Lab 2: PyGame and Touch Control

By Alec Newport [acn55] and Eldor Bekpulatov [eb654]
Thursday Lab - Section 403
October 10, 2019

Introduction

This lab was designed to get us familiar with setting up and registering external GPIO inputs; using perf, a performance analysis tool; understanding the benefits of interrupts and callbacks over polling for GPIO inputs; using PyGame, a Python gaming and GUI library; configuring the TFT touch screen; and integrating touch events with a GUI.

Lab consisted of five parts. The first part of the lab primarily focused on configuring the external hardware to properly and safely use GPIO pins as inputs in our Raspberry Pi. The second part consisted of modifying the way we register external inputs from polling to interrupts. The third part of the lab consisted of installing software tool to help us with performance analysis of our programs and evaluate the performance gains of interrupt handlers using the modified scripts from the previous part. The fourth part of the lab was to develop a GUI using PyGame library. The last part of the lab included setting up OS environment to enable touch screen input from piTFT and writing python scripts that employed PyGame, external GPIO inputs, and touch screen inputs to control the GUI developed in the last part of the lab. Overall, this lab was nicely aligned to build on top of the previous lab and also incorporate new features.

Design and Testing

Part 1

This part of the lab primarily focused on configuring the external hardware to properly and safely use GPIO pins as inputs in our Raspberry Pi. This part was mostly a step-by-step instructions on how to set up external buttons to interact with our pi. To get started we needed to use what we had already done in the last lab and add two more buttons to the four button video playback control script. The GPIO pins simply read a digital value, and depending on our code can either be active high or low. We wrote code to be active low, so we used the provided diagram with a pull-up resistor and the button connected to ground (Figure 1).

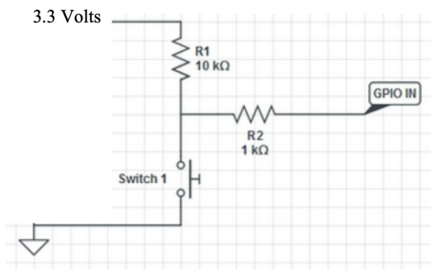


Figure 1. Pull-down diagram

As shown in the diagram above, closing the switch shorts the pin to GND (through the current-limiting resistor), and the GPIO will read 0. For the two-button circuit, each button was connected in the manner shown in Figure 1. Figure 2 shows our complete hardware setup. We connected our buttons to pin numbers 19 and 26. Figure 3 shows our initialization script for the two additional buttons. Since we used external pull-up resistors for these two, we didn't need to enable the internal pull-ups on their GPIO pins.

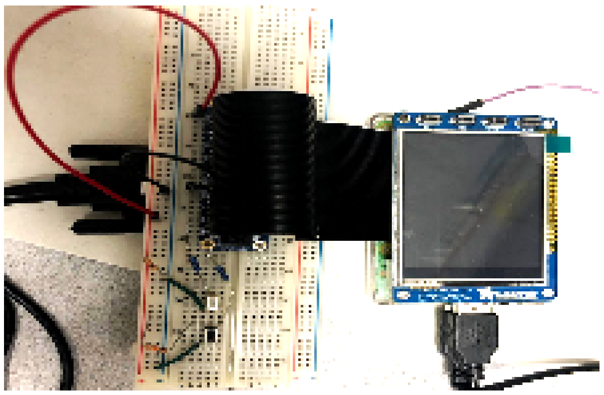
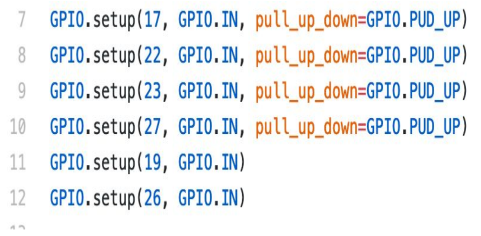


Figure 2. Pull-down setup for external buttons

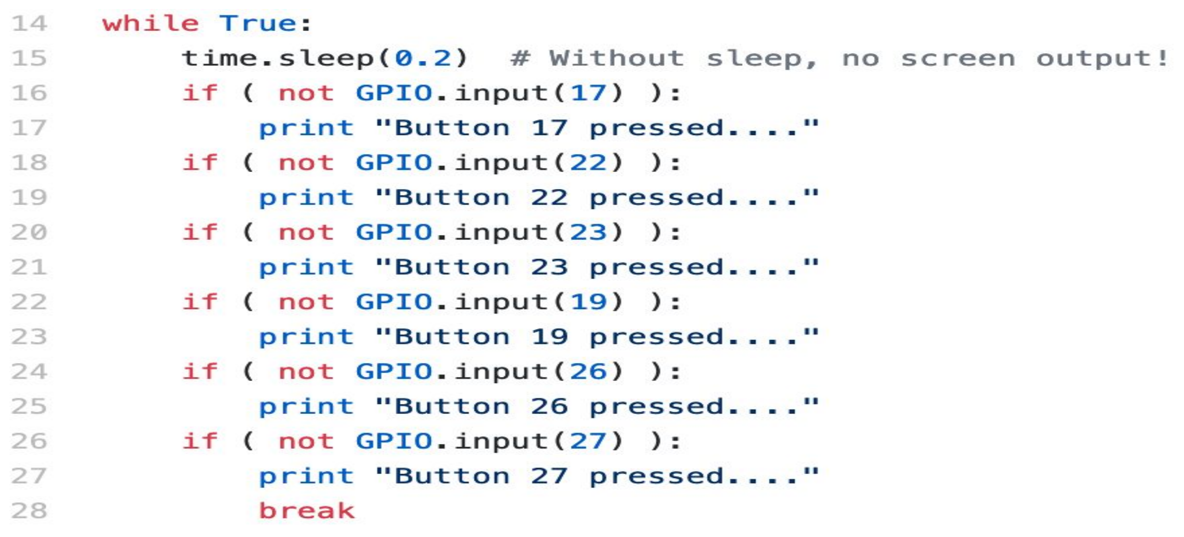


```
GPIO.setup(17, GPIO.IN, pull_up_down=GPIO.PUD_UP)
GPIO.setup(22, GPIO.IN, pull_up_down=GPIO.PUD_UP)
GPIO.setup(23, GPIO.IN, pull_up_down=GPIO.PUD_UP)
GPIO.setup(27, GPIO.IN, pull_up_down=GPIO.PUD_UP)
GPIO.setup(19, GPIO.IN)
GPIO.setup(26, GPIO.IN)
```

