**Week1:**

**Pygame: Bounce Program**

Starting from here, as well as the week 2 of Lab 2, we need to use two images as two balls for bouncing or colliding. We downloaded two images of the badge of Cornell and Berkeley. In order to reduce the complexity of velocity calculation, we resized all images to (50,50) squares, and treat them as round circles with radius of 25 pixels. And the two balls have the same mass.

**[bounce.py]**

In this program, we only used the Cornell badge as the single ball. The logic of the whole program is simple:

a) First, we initialized the pygame screen, with the width and height as 320 and 240 respectively. Then we loaded the image and get the corresponding rectangle using “ball.get\_rect()”. We can access to the position of the rectangle easily. We also needed to initialize the speed of the ball, we set it as (1,2).

b) After initialization, the program contains a forever loop, every step of which updates a single frame display of the pygame. In this while loop, it first calculates the next-frame-speed of the ball using “rect.move(speed)”. When the program detects whether there is a collision between any border and the ball. When collision happens, and if the ball hit the left or the right border, the horizontal speed of it gets reversed so it can move from the wall. Similarly, it the ball hit the top or the bottom border, the vertical speed gets reversed. After the right calculation of speed, the program update the display of pygame: 1. Uses “screen.fill()” to turn the screen to whole black; 2. Uses “screen.blit()” to add the image with new position to the screen; 3. Uses “pygame.display.flip()” to update the display.

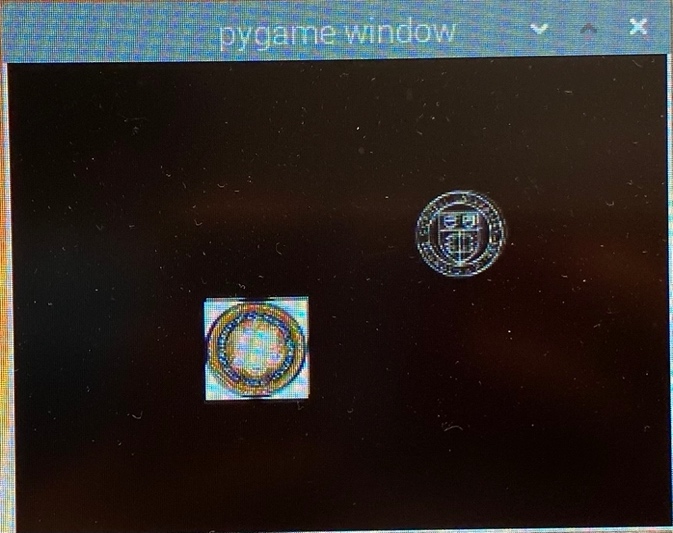
c) In order to enable us to stop the program even when we run the program directly on the piTFT but not via SSH, we added a quit button using GPIO. This is really important in this part. Before adding the GPIO button, we ran our programs several times on the piTFT and didn’t know how to quit it. Meanwhile, the power light of the Pi turned red, we thought that the Pi got stuck and pulled off the power. With the reminder from TAs and professor, we introduced this GPIO button, it helped a lot!

As c) mentioned, it occurred several times that we cannot terminate the pygame program, so we had to pow-off the Pi physically. When we power-up the Pi again, we met the problem that we could not enter the system, and the monitor was showing a rainbow spectrum. We debugged this problem with professor by putting our SD card in professor’s machine to see whether the SD card is the problem. Luckily the Pi is the broken one, otherwise we would have lost all the work we had done. We replaced the machine and kept proceeding the lab.



**[two\_bounce.py]**

Since we don’t need to add collision to this program, so two\_bounce.py is much similar to bounce.py except that we introduced another picture. So, in this program, what is different is that when loading images, apart from the Cornell badge, we also loaded Berkeley’s badge and got the rectangle object of Berkeley. In the main part of the program, which is the while loop, the only difference is that two speeds are updated based on the same rule since balls are “transparent” to each other.



**[two\_collide.py]**

Two\_collide.py is much different from two\_bounce.py since we need to detected whether the two balls collide. Before starting the implementation, we needed to figure out how speeds change when collision happens. We read the materials provided by Canvas, and the [website: <https://scipython.com/blog/two-dimensional-collisions/> ] shows when two balls collide, and no energy dissipates because the collision, then the speed of the two balls will be:

The physical mechanism of this program, as well as programs involved in balls collision, is based on this formula.

