see a bunch of rectangles fly past your screen. We’re getting somewhere now!

DETECTING COLLISIONS WITH OBSTACLES

1. Programming collision detection is usually a very tedious thing to do in game design and involves a lot of complex math. Thankfully, the entire reason that we use modules is to abstract away that level of detail. With PyGame, we can simply run a few lines of code that will automatically detect collisions. Right underneath where you draw all of the sprites on the screen, add this block of code:  
    # Check if any enemies have collided with the player  
    if pygame.sprite.spritecollideany(player, enemies):  
    # If so, then remove the player and stop the loop  
    player.kill()  
    running = False
2. Easy, right? If any kind of collision is detected between the player and the enemies, we remove the player from memory and stop the game. Game over!

IMPROVING THE AESTHETICS WITH IMAGES

1. If you run your code now, you should have the basics of a game up and running. You can move, there are obstacles, and you have a clear goal. However, it looks ugly, let’s be honest. Up until now, there hasn’t been much room for creativity with this lab, but now is the time for you to make this game your own.
2. Let’s replace our rectangular player and enemy sprites with something more exciting. Think of what you want your game to be about and download some images (ethically, of course) to make your dream a reality. Maybe you want your game to be a high-speed jet avoiding incoming missiles, or maybe a fish avoiding incoming sharks. Either way, come up with a theme for your game by downloading an image for:
   1. The player character
   2. The enemies / obstacles
   3. Background sprites (to add depth, for example: clouds, cars, buildings, anything that will scroll along the screen in the background)
3. If you are on a Pi with a slow or non-existent internet connection, you can download the “default” assets by typing this into your Terminal:

git clone https://github.com/josbor1/TEC-284-PyGame-Resources.git

1. Find your code for defining the “player” object. It should be near the top, right underneath the screen width / height constants. Replace the entire player class definition with this:

# Define the Player object by extending pygame.sprite.Sprite  
# Instead of a surface, use an image for a better-looking sprite  
class Player(pygame.sprite.Sprite):  
 def \_\_init\_\_(self):  
 super(Player, self).\_\_init\_\_()  
 self.surf = pygame.image.load("<PATH TO YOUR PLAYER IMAGE HERE>").convert()  
 self.surf.set\_colorkey((255, 255, 255), RLEACCEL)  
 self.rect = self.surf.get\_rect()

1. Obviously, you will need to replace the image.load function with the path to wherever you saved your player character image on your Pi. Remove the < > as well, as these are placeholders.
2. Similarly, find your enemy class declaration. Replace the entire declaration with this:

# Define the enemy object by extending pygame.sprite.Sprite  
# Instead of a surface, use an image for a better-looking sprite  
class Enemy(pygame.sprite.Sprite):  
 def \_\_init\_\_(self):  
 super(Enemy, self).\_\_init\_\_()  
 self.surf = pygame.image.load("<PATH TO YOUR ENEMY SPRITE HERE>").convert()  
 self.surf.set\_colorkey((255, 255, 255), RLEACCEL)  
 # The starting position is randomly generated, as is the speed  
 self.rect = self.surf.get\_rect(  
 center=(  
 random.randint(SCREEN\_WIDTH + 20, SCREEN\_WIDTH + 100),  
 random.randint(0, SCREEN\_HEIGHT),  
 )  
 )  
 self.speed = random.randint(5, 20)

1. Great! Now we have sprites for the player and enemies. If your vision calls for it, we can also make a background object class. In the code below, I’ll show you how to make a background cloud object to make it look like clouds are flying past. You can edit this code as needed to make whatever you want! Paste this code underneath the enemy class definition:

# Define the cloud object by extending pygame.sprite.Sprite  
# Use an image for a better-looking sprite  
class Cloud(pygame.sprite.Sprite):  
 def \_\_init\_\_(self):  
 super(Cloud, self).\_\_init\_\_()  
 self.surf = pygame.image.load("<PATH TO CLOUD IMAGE>").convert()  
 self.surf.set\_colorkey((0, 0, 0), RLEACCEL)  
 # The starting position is randomly generated  
 self.rect = self.surf.get\_rect(  
 center=(  
 random.randint(SCREEN\_WIDTH + 20, SCREEN\_WIDTH + 100),  
 random.randint(0, SCREEN\_HEIGHT),  
 )  
 )  
  
 # Move the cloud based on a constant speed  
 # Remove the cloud when it passes the left edge of the screen  
 def update(self):  
 self.rect.move\_ip(-5, 0)  
 if self.rect.right < 0:  
 self.kill()

1. Remember where we created a custom event for spawning the enemies? Replace that section of code with this to also summon new clouds:

# Create custom events for adding a new enemy and a cloud  
ADDENEMY = pygame.USEREVENT + 1  
pygame.time.set\_timer(ADDENEMY, 250)  
ADDCLOUD = pygame.USEREVENT + 2  
pygame.time.set\_timer(ADDCLOUD, 1000)

1. And where we created groups of sprites, we will need to add the clouds. Find your section of code where you created your groups and modify it like so:

# Create groups to hold enemy sprites, cloud sprites, and all sprites  
# - enemies is used for collision detection and position updates  
# - clouds is used for position updates  
# - all\_sprites is used for rendering  
enemies = pygame.sprite.Group()  
clouds = pygame.sprite.Group()  
all\_sprites = pygame.sprite.Group()  
all\_sprites.add(player)

1. Right underneath the block of code that starts with “Add a new enemy?” we will add this block of code:

# Add a new cloud?  
 elif event.type == ADDCLOUD:  
 # Create the new cloud and add it to sprite groups  
 new\_cloud = Cloud()  
 clouds.add(new\_cloud)  
 all\_sprites.add(new\_cloud)

1. Finally, right above where you make the screen black, we need to update the position of the clouds:

# Update the position of enemies and clouds  
enemies.update()  
clouds.update()

1. If you want to follow this cloud example, you can also edit the screen color to a nicer shade of blue. Use the RGB values of (135, 206, 250) for the screen.fill command.

ESTABLISHING THE GAME’S SPEED

1. Now, you may (or may not) have noticed that your enemies and background images move incredibly fast. This is because our code is being run as fast as the Pi’s processor will allow, which is not ideal. Let’s add some final lines of code to change that.
2. At the bottom of your program, find the line that says:

# Flip everything to the display  
 pygame.display.flip()

1. Just above that line, paste in this block of code:

# Setup the clock for a decent framerate  
clock = pygame.time.Clock()

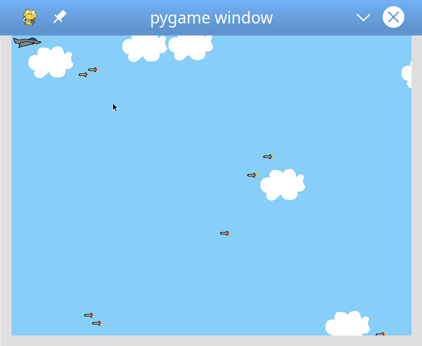
1. Below the “flip everything” line, add this:

# Ensure program maintains a rate of 30 frames per second  
clock.tick(30)

1. Great, now your program should run at a steady 30 frames per second (FPS). You can play around with this number to find an FPS that works best for you.

CONCLUSION

1. Now, when you run your program, you should have a working game with images representing sprites. You can go back through your program and tweak some variables here and there to fine-tune your game to your desired difficulty. Here’s what an example program may look like:



CONCLUSION

1. If this kind of thing interests you, there is a whole host of resources available for you to explore. Learn more about PyGame here: <https://www.pygame.org>

FINISHED PROGRAM

1. Once you have finished, copy and paste your entire program below: