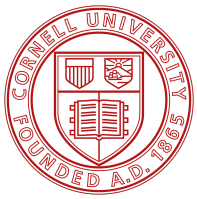
Cornell University



Electrical Computer Engineering

ECE 5725 - Embedded Operating System

LAB 2 Report

Lab Section: Wednesday 402
Lab Date: 09/14/22 & 09/21/22

Guangxu Zhai (gz244) & Cynthia Li(xl827)

September 28, 2022

1. Introduction

Lab 2 firstly introduced how to add external buttons to control the video `mplayer`. Then we did most of our code based on `PyGame` library, including the ball movements and also using physical and touch screen buttons to control the game.

2. Design And Testing

In this section, we will elaborate on lab procedures step by step and explain how we addressed issues encountered.

2.1 Adding external buttons

We utilized four resistors (two 1k ohm and two 10k ohm), a breadboard, two buttons and some wires to build up a circuit shown like below for adding two external controls connecting to the GPIO pin 6 and pin 26. (GPIO 6 and 26 are originally N/A so that we can make function adjustment for button controls). The reason why we used 1k ohm resistor here is to protect the Rpi4. Because the system we are using here is Linux. We all knows that Linux could have multiple users to control it. So if we don't have the 1k ohm resistor here, and if someone else uses software and treats the GPIO pins as an output, it could have both logical 1 and 0 into the circuit. Then we can not take a risk to directly connecting gpio pins to the ground, which might cause a short circuit and do damge to the RPi4. The selection of 1k here is since 3.3/1000 = 3.3mA, which has a safe value for the whole circuit.

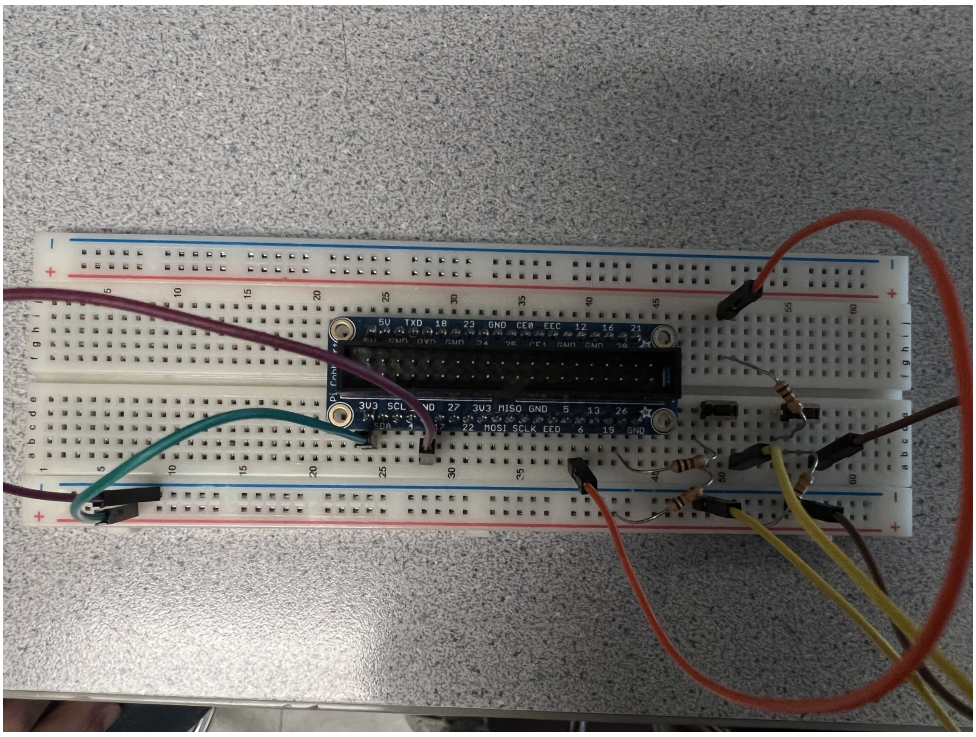


FIGURE 1: External Circuit for Adding buttons Screen

After we set up the external buttons, we made little adjustment on our `four_button.py` by adding two more buttons connected with GPIO 6 and 26. These two buttons have the function of fast forward and rewind 30s.