The main design of this program is similar to the two\_bounce.py. What is different is that in the while loop, the program detects whether the distance between two balls is less than 50(the sum of two balls’ radius). If true, it means that a collision occurred, we need to change the speed of two balls using the above formula.

Easier said than done, when it came to the actual implementation, we met two problems, and they were all proposed by professor on lectures after that week.

One problem we met is that after collision, one ball might be stuck in the wall and does not move away from the wall. This situation results from two balls colliding near to the wall. After the collision between balls, one ball’s speed might point from the wall to the empty space, although it is right in this step, problem will raise in the step of wall collision detection. Because this ball touches the wall, the speed related to the direction is reverse, and before the reverse, the direction is right, but after the reverse the speed is wrong as it leads to ball to keep moving into the wall. Our solution in this program is to make wall collision has a higher priority than ball collision, so the speed is restrictedly pointing away from the wall.

The second problem we met is that two balls might stick to each other after collision. To be honest I was so confused about this problem at the beginning. The strategy I used was using a counter to keep track of the times of collisions, if balls keep colliding for more than 50 times, the balls will set to separate positions and restart bouncing and colliding. This approach is flawed since it just rests the positions. In the week 2, we used a state machine to solve this problem more reasonably.



**Week 2:**

This week is mainly about pygame software development, I completed all pygame programs at home before the Lab2 Week2. So, on Thursday, all we did was to get the configuration of piTFT touch screen right and test our programs.

**[Configuration]**

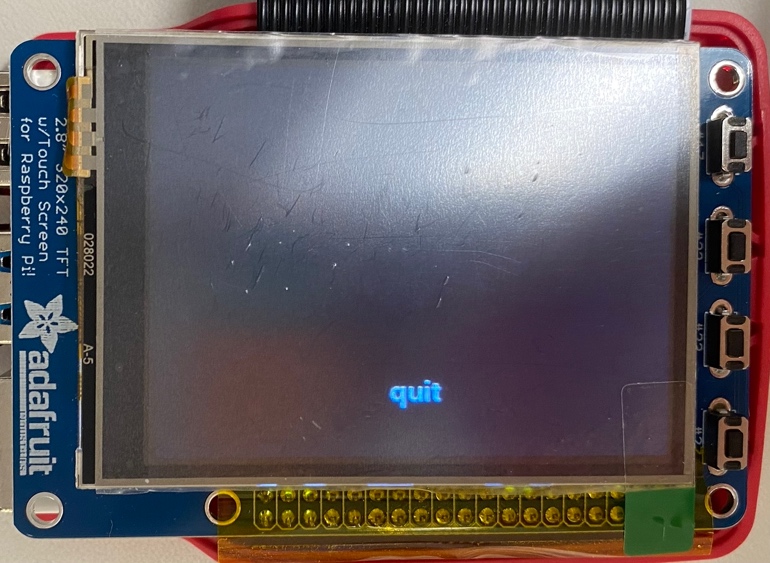
We followed the lab instruction and set up the Pi orderly, but we met a problem that our Pi could not fetch files from <http://legacy.raspbian.org/raspbian>. After consulting to professor, we found out that the problem resulted from last week when we changed our machine because the old one could not reboot. We did not know that we need to register the MAC address of the new Pi on Cornell IT, so there were some restrictions on it.

**[quit\_button.py]**

The main function of this program is to detect the piTFT touchscreen’s configurations are right. We added a quit button on the piTFT screen and in the main part of the program, we used “pygame.event.get()” to detect events on pygame, if it is a MOUSEBUTTONUP event and the event position is within the area of the quit button, the whole program stops and quits.

Although the logics and the implementations of the program were totally right, we came across a wired problem that when we touched the screen, the program just didn’t quit. We thought that there was something wrong with the piTFT configuration, so we spent much time getting connect to the ece5725 server to download the fix\_touchscreen script to test on our Pi. What was even weirder was that it totally worked well. Then I went back to my quit\_button.py to debug. After failing debugging for a while, I felt frustrated and used my finger to touch the quit button on the screen rudely, and it worked surprisingly and the program quit...

It turned out that I set up a strict rule in detecting MOUSTBUTTONUP events. I have to press exactly the quit button area, a small rectangle, as I shrunk down the front size. The previous failures actually did not exist except the algorithm is strict.