Figure 3. GPIO setup script

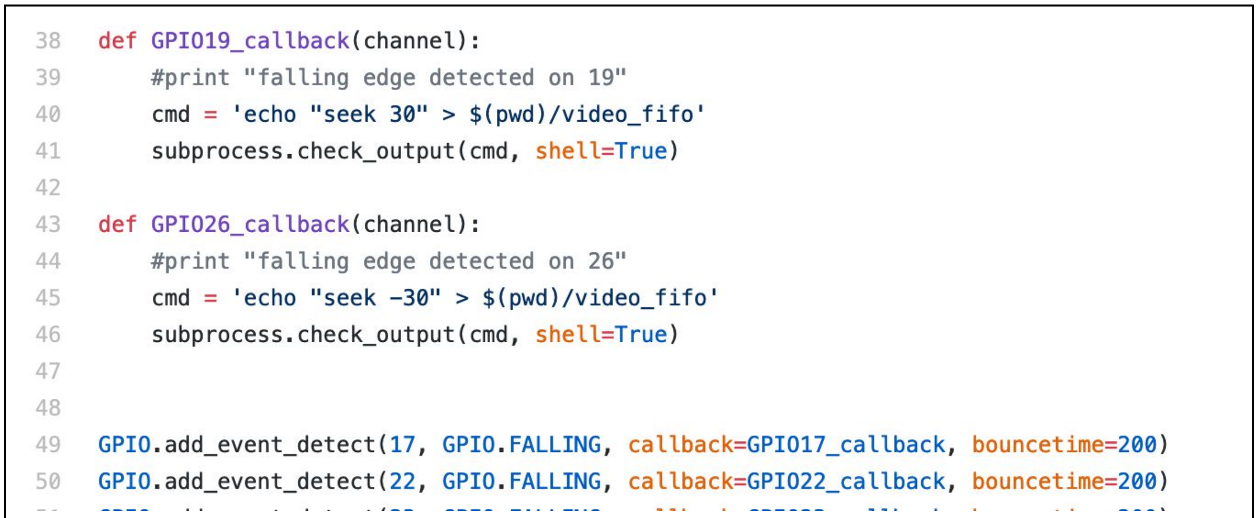
Part 2

This part of the lab consisted of modifying the way we register external inputs from polling to interrupts. In the previous lab, we used polling to register inputs from external buttons. Figure 4 shows the polling script. Since this wastes CPU cycles, we switched to using interrupts and callback routines to recognize and deal with button presses.



```
while True:
    time.sleep(0.2)  # Without sleep, no screen output!
    if ( not GPIO.input(17) ):
        print "Button 17 pressed...."
    if ( not GPIO.input(22) ):
        print "Button 22 pressed...."
    if ( not GPIO.input(23) ):
        print "Button 23 pressed...."
    if ( not GPIO.input(19) ):
        print "Button 19 pressed...."
    if ( not GPIO.input(26) ):
        print "Button 26 pressed...."
    if ( not GPIO.input(27) ):
        print "Button 27 pressed...."
        break
```

Figure 4. Polling script



```
def GPIO19_callback(channel):
    #print "falling edge detected on 19"
    cmd = 'echo "seek 30" > $(pwd)/video_fifo'
    subprocess.check_output(cmd, shell=True)

def GPIO26_callback(channel):
    #print "falling edge detected on 26"
    cmd = 'echo "seek -30" > $(pwd)/video_fifo'
    subprocess.check_output(cmd, shell=True)


GPIO.add_event_detect(17, GPIO.FALLING, callback=GPIO17_callback, bouncetime=200)
GPIO.add_event_detect(22, GPIO.FALLING, callback=GPIO22_callback, bouncetime=200)
```

Figure 5. Callback script

Figure 5 shows the script that handles inputs using callback routines. The parameters for the function add_event_detect specify the GPIO pin, the edge that will trigger the callback (in this case, the falling edge), the name of the callback function, and the length of time to wait between registering edges to debounce the signal. Using this idea, we were able to update the script controlling video playback in mplayer with buttons to use callbacks instead of a polling loop.