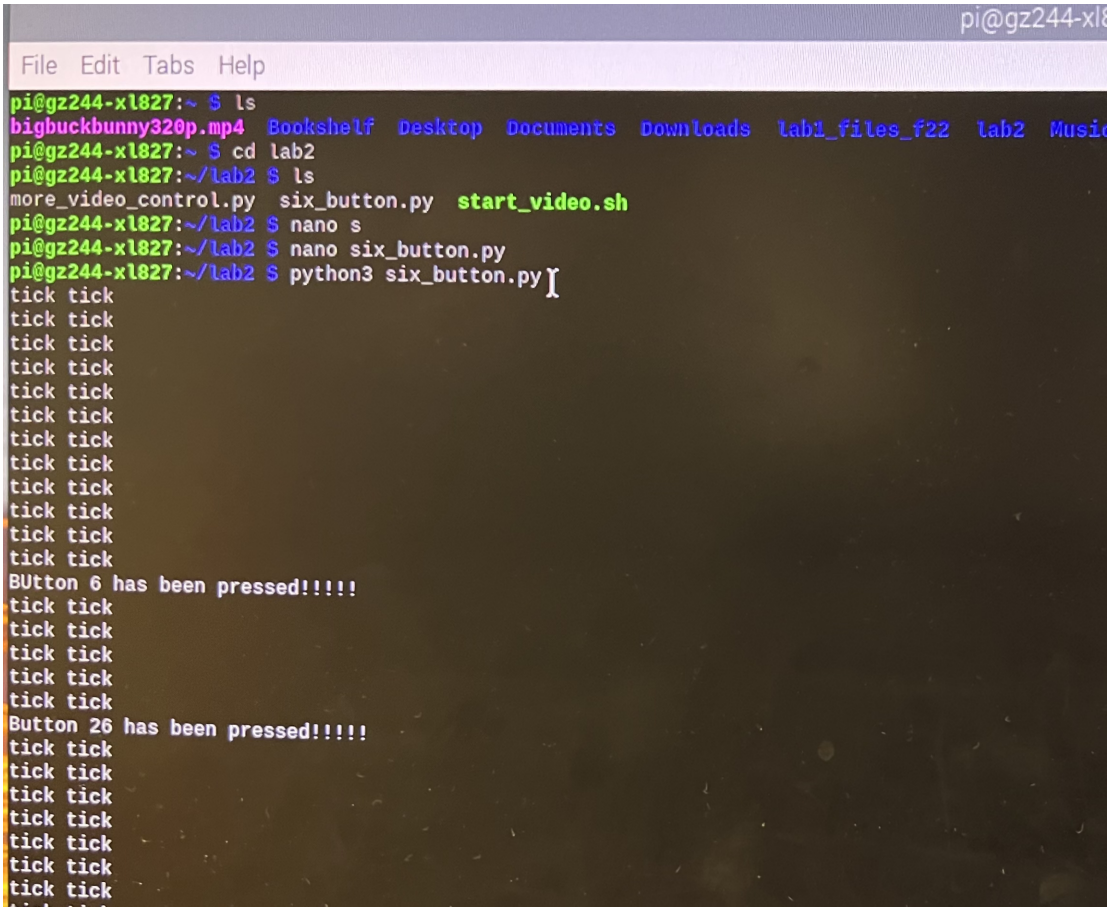


FIGURE 2: Running result of six_button.py

2.2 Interrupt callbacks

In this part, we changed our original video control code to a callback version by setting up callback routines and also defining events that access the callback routine. We followed the instruction code given in the class lectures and successfully passed this part.

For each pin, we define one event to do the sub-process command. And we detect the falling edge of those GPIO pins to know the timing of running the command.

2.3 Performance measurement with 'perf' utilities

Firstly, we installed perf by typing `sudo apt-get install linux-perf-5.10` in cmd. 'Perf' are capable of tracking the time and giving an assessment statics of performance.

Secondly we change both the callback code and non-callback code a little bit to limit the running time to a fixed 10s. For the non-callback code, we use a variable recording the curreent time before entering the while True loop. Then we add one if statement to jump out of the loop if the difference between the current time and previous recorded time exceed 10s; For the callback code, we simply added a time sleep function (10s) followed with `GPIO.leanup()` .

Thirdly, we modified `more_video_control.py` by setting different "sleep" time and used 'perf' to compare the statics. From the comparison of the results below, we found that if we decrease the polling loop times, the `Task-clock` time will decrease accordingly.