**[screen\_coordiantes.py]**

This program enables the pygame to display the coordinates that we touch on the screen. At the initialization stage, we used “pygame.font.Font()” to initialize a pygame text object. In the main loop, when the program detects a MOUSEBUTTONUP event, it reads the position corresponding to the event, and set the text as “(X,Y) is pressed!”, and the program also prints this message at console.

At first, we did not realize that we need to display the message on where we touched the screen, so we just put the text field on the top-center of the screen. When checking off, after TA’s comment we added a mechanism to let the message shows on the right place but not exceed the border.

As for the question on the instruction, if we want to collect taps of the screen, we can use a list in python, when a tap event occurs, the program appends the coordinates to the list, we can write records to a csv after the quit button is pressed and before the end of the program.



**[two\_buttons.py]**

Two\_buttons.py is more complicative than screen\_coordinates.py as aside from showing the touched coordinates, we also need to put basic controls on the two\_collide program.

a) First, we set up the env values, initialized the pygame, set up the screen size, initialized the turples for background, positions for each button, and loaded images and got their rectangles. In this program, we added a third button to clear the coordinates information that already on the screen. We also added a pause function for the Start button.

b) Rather than putting all codes in the loop and making the code messy, we modulized the code by separating different modules to different functions.

\*showControlButton(): display three buttons on the screen by three “screen.blit()”s.

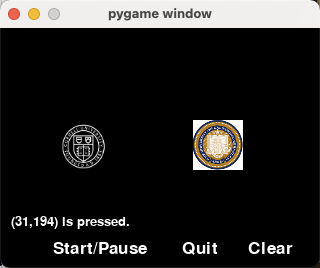
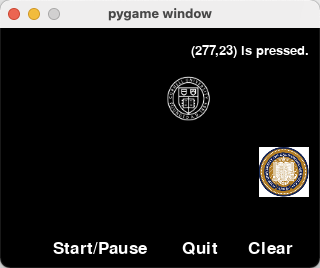
\*showCoordinates(pos): the input parameter is the position of the event, then this function will set up the text box position and the text it should show on the screen.

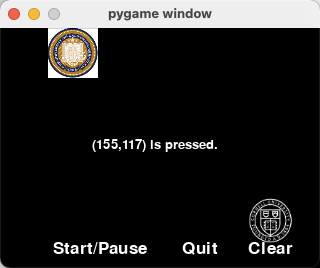
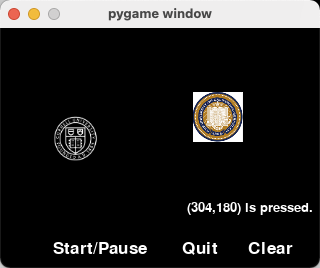
\*ballsMovement(): this function detects whether there is a collision between balls or between a ball and the wall. If true, the corresponding speed will update according to the formular mentioned before.

\*showBalls(): similar to showButton(). Displays balls on the screen with “screen.blit()”s.

c) We introduced several state variables for the control of pygame. “\_moving” is a mark of whether the program is paused. “\_stick\_count” is a counter for separating two sticked balls after 50 continuous collisions (a flawed method mentioned before, and we will use a state machine in the next program, control\_two\_coiilde.py).

d) Each step in the loop, the program firstly fills the whole screen with a black background. Secondly, it detects whether a button is pressed, it true, it will trigger the function of that button. After event detection, the program displays the new coordinates, moves the balls and updates their speeds, displays buttons and flips the screen.





**[control\_two\_collide.py]**

This is a fancy version of two\_button.py, as it has two control menus, and can speed up and slow down balls. Since we used module functions in the last program, it is much easier for us to implement this one.

First, we need to consider how to change to speed when tapping the touch screen. The FPS method proposed by professor is great as it can not only change the speed, but also normalize the speed on different devices. We initialized the FPS as 30 at the beginning, when pressing the Fast button, the FPS increases by 2 with an upper bound of 120; on the other hand, when pressing the Slow button, the FPS decreases by 2 with a lower bound of 20.

Since we have two control menus, we introduced a state variable “\_command\_page” to lead the program execute right functions at the corresponding page.

For the ball sticking problem, we introduced a state variable “\_collided” to solve it. Whenever it is detected that the two balls do not collide, this variable is set as False. When the distance of two balls is less than the threshold, if this variable is True, then just skip the speed update module; otherwise, set this state variable as True and update the speeds with the formula.