Part 3

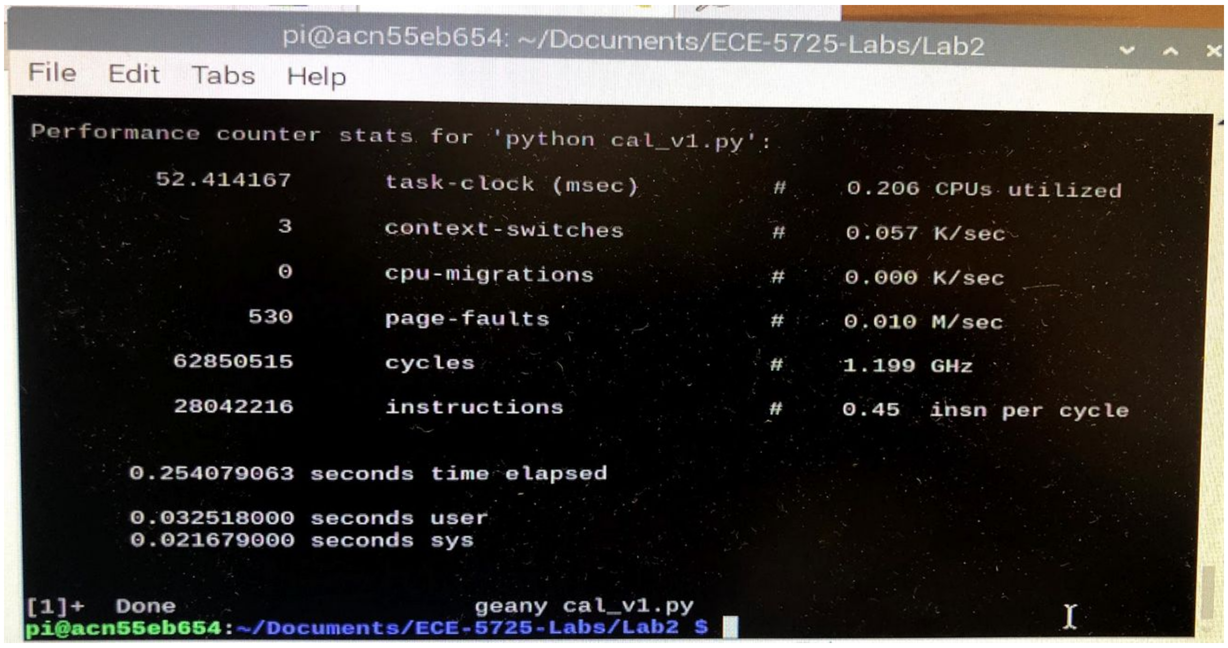
This part of the lab consisted of installing the software tool called perf to help us with the analysis of our programs and evaluate the performance gains of interrupt handlers using the modified scripts from the previous part. We know that using a callback routine saves a lot of CPU cycles. It does not use cpu cycles during the wait time. Polling requires an active process to check each independent GPIO pin intermittently thus not allowing it to scale well with increasing number of external sources.

To install perf we ran:

```
sudo apt-get install linux-perf-4.18
```

To verify that we can use perf, we created a python script called cal_v1.py which simply slept for 0.2 seconds. We ran the perf tool with our python script using the following command:

```
sudo perf_4.18 stat -e task-clock, context-switches,
cpu-migrations, page-faults, cycles, instructions python
cal_v1.py
```

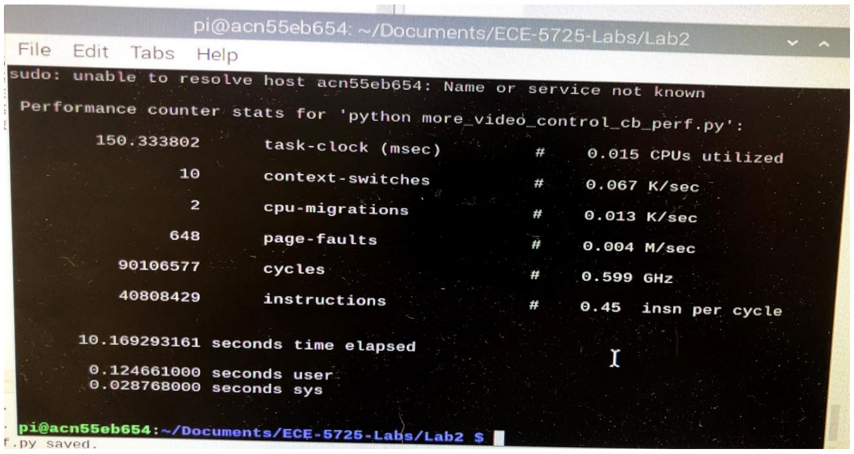


```
Performance counter stats for 'python cal_v1.py':

        52.414167      task-clock (msec)         #    0.206 CPUs utilized
                3      context-switches          #    0.057 K/sec
                0      cpu-migrations            #    0.000 K/sec
              530      page-faults               #    0.010 M/sec
         62850515      cycles                    #    1.199 GHz
         28042216      instructions              #    0.45  insn per cycle

      0.254079063 seconds time elapsed

      0.032518000 seconds user
      0.021679000 seconds sys

[1]+  Done                    geany cal_v1.py
pi@acn55eb654:~/Documents/ECE-5725-Labs/Lab2 $ 
```

Figure 6. Perf output for cal_v1.py

Now that we had confirmed that the tool was working (Figure 6), it was time for us to analyze and compare the performance of the scripts handling external inputs from the previous part.

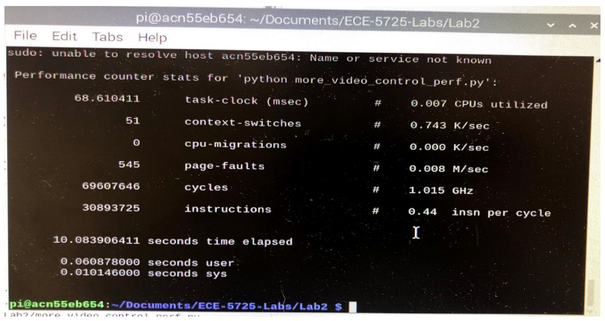
We ran one perf call on the callback routine version, and 6 perf calls on the polling version of the video control buttons. Figure 7 shows the callback routine script running with perf.



```
sudo: unable to resolve host acn55eb654: Name or service not known
 Performance counter stats for 'python more_video_control_cb_perf.py':

        150.333802      task-clock (msec)         #    0.015 CPUs utilized
                10      context-switches          #    0.067 K/sec
                 2      cpu-migrations            #    0.013 K/sec
               648      page-faults               #    0.004 M/sec
          90106577      cycles                    #    0.599 GHz
          40808429      instructions              #    0.45  insn per cycle

      10.169293161 seconds time elapsed

       0.124661000 seconds user
       0.028768000 seconds sys

pi@acn55eb654:~/Documents/ECE-5725-Labs/Lab2 $ 
f.py saved.
```

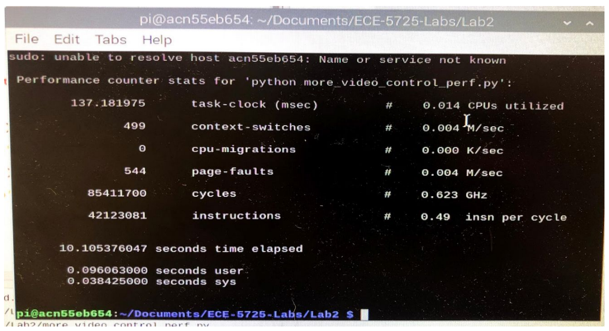
Figure 7. Perf output for callback routine

The reason for the 6 perf calls on polling is that we wanted to know what happens if we increase the frequency of polls and the effect on the performance. As you can tell, increasing the frequency of polls will definitely affect the performance negatively. Figures 8-13 show the perf outputs for polling routine happening in every 200ms, 20ms, 2ms, 0.2ms, 0.02ms, and 0ms respectively. Note that by reducing the amount of time in between each poll, CPU utilization increases dramatically until all of the CPU is utilized (Figure 13).