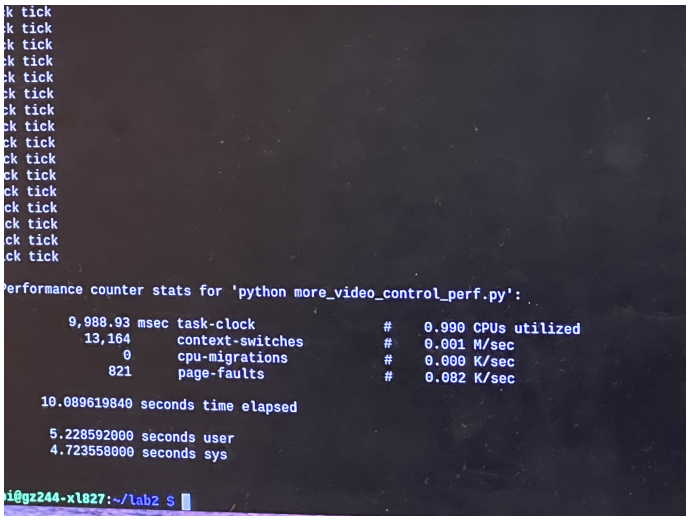


FIGURE 3: Performance measurement of video_control.py (20 milliseconds)

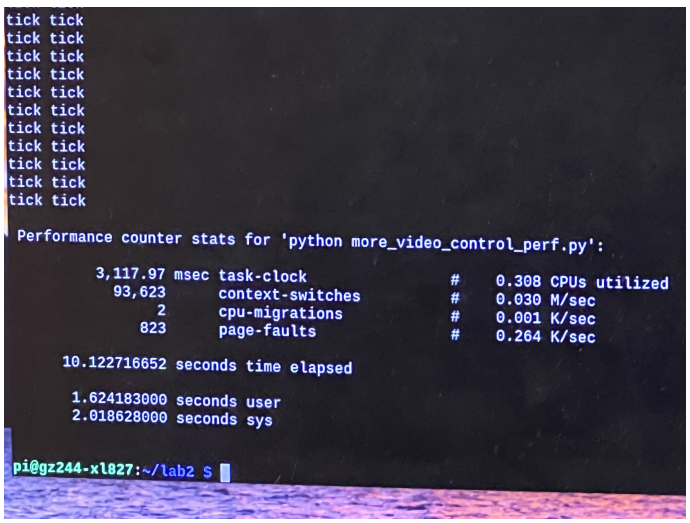


FIGURE 4: Performance measurement of video_control.py (2 milliseconds)

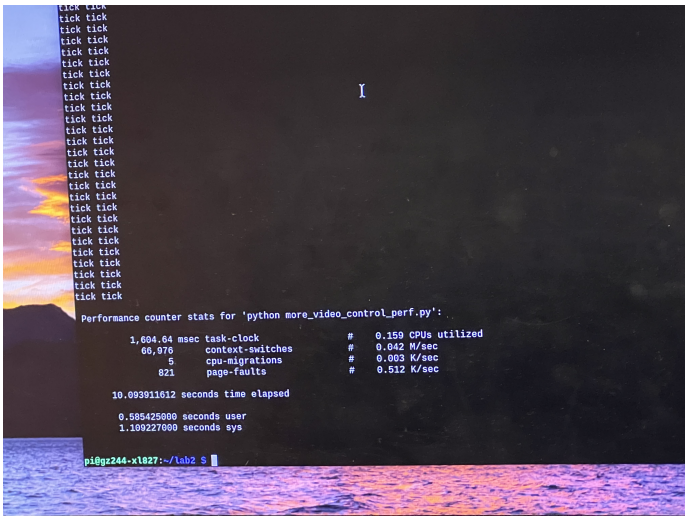


FIGURE 5: Performance measurement of video_control.py (200 microseconds)

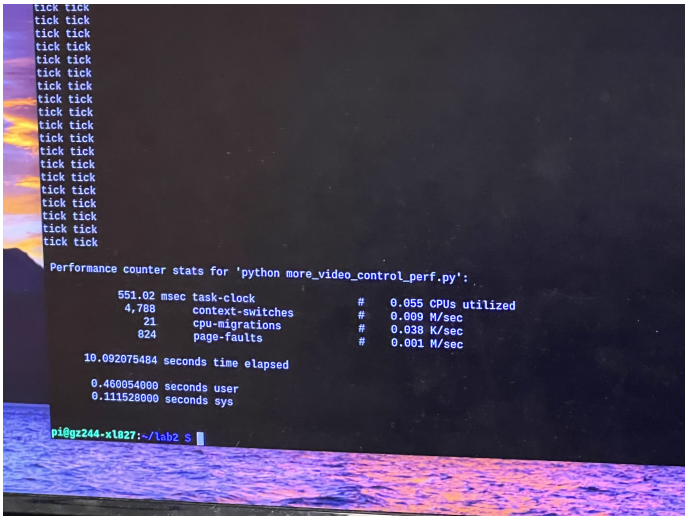


FIGURE 6: Performance measurement of video_control.py (20 microseconds)

