Lab 2

Lab Section: Monday

October 10, 2020

Jonathan Nusantara, Eric Kahn,
Eric Hall
jan265, edk52, ewh73

Introduction

The main objectives of this lab are to have us become more familiar with GPIO PiTFT, and at the same time introduce new topics like performance, PyGame, callback interrupts on the Raspberry Pi and the touchscreen capabilities in PiTFT. We will accomplish this by adding external buttons to Raspberry Pi where we will experiment in physically setting up as pull-up or pull-down resistors instead of doing it through software. The introduction of callback interrupts also leads us to comparing the performance of it to polling loops, where we will be using the “perf” tool to compare the examined statistics. PyGame will be explored by creating a GUI and animations, as well as integrating it along with the touchscreen capabilities of the PiTFT screen.

In week 2, we will be focusing on creating an “embedded-like system” by starting scripts from the terminal, and then interacting with it through GPIO using the PiTFT buttons and screen. This is important as in future labs we will be designing and building a robot that should be able to move around without being connected to a monitor.

Design and Testing

Week 1:

Setting up more buttons:

This lab starts with implementing more buttons that can be used for the Raspberry Pi system. Currently four buttons are used form the PiTFT and two more buttons will be integrated. These two buttons will be set up externally on a breadboard and connected to the GPIO pins of Raspberry Pi. The buttons are set up based on this diagram below:

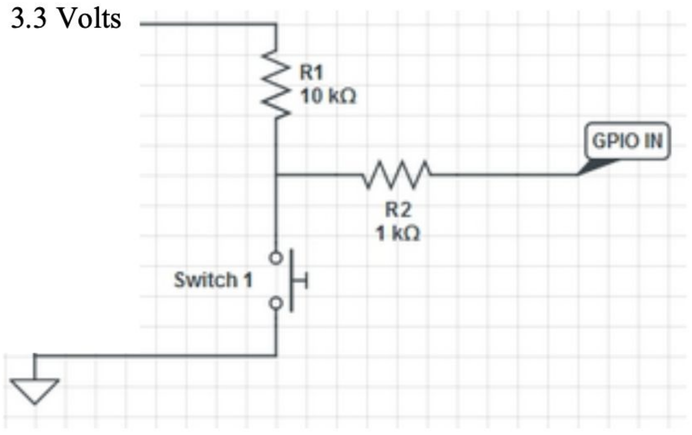


Figure 1 - Schematic diagram for button set up.

Based on the schematic diagram above, two buttons are set up and connected to the GPIO pins. The schematic diagram above shows the setup of the buttons with external pull-up resistors. This means that the GPIO input will detect a high when the button is not pressed, and detect a low when the button is pressed. R1 is the pull-up resistor while R2 is a resistor that is used to protect the GPIO input. R2 is used to prevent a sudden surge of current and to limit the current value when GPIO is accidentally set to output (which may cause the 3.3V to be shorted to low ground). The R2 is set to be 1k Ohms because it will limit the current to the acceptable amount of 3.3mA.

The PiCobbler breakout cable is first connected to the breadboard. After setting up the cables and power to the breadboard, a voltmeter is first used to make sure that we are using the right 3.3V and ground pins. The two buttons are then set up to be connected to GPIO 19 and 26, based on the Broadcom numbering system. The GPIO pins are being considered after referring to the GPIO pin diagram below. The result of the button connections on the breadboard is as below:

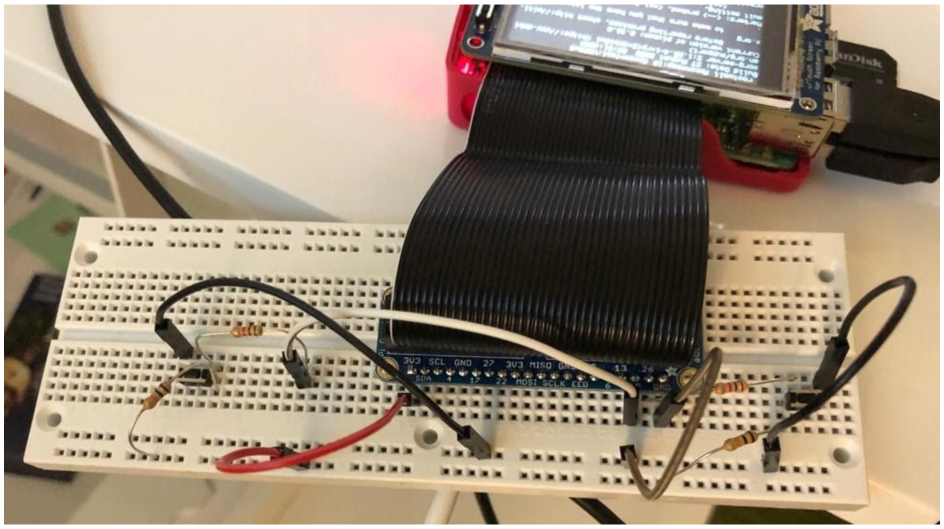
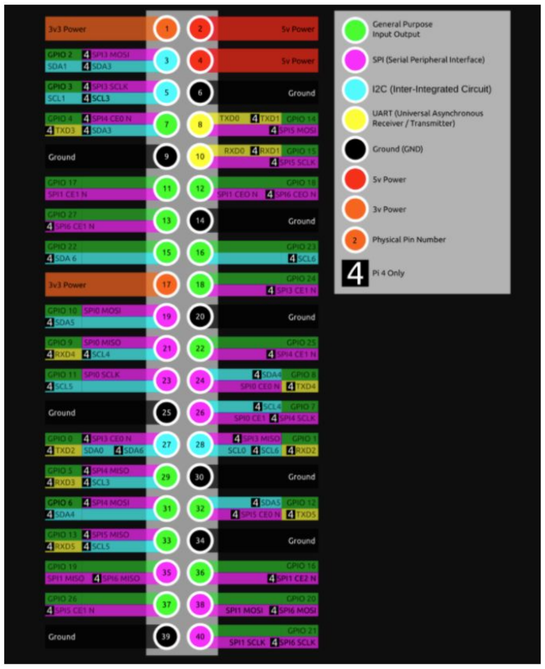


Figure 2 - Buttons setup on a breadboard connected to RaspberryPi.


General Purpose Input Output
SPI (Serial Peripheral Interface)
I2C (Inter-Integrated Circuit)
UART (Universal Asynchronous Receiver / Transmitter)
Ground (GND)
5v Power
3v Power
Physical Pin Number
Pi 4 Only
Figure 3 - GPIO pins diagram on Raspberry Pi 4.

Next, a python program called "`six_buttons.py`" is created based on the "`four_buttons.py`" program that was created in Lab 1. The setup for GPIO 19 and 26 were done in the program, as they are set to inputs and to have internal pull up resistors.

```
GPIO.setup(26, GPIO.IN, pull_up_down=GPIO.PUD_UP)
GPIO.setup(19, GPIO.IN, pull_up_down=GPIO.PUD_UP)
```

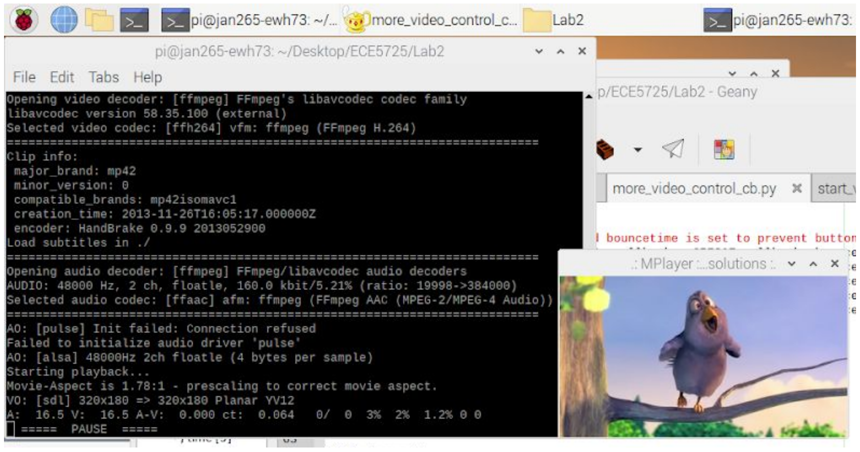
Although we technically do not need the internal pull-up resistor anymore, we did it anyways as it causes no harm. Next, in the infinite loop, GPIO 19 and 26 are added to be monitored, and a button press message will be printed if a button press is detected.

The program is then tested by running the program in terminal. It was verified that when one of the six buttons are pressed, a message stating which button is pressed is displayed in Terminal. One issue that Jonathan faced was in setting up one of the buttons. Part of the breadboard seemed to be failing, which caused the button to not be working. This issue was resolved when the ground pin was moved into another point on the rails.

This addition of two buttons is then implemented to the "`video_control.py`". The new python program is called "more_video_control.py". The only upgrade that this program has compared to "`video_control.py`" is to add the two extra buttons. These two buttons are added to the polling loop, and each of the buttons will now be able to either fast forward 30 seconds or rewind 30 seconds. The additional code inserted for fast forward 30 second is as follows:

```
send_command = 'echo seek 30 0 > video_fifo'
subprocess.check_output(send_command, shell = True)
```

The code above will create a string to print the mplayer command and send it to the fifo. This string will then be printed to the shell if the button is pressed. The "`./start_video`" bash script is also modified. It will now call the program "`more_video_control.py`" rather than "`video_control.py`".



```
Opening video decoder: [ffmpeg] FFmpeg's libavcodec codec family
libavcodec version 58.35.100 (external)
Selected video codec: [ffh264] vfm: ffmpeg (FFmpeg H.264)
==========================================================================
Clip info:
 major_brand: mp42
 minor_version: 0
 compatible_brands: mp42isomavc1
 creation_time: 2013-11-26T16:05:17.000000Z
 encoder: HandBrake 0.9.9 2013052900
Load subtitles in ./
==========================================================================
Opening audio decoder: [ffmpeg] FFmpeg/libavcodec audio decoders
AUDIO: 48000 Hz, 2 ch, floatle, 160.0 kbit/5.21% (ratio: 19998->384000)
Selected audio codec: [ffaac] afm: ffmpeg (FFmpeg AAC (MPEG-2/MPEG-4 Audio))
==========================================================================
AO: [pulse] Init failed: Connection refused
Failed to initialize audio driver 'pulse'
AO: [alsa] 48000Hz 2ch floatle (4 bytes per sample)
Starting playback...
Movie-Aspect is 1.78:1 - prescaling to correct movie aspect.
VO: [sdl] 320x180 => 320x180 Planar YV12
A:  16.5 V:  16.5 A-V:  0.000 ct:  0.064   0/  0  3%  2%  1.2% 0 0
 =====  PAUSE  =====
```

Figure 4 - "more_video_control.py" controlling mplayer video playback.

This change is then tested on our Raspberry Pi. In the terminal, "`./start_video`" was run. It will run the python program on the background to watch for button presses and then run mplayer with the video playback. All of the six buttons are tested and the expected output of operations on the video playback were achieved, verifying that this part of the lab works. This section is overall pretty straightforward with no major challenges.

Interrupts callbacks:

In this section, we are learning about callbacks. Our current button implementation is using an infinite polling loop which is very inefficient. Now, we will be modifying the button presses to use threaded callback interrupt routines. A new python program named "`more_video_control_cb.py`" is created based on "`more_video_control.py`". The only difference is that this program will be using callbacks to detect button presses.

In this program, the section of the code that set up the 6 GPIO pins as input are adopted. Next, a callback function for each GPIO (except for GPIO 27) is created as below:

```
def GPIO17_callback(channel):
        send_command = 'echo pause > video_fifo'
        subprocess.check_output(send_command, shell = True)
```

The example above is the callback function for GPIO 17. This function will be called every time button 17 is pressed. Inside the function are all the code that would like to be called when the button is pressed, and in this case it will be to pause the mplayer. The content in each button's function is customized to match what their action should be.

Next, event detections for each GPIO (except for GPIO 27, which will be explained later) are added to the code. This event detection is used to watch for events when the state of GPIO changes, in this case because of button presses, and then the callback function will be called to execute the related commands. An example (for GPIO 17) is as below:

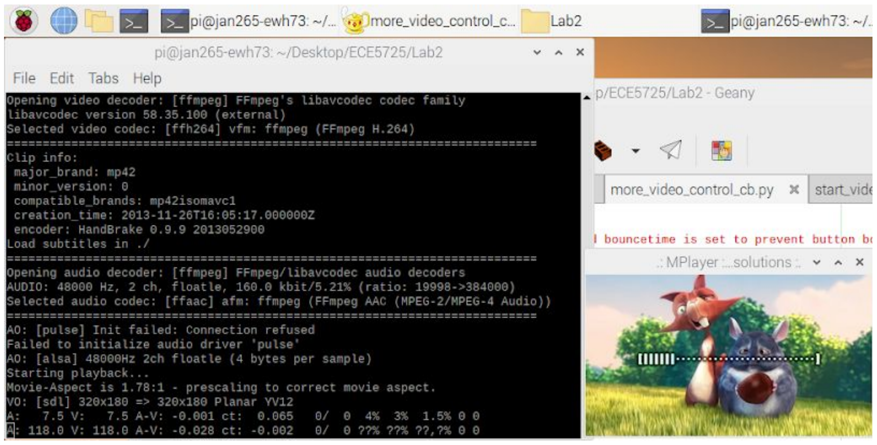
```
GPIO.add_event_detect(17, GPIO.FALLING, callback = GPIO17_callback,
bouncetime = 300)
```

In this example, GPIO 17 is being watched for a falling edge, from high to low, when the button is pressed. A callback is also defined to the callback function created for this button. The bouncetime is also set, which is to prevent the mechanical/electrical bouncing of the button. Similar to a "`time.sleep(300)`" used previously, this means that the program will not read any button press for the next 300ms.

The final change would be to add a "try and except" in the code. A snippet of the code is as below:

```
try: # Wait for button 27 to be pressed
       GPIO.wait_for_edge(27, GPIO.FALLING)
```

```
                send_command = 'echo quit > video_fifo'
                subprocess.check_output(send_command, shell = True)

        except KeyboardInterrupt:
                GPIO.cleanup() # GPIO clean when ctrl + c is passed
```
This code will execute the code in the "try", unless "`KeyboardInterrupt`" is passed. If no keyboard interrupt is passed, then a GPIO wait for edge function is called, and in this case it will wait for GPIO 27's falling edge, which is when button 27 is pressed. Then, it will send the quit command to mplayer via fifo, and the program will also close. At the very end, "`GPIO.cleanup()`" was added at the end of the file to clean all GPIO settings for the next run.



```
Opening video decoder: [ffmpeg] FFmpeg's libavcodec codec family
libavcodec version 58.35.100 (external)
Selected video codec: [ffh264] vfm: ffmpeg (FFmpeg H.264)
==========================================================================
Clip info:
 major_brand: mp42
 minor_version: 0
 compatible_brands: mp42isomavc1
 creation_time: 2013-11-26T16:05:17.000000Z
 encoder: HandBrake 0.9.9 2013052900
Load subtitles in ./
==========================================================================
Opening audio decoder: [ffmpeg] FFmpeg/libavcodec audio decoders
AUDIO: 48000 Hz, 2 ch, floatle, 160.0 kbit/5.21% (ratio: 19998->384000)
Selected audio codec: [ffaac] afm: ffmpeg (FFmpeg AAC (MPEG-2/MPEG-4 Audio))
==========================================================================
AO: [pulse] Init failed: Connection refused
Failed to initialize audio driver 'pulse'
AO: [alsa] 48000Hz 2ch floatle (4 bytes per sample)
Starting playback...
Movie-Aspect is 1.78:1 - prescaling to correct movie aspect.
VO: [sdl] 320x180 => 320x180 Planar YV12
A:   7.5 V:   7.5 A-V: -0.001 ct:  0.065   0/   0  4%  3%  1.5% 0 0
A: 118.0 V: 118.0 A-V: -0.028 ct: -0.002   0/   0 ??% ??% ??,?% 0 0
```

Figure 5 - "more_video_control_cb.py" controlling mplayer video playback.

Before testing a new program, a bash script called "`./start_video_cb`" is created based on "`./start_video`". However, python program "`more_video_control_cb.py`" will be called instead of "`more_video_control.py`". In terminal, the bash script is then called and mplayer is launched alongside the python program in the background. All of the buttons work in controlling the video playback. The user will have a similar experience to the previous polling loop program in running the video playback and inputting the different controls like pausing and playing the video. In this section, it is more of discovering the different functions that need to be created. One confusing aspect is on how to quit after button 27 is pressed. We ended up going with the "`wait_for_edge`" function which works really well for this implementation.

Performance measurement with “perf”:

In this section, we are exploring the Linux “`perf`” utility which is used to measure the performance of the applications or programs that are being run. Firstly, the following command is run in terminal to install the required tools and the correct version of “perf”:

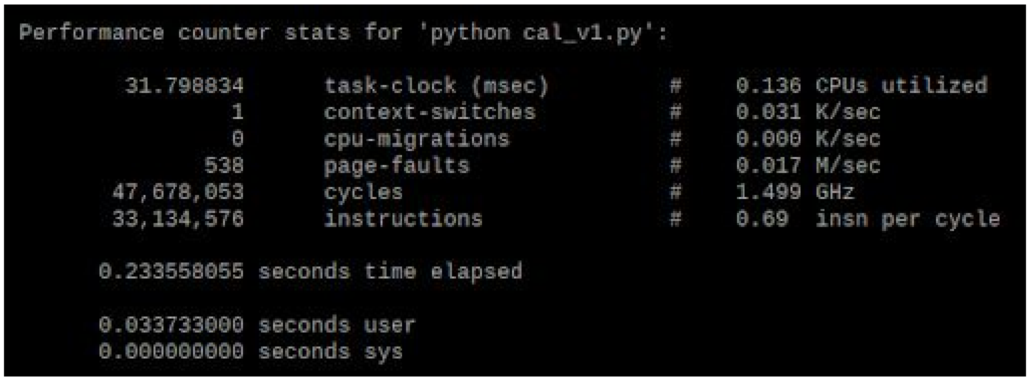
```
sudo apt-get install linux-tools
 sudo apt-get install linux-perf-4.18
```

The perf tool is then explored by using the `"--help"`, `"list"`, and “`version`” flag. The output of these flags would be useful information to know. For example, the `"list` flag will provide you with the list of events that `"list` is able to track. Interestingly, we realized that when we run the “perf” command on its own, we get an error message saying that “linux-perf-5.4 is not installed”. We looked up the /usr/bin/perf to find out what has happened. It turns out that this is because when running “uname -r” it will output version 5.4, and that will be the perf version that will be searched for. Unfortunately, that version of perf is not installed and thus this error happens.

The next step that we did is to test this perf utility. A simple program called “`cal_v1.py`” was created that will import the time library and sleep for 0.2ms. Then the perf utility was ran in terminal as below:

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

Each of the flags above relates to the performance statistics that we want to be displayed as output. The output of the statistics is as below:



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

       31.798834      task-clock (msec)         #    0.136 CPUs utilized
               1      context-switches          #    0.031 K/sec
               0      cpu-migrations            #    0.000 K/sec
             538      page-faults               #    0.017 M/sec
      47,678,053      cycles                    #    1.499 GHz
      33,134,576      instructions              #    0.69  insn per cycle