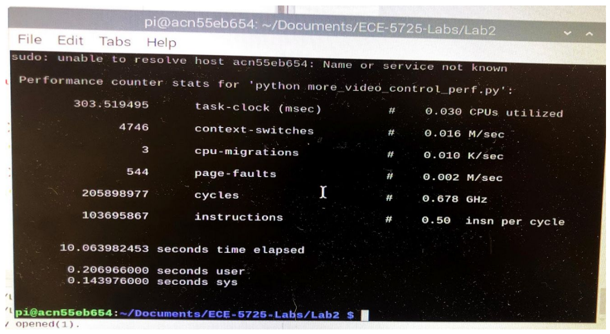
```
pi@acn55eb654: ~/Documents/ECE-5725-Labs/Lab2
File  Edit  Tabs  Help
sudo: unable to resolve host acn55eb654: Name or service not known
 Performance counter stats for 'python more_video_control_perf.py':

        68.610411      task-clock (msec)         #    0.007 CPUs utilized
               51      context-switches          #    0.743 K/sec
                0      cpu-migrations            #    0.000 K/sec
              545      page-faults               #    0.008 M/sec
         69607646      cycles                    #    1.015 GHz
         30893725      instructions              #    0.44  insn per cycle

     10.083906411 seconds time elapsed

      0.060878000 seconds user
      0.010146000 seconds sys

pi@acn55eb654:~/Documents/ECE-5725-Labs/Lab2 $
```

Figure 8. Poll every 200ms



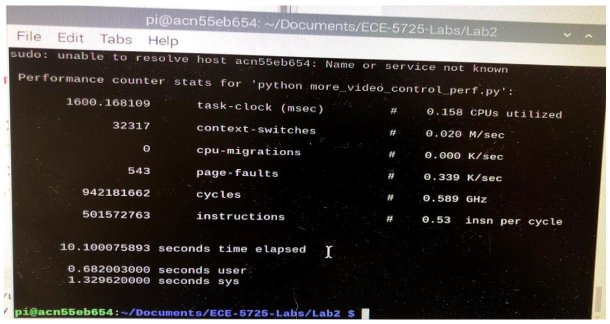
```
pi@acn55eb654: ~/Documents/ECE-5725-Labs/Lab2
File  Edit  Tabs  Help
sudo: unable to resolve host acn55eb654: Name or service not known
 Performance counter stats for 'python more_video_control_perf.py':

       137.181975      task-clock (msec)         #    0.014 CPUs utilized
              499      context-switches          #    0.004 M/sec
                0      cpu-migrations            #    0.000 K/sec
              544      page-faults               #    0.004 M/sec
         85411700      cycles                    #    0.623 GHz
         42123081      instructions              #    0.49  insn per cycle

     10.105376047 seconds time elapsed

      0.096063000 seconds user
      0.038425000 seconds sys

pi@acn55eb654:~/Documents/ECE-5725-Labs/Lab2 $
```

Figure 9. Poll every 20ms



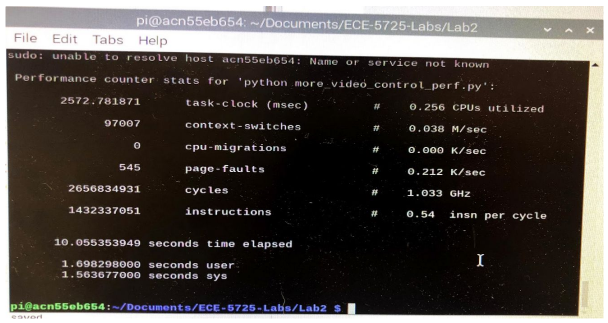
```
pi@acn55eb654: ~/Documents/ECE-5725-Labs/Lab2
File  Edit  Tabs  Help
sudo: unable to resolve host acn55eb654: Name or service not known
 Performance counter stats for 'python more_video_control_perf.py':

       303.519495      task-clock (msec)         #    0.030 CPUs utilized
             4746      context-switches          #    0.016 M/sec
                3      cpu-migrations            #    0.010 K/sec
              544      page-faults               #    0.002 M/sec
        205898977      cycles                    #    0.678 GHz
        103695867      instructions              #    0.50  insn per cycle

     10.063982453 seconds time elapsed

      0.206966000 seconds user
      0.143976000 seconds sys

pi@acn55eb654:~/Documents/ECE-5725-Labs/Lab2 $
```

Figure 10. Poll every 2ms



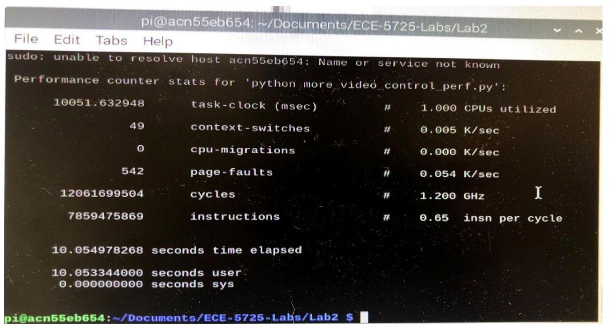
```
pi@acn55eb654: ~/Documents/ECE-5725-Labs/Lab2
File  Edit  Tabs  Help
sudo: unable to resolve host acn55eb654: Name or service not known
 Performance counter stats for 'python more_video_control_perf.py':

      1600.168109      task-clock (msec)         #    0.158 CPUs utilized
            32317      context-switches          #    0.020 M/sec
                0      cpu-migrations            #    0.000 K/sec
              543      page-faults               #    0.339 K/sec
        942181662      cycles                    #    0.589 GHz
        501572763      instructions              #    0.53  insn per cycle

     10.100075893 seconds time elapsed

      0.682003000 seconds user
      1.329620000 seconds sys

pi@acn55eb654:~/Documents/ECE-5725-Labs/Lab2 $
```