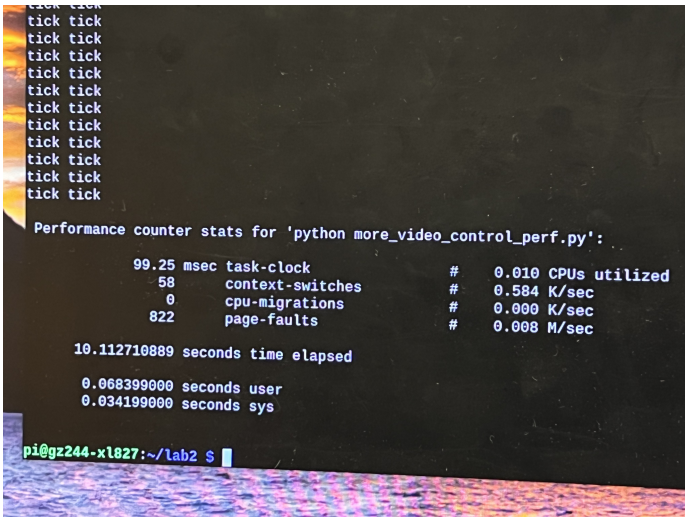


FIGURE 7: Performance measurement of video_control.py (no sleep at all)

2.4 PyGame: Bounce Program

I believe PyGame is the most challenging part during this lab and it truly consumed us loads of time and made us get into troubles. We firstly copy and paste the physics laws code of one ball and finshed `two_bounce.py`. Then we tried to duplicate the similar bouncing rules to colliding rules. However, different from the ball hitting the barrier (that just considering the ball reaches to the barrier locations), we need to figure out the statement if two balls collide into each other.

This first difficulty was helped by the professor that we were told to used a PyGame given function called `colliderect`. This function simply solved our issue of judging the timing of two balls collide.

Then came our second trouble: the image of balls have some invisible colliding volume so that they will have some distance apart from the screen at the collision. We solved this problem by using `inflate` function to decrease its invisible volume.

The third problem we met here is that our two balls starts at the same locations with same moving direction so that it is hard for them to simulate multiple collision to test our code. We checked the online PyGame functions list and change `ballrect.center` to another point.

The next problem we encountered is our two balls were too big so when they are displayed in the small PiTFT screen, it is difficult to display their movements. Thanks to the TAs, we found the `transform.scale` function to shrink their size by scales.

The final problem we solved is about the ball speed. Initially we didn't include a sleep function inside the while running loop, causing our balls moving so fast and cannot control the speed. After hours of debugging, we successfully understood every details of this collide code and passed the demo.

2.5 Fix Wheezy For Touch Screen Operation

In this section, we followed the lab manual to enable the Wheezy package and set it as default source with higher priority than Buster package for libsdl. Several files were created during the process. After saving these changes, we installed the changes and rebooted the system. Now the touch screen is ready for use.

2.6 Implementation of Quit Button On Screen

In this section, we wrote a python program named quit_button.py that displays a button named quit at the lower left corner of the screen that would quit the program and go back to console screen once touched.

We used the code from lab2 (two_collide.py) for the basic setup of screen and GPIO. Then, we looked at the example code for enabling both start and quit button as a template.

The major part of this section is learn to use multiple text button functions, such as setting font using `pygame.font.Font()`, setting text color using `.render()`, setting shape using `.get_rect()`. We can still use `screen.blit()` to display the button from workspace to the screen. Then in the `while` loop, we used `event.get()` and `event.type()` to detect mouse clicking. And after mouse is released, we want to know where the mouse is clicking by `pygame.mouse.get_pos()`, then by evaluating whether the received coordinate is within the coordinate range of our button, we would know if our button is pressed. In addition, we used button 17 on the piTFT as the "bail button" and used `time.time()` (set a start time and get current time to count the runtime) to enforce a time-out function to prevent excessive power on/off situations. During testing, we used a time-out time as 10 seconds, and then in demo we used 120 seconds. We added two more environment variables `SDL_MOUSEDRV` and `SDL_MOUSEDEV` and `pygame.mouse.set_visible()` command as instructed in the lecture to track the mouse on the screen. But we commented out these lines when we were first testing on monitor screen and uncommented them after proving monitor testing was successful and the program was ready to be tested on PiTFT.