     0.233558055 seconds time elapsed

     0.033733000 seconds user
     0.000000000 seconds sys
```

Figure 6 - Performance statistics of “cal_v1.py”.

From the performance above, we learned about the different statistics of such a simple program. This can be used as a benchmark or for comparison. What was surprising is how many instructions and cycles this program has considering how simple it is!

Performance measurement of video_control.py:

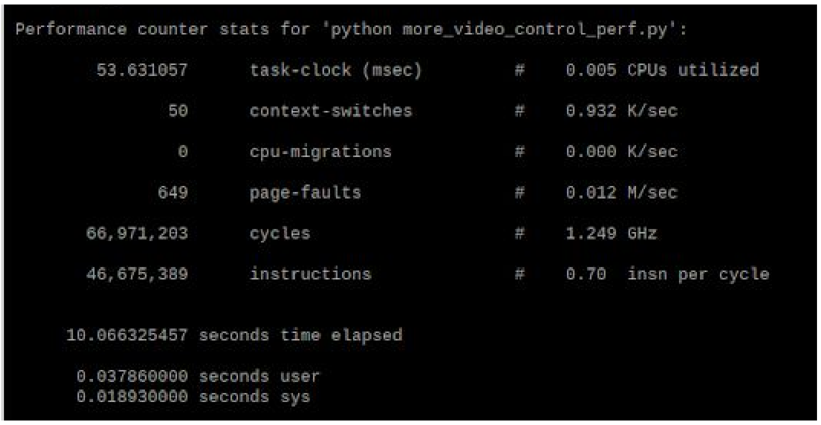
Once we get familiar with the perf utility, this tool will now be used to differentiate the performance between using a polling loop or callback interrupt routines. One thing to note is that this test should only be done using python2 for consistency. Another thing is that both python video_control programs should be modified to only run the program for 10 sec for consistency in performance comparison.

A python program "`more_video_control_perf.py`" is created for this purpose. This program will now have a variable initialized with the current time "`time.time()`" at the beginning of the program. Then, this variable will be used to check if the polling loop has run for 10 seconds and to stop it if it has been more than that. A python program "`more_video_control_cb_perf.py`" is also created based on "`more_video_control_cb.py`". This program is also limited to run for 10 seconds by adding the "`timeout = 10000`" parameter to the `GPIO.wait_for_edge()` function for button 27. This means that the program will only wait for that button press for 10 seconds.

Once the python program is ready, the next step is to test the performance of these programs. First, a "`time.sleep()`" of 200milliseconds is added to the beginning of the polling loop, as we want to see the effect of this sleep. The perf tool is ran for this program as below:

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

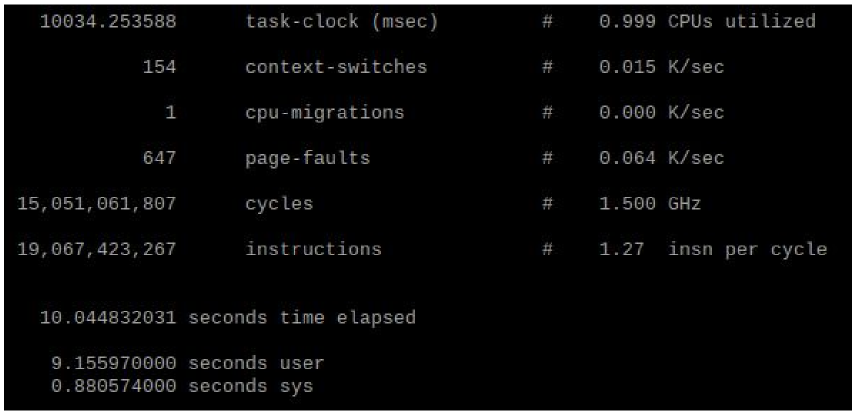
The output statistics from the perf utility is as below:



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

        53.631057      task-clock (msec)         #    0.005 CPUs utilized
               50      context-switches          #    0.932 K/sec
                0      cpu-migrations            #    0.000 K/sec
              649      page-faults               #    0.012 M/sec
       66,971,203      cycles                    #    1.249 GHz
       46,675,389      instructions              #    0.70  insn per cycle

     10.066325457 seconds time elapsed

      0.037860000 seconds user
      0.018930000 seconds sys
```

Figure 7 - Performance of polling loop with 200milliseconds sleep.

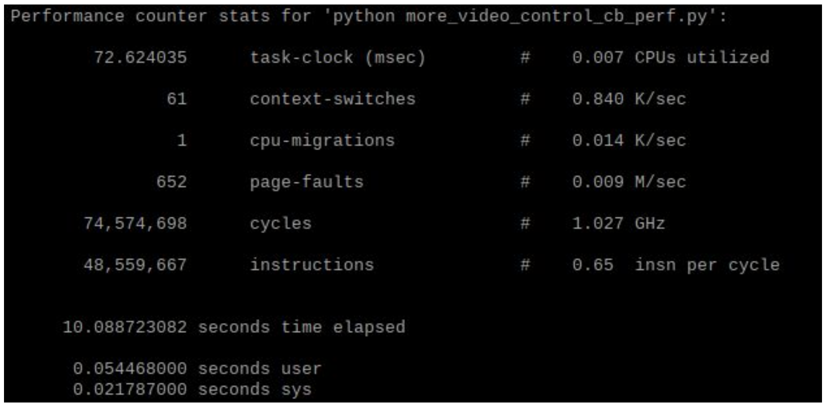
This result may be difficult to explain, as there is nothing yet to compare to. The next step is to retest the performance of this program using several sleep times (20 milliseconds, 2 milliseconds, 200 microseconds, 20 microseconds, and no sleep). The results from these 6 results are analyzed and a trend was found. We learned that as the sleep time decreases, the amount of instructions, along with all of the measured parameters, tends to increase. This just means that since the amount of sleep decreases, the system checks for button presses more oftenly, thus causing this increase in the number of operations. For context, the performance statistics for the polling loop with no sleep is attached below:



```
  10034.253588      task-clock (msec)         #    0.999 CPUs utilized

           154      context-switches          #    0.015 K/sec

             1      cpu-migrations            #    0.000 K/sec

           647      page-faults               #    0.064 K/sec

15,051,061,807      cycles                    #    1.500 GHz

19,067,423,267      instructions              #    1.27  insn per cycle


  10.044832031 seconds time elapsed

   9.155970000 seconds user
   0.880574000 seconds sys
```

Figure 8 - Performance of polling loop with no sleep.