Figure 11. Poll every 0.2ms



```
pi@acn55eb654: ~/Documents/ECE-5725-Labs/Lab2
File  Edit  Tabs  Help
sudo: unable to resolve host acn55eb654: Name or service not known
 Performance counter stats for 'python more_video_control_perf.py':

      2572.781871      task-clock (msec)         #    0.256 CPUs utilized
            97007      context-switches          #    0.038 M/sec
                0      cpu-migrations            #    0.000 K/sec
              545      page-faults               #    0.212 K/sec
       2656834931      cycles                    #    1.033 GHz
       1432337051      instructions              #    0.54  insn per cycle

     10.055353949 seconds time elapsed

      1.698298000 seconds user
      1.563677000 seconds sys

pi@acn55eb654:~/Documents/ECE-5725-Labs/Lab2 $
```

Figure 12. Poll every 0.02ms



```
pi@acn55eb654: ~/Documents/ECE-5725-Labs/Lab2
File  Edit  Tabs  Help
sudo: unable to resolve host acn55eb654: Name or service not known
 Performance counter stats for 'python more_video_control_perf.py':

     10051.632948      task-clock (msec)         #    1.000 CPUs utilized
               49      context-switches          #    0.005 K/sec
                0      cpu-migrations            #    0.000 K/sec
              542      page-faults               #    0.054 K/sec
      12061699504      cycles                    #    1.200 GHz
       7859475869      instructions              #    0.65  insn per cycle

     10.054978268 seconds time elapsed

     10.053344000 seconds user
      0.000000000 seconds sys

pi@acn55eb654:~/Documents/ECE-5725-Labs/Lab2 $
```

Figure 13. Continuous polling

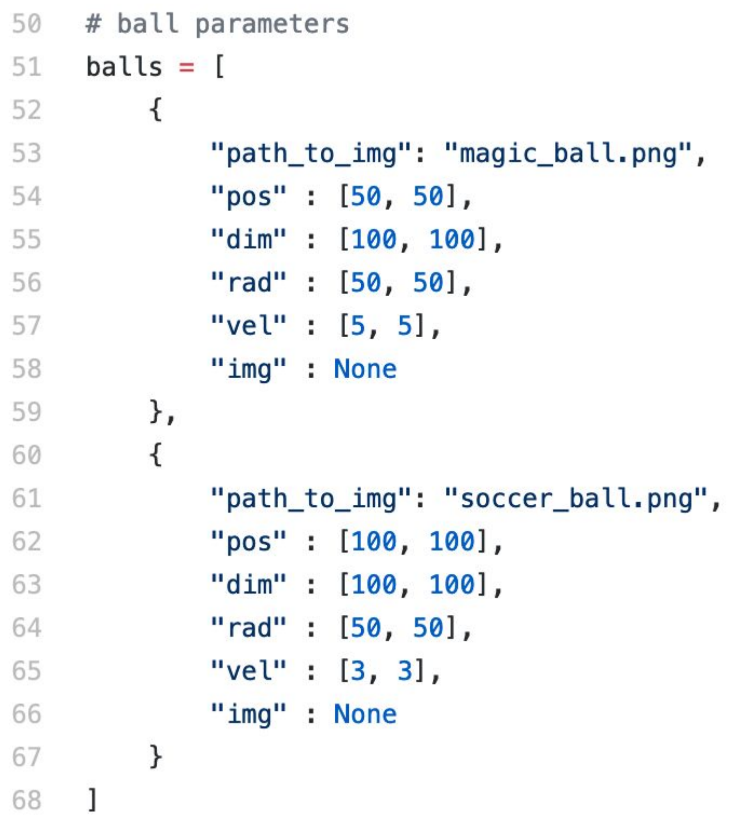
Part 4

The fourth part of the lab was to develop a GUI using PyGame library. Here we familiarized ourselves with PyGame semantics and developed simple scripts that animated a given image around the piTFT screen by making the image bounce around the walls.

We were instructed to write three scripts: bounce, in which an image is loaded to the screen, moves around, and bounces when it hits the edges of the screen; two_bounce, in which two images are loaded, move around, and bounce against the edges of the screen but not each other; and two_collide, in which the two images bounce off each other as well as the edges of the screen.

Bounce.py was a relatively easy task since most of the code was given to us in lectures. We learned the semantics of loading an image using PyGame wrapper classes. We animated the movement of the balls using condition (while) loops and have a callback routine that breaks out of the loop to simulate a quit ("bail out") button.