We tested bail button and time-out in sequence and they worked well. Then we used `print()` method to print the mouse's button-up coordinates to refine the range of the button and updated the effective range of x, y under detection of `MOUSEBUTTONUP` event. In the end, the program worked well that it would only quit when we press within the region of the text (or time-out or pressing bail button) and would not do so when pressing anywhere else.

2.7 Implementation of Displaying Coordinates Touched On The Screen

In this section, we wrote a python program named as screencoordinates.py based on quit_button.py. This will print the current coordinates when mouse button is released to the screen.

This part is intuitively easy because we are just initializing a new text and updating its text during `MOUSEBUTTONDOWN` and `MOUSEBUTTONUP` events. Specifically, we created a local string `msg` and concatenated it as the text of the button and set its location at (160, 120).

After adding a new text at the center of the screen, we tried pressing different locations on the screen multiple times to see if the detected coordinates is correct. This was tested both on the monitor and the piTFT. However, since that the piTFT will update the coordinate each press (so it's hard to collect the coordinate message for 20 presses) and that the `print()` to console and the update of text on piTFT happens at the same locality, we will simply show a 20 coordinate message on the next page in Fig.8.

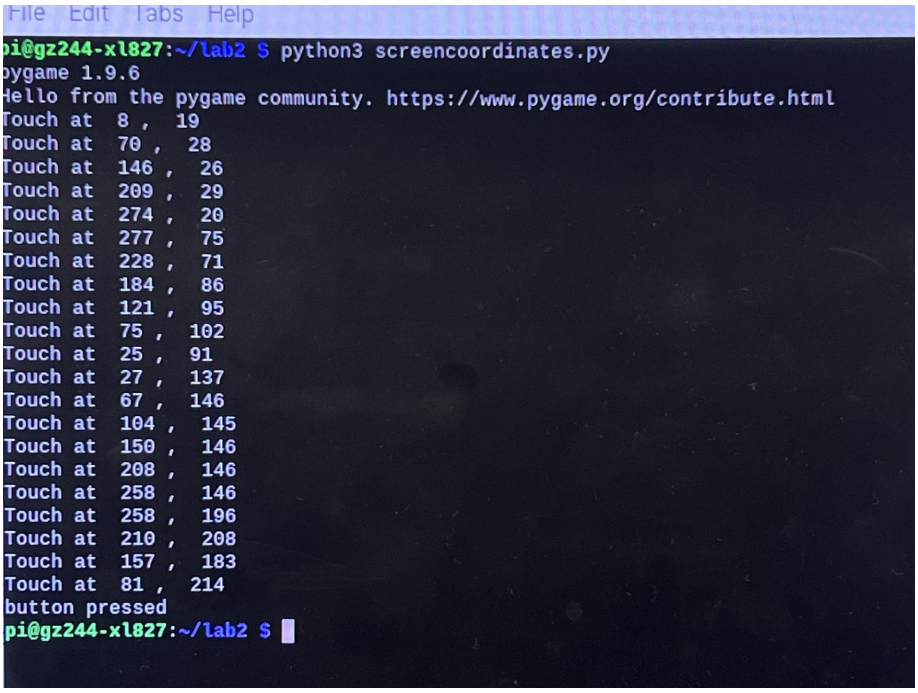


FIGURE 8: Coordinate Testing of 20 Touches On Console

2.8 Two-Button Touch Screen Pygame

In this section, we implemented a python program named as two_button.py based on screencoordinates.py and two_collide.py. The program would display two buttons, "start" and "quit" on the lower edge of the screen to control the game of two_collide: "start" will start the animation, and "quit" will quit the program, and pressing any places other than the text button will print its coordinates to the center of screen.

In this section, we are basically integrating the code from both file: screencoordinates.py which has the printing coordinate function and the two buttons, and the two_collide.py which has the animation. We first copy and pasted the initial setup for screen, balls, and texts (one button and the printing message). First, we initialized the "start" button together with "quit" but at a different location (now "start" is at (80, 120) and "quit" is at (240, 220)). Then, in the `while` loop, we need both the event detection of the mouse, the responding coordinate printing and checking, and the animation to update ball positions and the ball reaction to different boundary crossing or collision events. However, this simply putting in the code wouldn't allow mouse clicks to "trigger" the animation, what we did is that we created a global variable start_game that is initially set to 0. Then when in the `MOUSEBUTTONUP` event, if the click is within the range of the start button, we would set it to 1. Then, we put the entire chunk of ball animation code under the condition that the start game is true (equal to 1). In this way, only when the start button is clicked, the animation would play, and clicking quit would stop the entire program directly. The function of time out and bail button is kept as well. Last, we tested several times to refine the new effective range of the two buttons and the range of the animation so that the two wouldn't interfere with each other.