Now that we have learned how the polling loop is pretty inefficient, we will compare the program with callback interrupt routines. Below is the performance statistic of that program:



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

         72.624035      task-clock (msec)         #    0.007 CPUs utilized

                61      context-switches          #    0.840 K/sec

                 1      cpu-migrations            #    0.014 K/sec

               652      page-faults               #    0.009 M/sec

        74,574,698      cycles                    #    1.027 GHz

        48,559,667      instructions              #    0.65  insn per cycle


      10.088723082 seconds time elapsed

       0.054468000 seconds user
       0.021787000 seconds sys
```

Figure 9 - Performance of callback interrupt routine program.

The performance is pretty much similar to when 200milliseconds sleep is used in a polling loop program. The result is pretty surprising as we first expected the callback to even be better than the polling loop with 200milliseconds. But perhaps, this is as efficient as the program can go. However, when compared with the polling loop with no sleep, there is a dramatic difference in efficiency. In the future, we would be using callback interrupt routines instead of polling loops for the reason that is proven in this analysis. Overall, this section is pretty straightforward, despite at first being rather confused by the procedure in the lab. In the end, we are able to perform all of these tests.

PyGame Bounce program:

In this section, we will learn about the implementation of PyGame in Raspberry Pi. We will learn this by creating a bounce program, which is to display a bouncing ball in the PiTFT display. First, we created the "`bounce.py`", which is provided in the PyGame documentation website:

```
https://www.pygame.org/docs/tut/PygameIntro.html
```

We then learned about the different important parts of this sample program. The pygame was first initialized by running:

```
pygame.init()
```

Then, variables are created to set the size of the screen (width and height), speed of the object (object being the ball), and background color of the screen. Then, the workspace, which will act like a drawing canvas is created by the following command, based on the size we determined:

```
pygame.display.set_mode(size)
```

Next, the image of the object (the ball) is loaded to the program using this function:

```
pygame.image.load("magic_ball.png")
```

The png ball image is downloaded off the internet, and we can choose whatever image we want. Some other format beside .png works too! A rectangle is then created for the ball object and stored to a variable:

```
ballrect = ball.get_rect()
```

What the rectangle does is that it will enclose the object. So when we want to animate or move the object, we are doing this by moving the rectangle of the object.

In our infinite while loop, we first have the command to move the location of the object based in the speed we determined:

```
ballrect = ballrect.move(speed)
```

Then, we have if conditional statements that will check if the rectangle location exceeds the workspace limit, and if it does, the value of the speed array of that object will be changed to its negative. What this will do is to have the object go the reverse direction in the next loop. Then, the rectangle containing the ball will be reflected in the workspace and screen. This is done by the following commands:

```
screen.fill(black)
screen.blit(ball, ballrect)
pygame.display.flip()
```

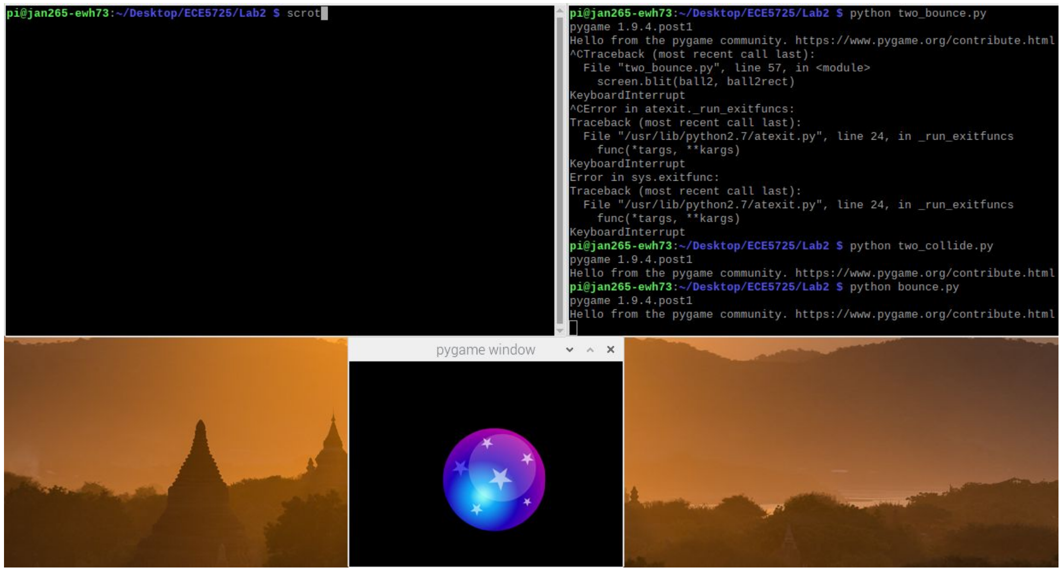
What happens here is that the workspace will be reset to black color. Then, the blit command will print the ball and its rectangle unto the workspace. Finally, the content on the workspace will be reflected to the pygame display, which may be the PiTFT or the display on the monitor/startx. Outside the loop, “`pygame.quit()`” is called and GPIO is being cleaned to reset for the next program.

The above is the explanation for what was provided in the pygame documentation. However, some changes are needed. Firstly, pygame should quit when a button is pressed or after more than a time limit. This is done by setting GPIO 17 as input to the program. Then, the while loop will depend on the variable “`code_run`”, which is first set as “`True`”, and variable “`start_time`” is less than 10 seconds, similar to the method used in performance testing in the previous section. In the polling loop, when button press is detected, “`code_run`” will be set to “`False`” to stop the loop and end the program

The environment variable were also ran in the beginning of the program:

```
import os
os.putenv('SDL_VIDEODRIVER', 'fbcon')
os.putenv('SDL_FBDEV', '/dev/fb1')
```

The purpose is to allow this pygame to be displayed in the PiTFT. With all these changes, this program is ready. It is then tested in PiTFT and in startx. The magic ball will be bouncing all over the screen and will get reflected off the wall. The button 17 will quit the game when pressed, or the game will quit on its own after 10 seconds. We did not find any problems in this section as we are building from a program that was provided.



```
pi@jan265-ewh73:~/Desktop/ECE5725/Lab2 $ scrot
```

```
pi@jan265-ewh73:~/Desktop/ECE5725/Lab2 $ python two_bounce.py
pygame 1.9.4.post1
Hello from the pygame community. https://www.pygame.org/contribute.html
^CTraceback (most recent call last):
  File "two_bounce.py", line 57, in <module>
    screen.blit(ball2, ball2rect)
KeyboardInterrupt
^CError in atexit._run_exitfuncs:
Traceback (most recent call last):
  File "/usr/lib/python2.7/atexit.py", line 24, in _run_exitfuncs
    func(*targs, **kargs)
KeyboardInterrupt
Error in sys.exitfunc:
Traceback (most recent call last):
  File "/usr/lib/python2.7/atexit.py", line 24, in _run_exitfuncs
    func(*targs, **kargs)
KeyboardInterrupt
pi@jan265-ewh73:~/Desktop/ECE5725/Lab2 $ python two_collide.py
pygame 1.9.4.post1
Hello from the pygame community. https://www.pygame.org/contribute.html
pi@jan265-ewh73:~/Desktop/ECE5725/Lab2 $ python bounce.py
pygame 1.9.4.post1
Hello from the pygame community. https://www.pygame.org/contribute.html
```

Figure 10 - bounce.py running in startx.

The next part is to create "`two_bounce.py`", which is built from "`bounce.py`", but will have two balls bouncing. However, the balls will be transparent from each other. This is fairly pretty simple. This time the program should be displayed on monitor so the environment setting is changed to "`/dev/fb0`". Then, another ball png "`soccer_ball.png`" is first loaded and a rectangle for the ball is created. A speed array was also created for the ball, and a different speed was used for this ball. The next changes needed are in the while loop. Everything done for the first ball needs to be done for the second ball too. So this involves moving the rectangle of the ball, checking if the ball hits the wall, and printing the ball on the workspace.

This program is then tested on startx. We can see that there are now two balls on the screen and they are transparent to each other. The button to quit is also working and the program runs for a maximum of 10 seconds. We have verified that this program is running correctly. We encounter no problem in this section as we are just duplicating what we have done for the first ball.

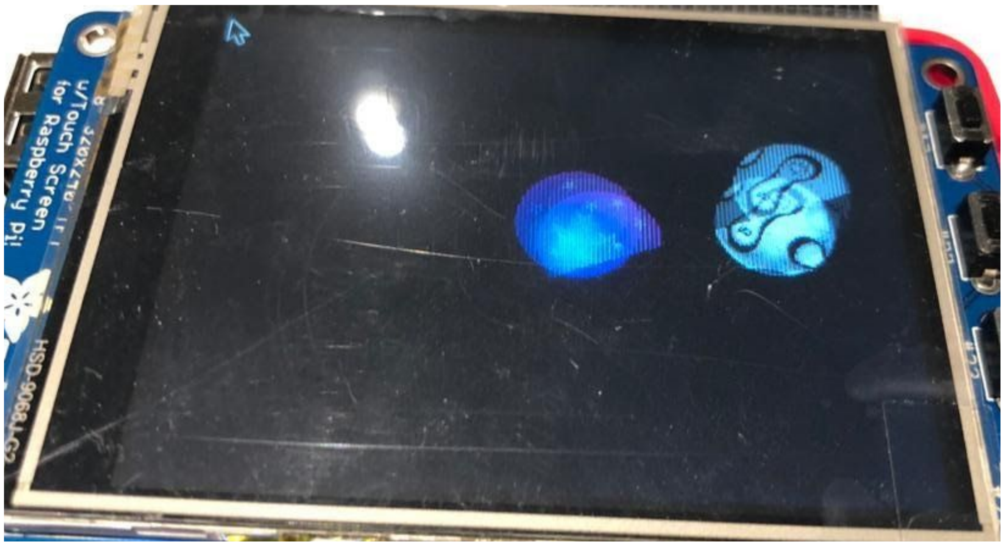


Figure 11 - two_bounce.py running in PiTFT. It may seem like there are duplicate balls, but it is just because of the camera used to take a picture of the display.

The last section for the week is to create a "`two_collide.py`" program. This will be built from the previous program, but this time the two balls will collide off of each other. Overall, this was one of the most challenging parts of the lab. With two balls, not only must you ensure that neither goes off the screen, but they must also bounce off one another. This becomes especially difficult because interactions with the screen edge are not independent from interactions with another ball. After some struggle, we found some ways of handling this. The first is to only move each ball once per iteration of our loop. In other words, we should complete all the calculations necessary for the balls to collide and stay on screen and then move the image of the ball. The second is that we can set a flag when the balls collide. This flag can be decremented each iteration of our loop, but as long as the flag is not 0, our program ignores collision between the balls. This helps ensure that the balls do not become stuck together.

There are several changes that need to be made. First, we realize that the size of the two balls are too big and we need to scale it down. The following function is called on the ball image variable to scale it down:

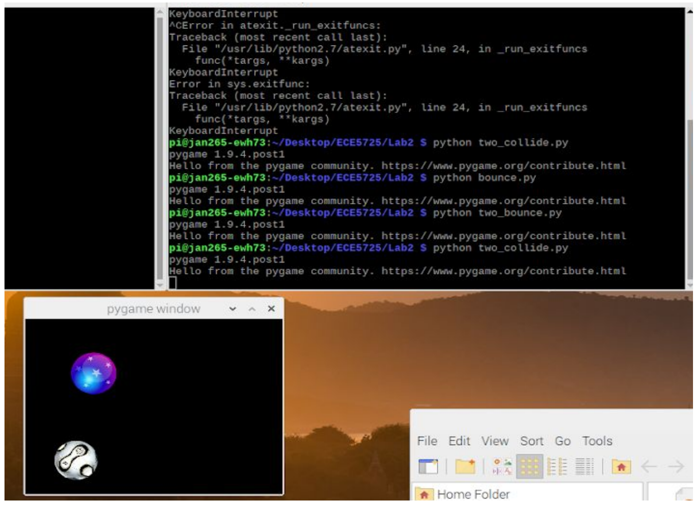
```
pygame.transform.scale()
```

Next, we also realize that the starting position of the balls needs to be adjusted or it will just start from the same position and start colliding. This is done by passing the "`center =`" parameter when setting the rectangle of the balls. Then, we also need to slow down the frame speed, as the ball is moving too fast. The speed array cannot be set to be decimals, so an alternative is found by using "`pygame.time.Clock()`". In the loop, the function "`clock.tick(200)`" is called, which means there can only be 200 frames per second.

After setting up the environment to better support ball collision animation, it is time to design how to create this algorithm. We imagined collisions to happen if the distance between two centers of the balls are equal or less than the total radius of the two balls. So, the minimum distance is calculated by adding the radius of each ball, which is half of the width of the balls' rectangle. This is done by utilizing the "`.left`" and "`.right`" function of the rectangle. In the loop itself, in each iteration the center coordinate of each ball is calculated, which is based on the top, bottom, left, and right location of the rectangle. Then, the distance between two centers of the rectangles are calculated based on this formula:

$$\mathbf{d} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

In the loop, there is then an if statement to check if the midpoint distance of the two balls are less than the total of the two radii, a collision is detected and the next direction of both balls will be reversed. There also needs a "counter" needed, which is to count for the number of iterations since the last collision happened. If it has just recently happened, then we will not detect any collisions first as the balls need to be allowed to move away from each other.