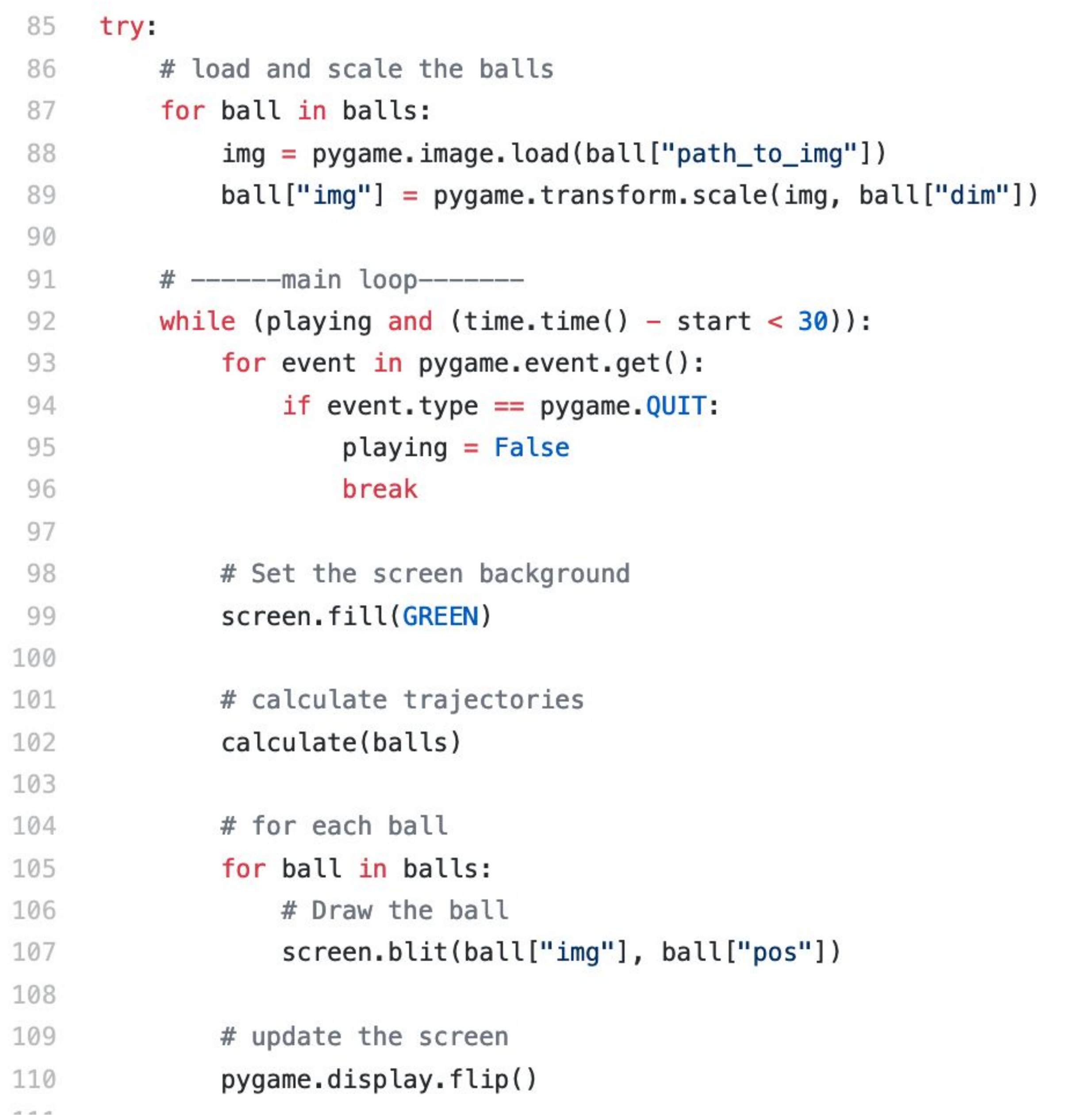
Two_bounce.py was also easy, but now that we had two balls to animate, we thought it would be a good idea to group the ball attributes like image, position, radius, dimension, and a velocity in a single grouping. Figure 14 shows the data structure derived to represent the array of balls to animate on screen.



```
# ball parameters
balls = [
    {
        "path_to_img": "magic_ball.png",
        "pos" : [50, 50],
        "dim" : [100, 100],
        "rad" : [50, 50],
        "vel" : [5, 5],
        "img" : None
    },
    {
        "path_to_img": "soccer_ball.png",
        "pos" : [100, 100],
        "dim" : [100, 100],
        "rad" : [50, 50],
        "vel" : [3, 3],
        "img" : None
    }
]
```

Figure 14. Data structure to implement array of balls

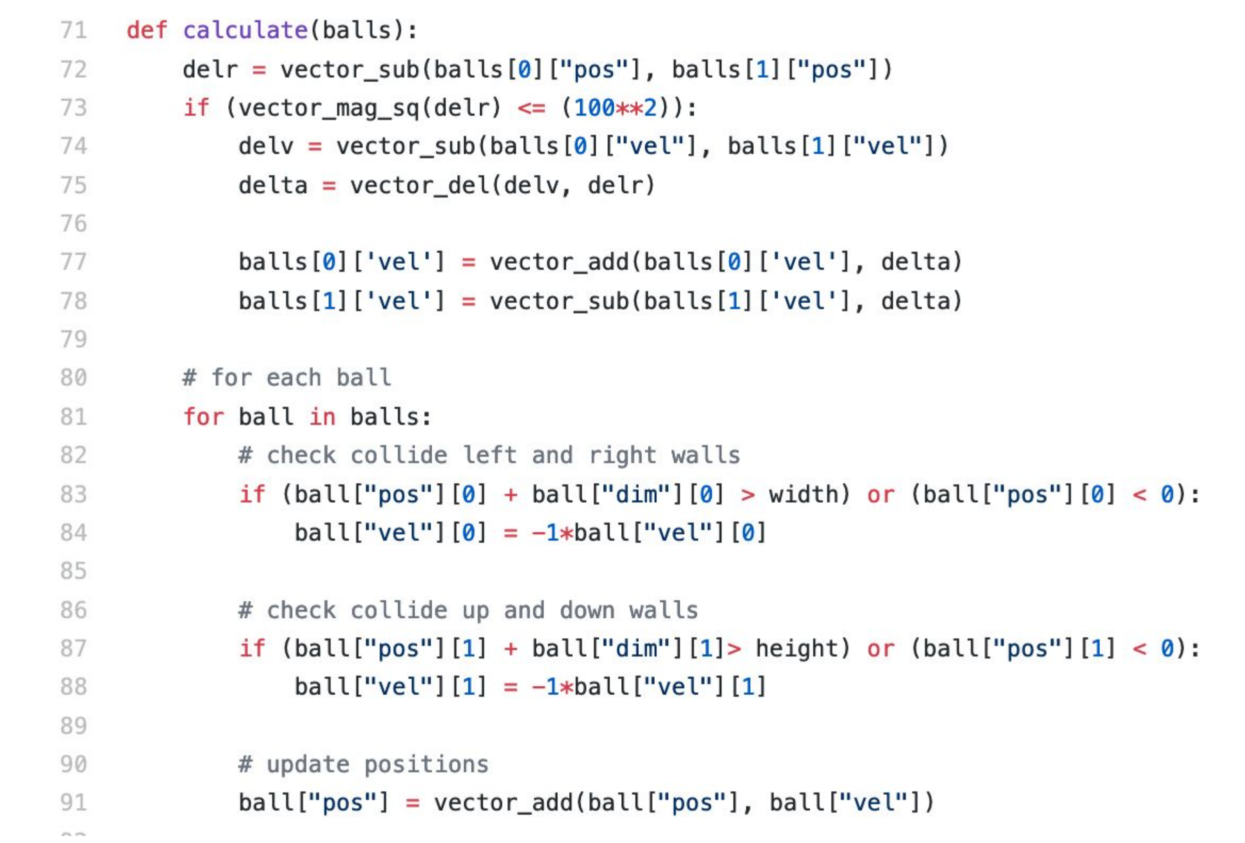
Two_collide.py was relatively complex, because now we had to simulate collisions between the two balls. Having the data structure implemented in the previous part helped significantly simplify our code. Now we could follow the template below to separate the collision detection functionality separately. Figure 15 shows the animation template where comments are provided to explain the code.



```python
try:
    # load and scale the balls
    for ball in balls:
        img = pygame.image.load(ball["path_to_img"])
        ball["img"] = pygame.transform.scale(img, ball["dim"])

    # ------main loop-------
    while (playing and (time.time() - start < 30)):
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                playing = False
                break

        # Set the screen background
        screen.fill(GREEN)

        # calculate trajectories
        calculate(balls)

        # for each ball
        for ball in balls:
            # Draw the ball
            screen.blit(ball["img"], ball["pos"])

        # update the screen
        pygame.display.flip()
```