2.9 Two-Level Six-Button Touch Screen Pygame

In this section, we built a program named as Control_two_collide.py. It will have two pages where the first page is the same as screencoordinates.py, and once we click "start", it will go to the second page and start animation. On the second page, there are four buttons: "Pause/restart," "Fast," "Slow," "Back." "Pause" would simply stop the game on the first click and restart the game on the second click, Fast and slow will control the speed of the ball and back would stop the animation and go back to the first page.

To start, we used two_button.py as a template. Since we want to have two-level screens, we used the start_game variable to control the switching between the two screens. This is pretty intuitive, because when start_game=0 (false: game hasn't start), the screen is on level 1, which is the entire chunk of two button's click detection code that plays with "start" and "quit" button and the coordinates, and when start_game=1 (true: game starts), the screen is on level 2, which is the entire chunk of the animation code. With this intuition, we simply use start_game as a condition and add what needs to be displayed or erased under each condition using `screen.blit()`, `pygame.display.flip()`, and `screen.fill()` to control the switch of screen levels.

Then we went on to implement the four buttons on the second level. The initialization of these buttons are the same as before and how they should be displayed when start_game is 1 can be referred to other buttons as well.

The last thing is to implement how the balls/animation should respond to each of the four button clicks. So under the start_game=1, we also add the mouse click detection code, and we want to have all four react when `MOUSEBUTTONUP` is detected. For "pause" button, we added a global variable to count the number of clicks on the button, and when the button is clicked for odd times, the program will store the current speed to a global variable speed_tmp to be refered later and set both balls' speed to 0, while when the button is clicked for even times, the program will set the balls' speed to the previously stored speed_tmp. For "fast" and "slow", we used `time.sleep()` as a control since from previous week we were also using `time.sleep()` as a buffer to slow down the ball (since the system clock is super fast). We created a global variable time_wanna_sleep for changes. The time_wanna_sleep is set to 0.002 as default, a parameter that we tested from week1 and thought works reasonable. Then for "fast", we multiply time_wanna_sleep by a constant, and for "slow" we divide time_wanna_sleep by the same constant. Initially, we used 10 for the constant, but then during testing, we realized the speed would slow down so fast that the R-pi can crash (none of `esc`, or `ctrl+backspace` or `ctrl+x` would work). In the end, we used a constant of 2 which is much more reasonable. Last, for "back", we simply change start_game to finish the animation, erase the screen and display screen level 1.

In testing of these programs, we first ensured that the bail button and the time out would work with a small value on time out and then continued to test the real functions of programs. During the function test, we tested and implemented the code step by step and used a lot of `print()` method

to show the internal variable values and whether the output was as expected as the variable values suggests. And we tested all the programs firstly on the monitor with the environment variable commented and mouse visible and then uncommented the environment variables and set mouse visibility to false to test the program on piTFT. There were also some minor problems encountered, such as when to display the buttons, etc. And all issues were resolved quickly with the assistance of TAs. Luckily, none of the difficulties mentioned in the lecture, such as failure installing package, coordinates don't work, or cannot run on piTFT, occurred to us and we finished the tasks of week 2 early.

3. Conclusions

In this lab, we learned more about GPIO broad communication by reading the schematics and diagrams and connecting physical buttons to piTFT, further explored `mPlayer` video playing using external buttons and `perf`, and had a lot of experience playing with `Pygame`. This lab is intellectually stimulating and intriguing.

In week 1, we had to spend some extra time because we had many bugs with two_collide.py as discussed in 2.4. Problems with ball size, shape (the image is an rectangle so during collide, the ball wouldn't look like it touches on the screen) were solved after looking into `Pygame` libraries, Ball speed problem was solved by adding in a `sleep` function. In week 2, our main struggle was how to create a button or edit the text for coordinates, and when to erase the screen and redraw the buttons. This problem was solved after we consulted TAs and understood the example code on initializing and displaying buttons.

One interesting we find is that R-pi has a text editor that allows us to directly access the python file to edit and compile rather than using `vim` or `nano` which are more complex and inconvenient. However, when we want to run the program on piTFT, this would cause the R-pi to crash because this command requires administrator permissions, so we still have to execute the files in the terminal with `sudo`.