Figure 15. Template to animate the balls

Note that calculate(balls) is where we are doing the vector math and identifying any collision between balls or the walls. Figure 16 shows the full collision detection logic. We implemented this logic using the resources provided in the lab handout. To achieve a full functionality we developed a util.py file that defined most of the basic vector math helper functions.



```
def calculate(balls):
    delr = vector_sub(balls[0]["pos"], balls[1]["pos"])
    if (vector_mag_sq(delr) <= (100**2)):
        delv = vector_sub(balls[0]["vel"], balls[1]["vel"])
        delta = vector_del(delv, delr)

        balls[0]['vel'] = vector_add(balls[0]['vel'], delta)
        balls[1]['vel'] = vector_sub(balls[1]['vel'], delta)

    # for each ball
    for ball in balls:
        # check collide left and right walls
        if (ball["pos"][0] + ball["dim"][0] > width) or (ball["pos"][0] < 0):
            ball["vel"][0] = -1*ball["vel"][0]

        # check collide up and down walls
        if (ball["pos"][1] + ball["dim"][1]> height) or (ball["pos"][1] < 0):
            ball["vel"][1] = -1*ball["vel"][1]

        # update positions
        ball["pos"] = vector_add(ball["pos"], ball["vel"])
```

Figure 16. Collision logic

Part 5

The final part of this lab took place in week 2. The goal for this section was to integrate everything we had done so far with touch screen controls. To do this, we needed to install some new packages and drivers that allow mouse clicks to be received from the TFT.

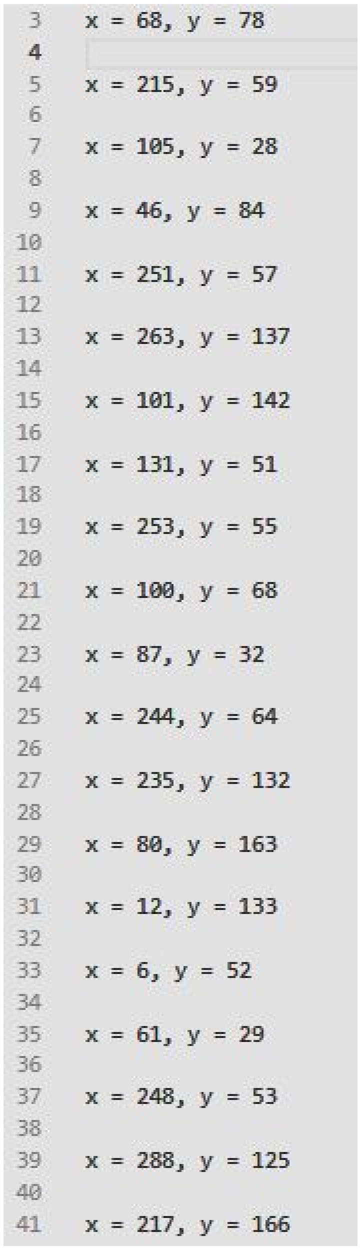
With those installed, the only change that needs to be made to the Python scripts to accept input from the TFT is adding the following five lines:

```
os.putenv('SDL_VIDEODRIVER', 'fbcon') # Display on piTFT
os.putenv('SDL_FBDEV', '/dev/fb1') #
os.putenv('SDL_MOUSEDRV', 'TSLIB') # Track mouse clicks on piTFT
os.putenv('SDL_MOUSEDEV', '/dev/input/touchscreen')
pygame.mouse.set_visible(False)
```

These environment variables change the video output to display on the TFT and then change the mouse input to come from the TFT as well. The final line removes the cursor from the screen (since touch screens usually do not employ a cursor).

The first script we wrote to test this new functionality was quit_button.py. This program simply spins with a quit button displayed in the lower left corner of the screen until a click is registered from the TFT in the area around the button. This input is treated the same as the external bail-out button, and the program terminates.

The next touch screen script we wrote, screen_coordinates.py, records the coordinates of each touch and prints them to the TFT screen, the Linux console, and a txt file. The txt file results of 20 such taps are shown in Figure 17 below. This code also contains an on-screen quit button as well as a physical bail-out button to terminate the program.



```
x = 68, y = 78

x = 215, y = 59

x = 105, y = 28

x = 46, y = 84

x = 251, y = 57

x = 263, y = 137

x = 101, y = 142

x = 131, y = 51

x = 253, y = 55

x = 100, y = 68

x = 87, y = 32

x = 244, y = 64

x = 235, y = 132

x = 80, y = 163

x = 12, y = 133

x = 6, y = 52

x = 61, y = 29

x = 248, y = 53

x = 288, y = 125

x = 217, y = 166
```

Figure 17. Screen clicks

Finally, we wrote two scripts that control execution of the two_collide.py program we wrote earlier: two_button.py and control_two_collide.py. Both employ a two-level execution scheme as well as an external bail-out button.

In the first level, two_collide has not started running, and only Start and Quit buttons are shown (Figure 18). If the user clicks on Start, two_collide starts running. If they click on Quit, the program terminates. If the click is not on one of these two buttons, the coordinates of the click are displayed.

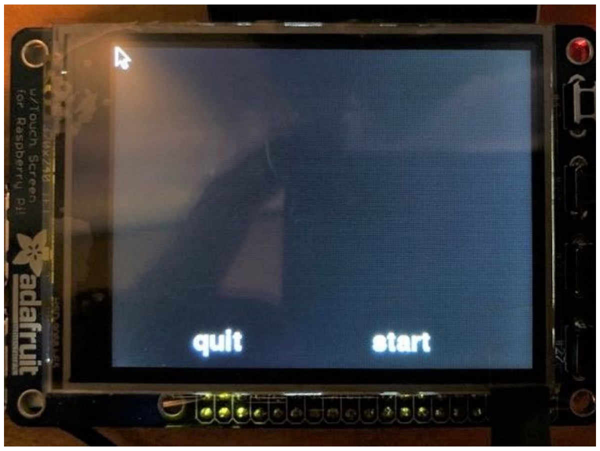


Figure 18. Level 1 menu

The second level is where the two programs differ. In two_button, the same two buttons are displayed as in the first level: Start and Quit. Start doesn't do anything in this case, and Quit terminates the program as usual. In control_two_collide, however, four new buttons are displayed: Pause, which pauses the animation but stays in the second execution level; Fast, which increases the frame rate; Slow, which decreases the frame rate; and Back, which returns execution to the first level (Figure 19). When Pause is pressed, it changes to Resume, which restarts the animation. All other buttons retain functionality when the animation is paused. There is also a maximum and minimum frame rate set, so if the user presses Fast or Slow when at the maximum or minimum frame rate respectively, a warning is displayed for two seconds, and the framerate doesn't change (Figures 20-21).

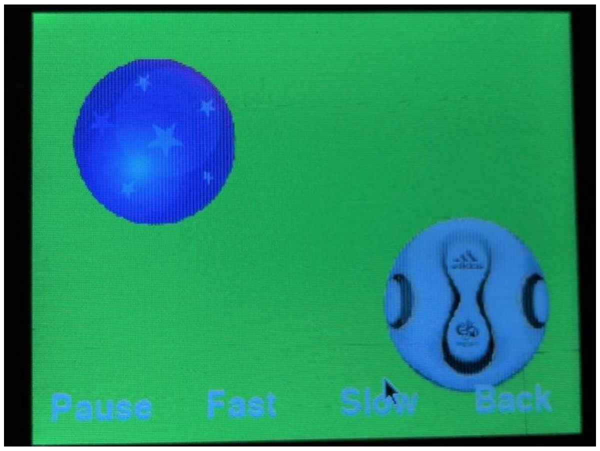
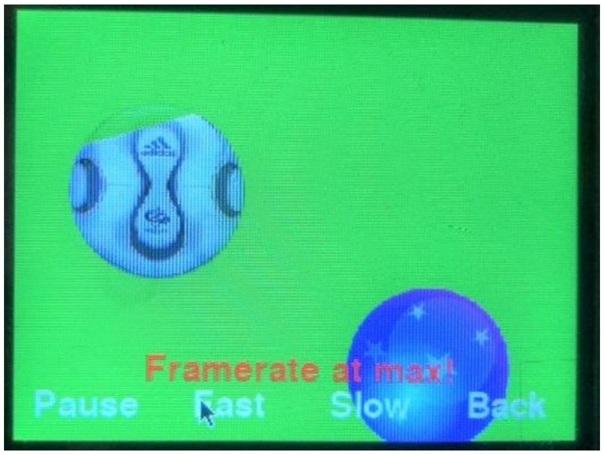
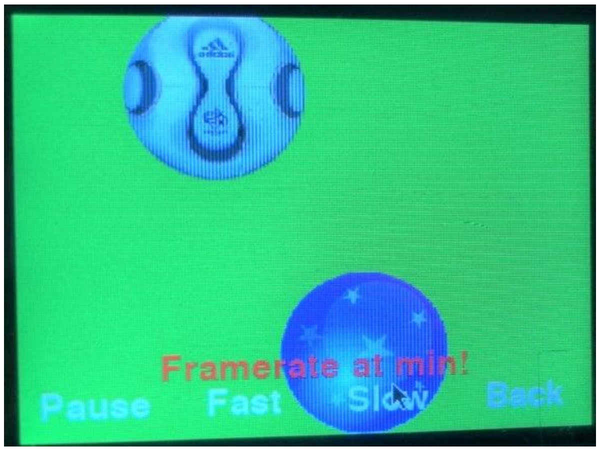


Figure 16. Level 2 menu

 Figure 20: Max frame rate warning	 Figure 21: Min frame rate warning

Conclusions

We found this lab to be fairly straightforward. We both had experience with GPIO, interrupts vs polling, and simple animation, so we didn't have any difficulties. The only clarification to add would be whether the timeout was necessary for the demo. From the lab handout, we thought timeouts were only for testing, but the TAs wanted to see them in our demos. It was trivial to implement, but maybe a clarifying point in the lab that this should be demoed would be appreciated.

Code

Code stored on server at /home/Lab2/acn55_eb654_Lab2