```
KeyboardInterrupt
^CError in atexit._run_exitfuncs:
Traceback (most recent call last):
  File "/usr/lib/python2.7/atexit.py", line 24, in _run_exitfuncs
    func(*targs, **kargs)
KeyboardInterrupt
Error in sys.exitfunc:
Traceback (most recent call last):
  File "/usr/lib/python2.7/atexit.py", line 24, in _run_exitfuncs
    func(*targs, **kargs)
KeyboardInterrupt
pi@jan265-ewh73:~/Desktop/ECE5725/Lab2 $ python two_collide.py
pygame 1.9.4.post1
Hello from the pygame community. https://www.pygame.org/contribute.html
pi@jan265-ewh73:~/Desktop/ECE5725/Lab2 $ python bounce.py
pygame 1.9.4.post1
Hello from the pygame community. https://www.pygame.org/contribute.html
pi@jan265-ewh73:~/Desktop/ECE5725/Lab2 $ python two_bounce.py
pygame 1.9.4.post1
Hello from the pygame community. https://www.pygame.org/contribute.html
pi@jan265-ewh73:~/Desktop/ECE5725/Lab2 $ python two_collide.py
pygame 1.9.4.post1
Hello from the pygame community. https://www.pygame.org/contribute.html
```

Figure 12 - "two_collide.py" running in startx.

This program is tested in both startx and PiTFT and both works as expected. There are a lot of challenges in this section. The first part is in choosing the method for ball collision. We first went ahead to use rectangle's "`colliderect()`" function, but it did not work well as it collides even when the pointy edges of the rectangle touch with each other. This means that visually the balls have not collided with each other. As a result we need to find a different method. Another challenge that we have is in finding functions that would allow us to scale the ball and make the frame rate go slower to lower down the speed of the ball for better visualization. Overall, from this program, we have learned a lot about using PyGame in creating a simple animation and combining it with button inputs from GPIO.

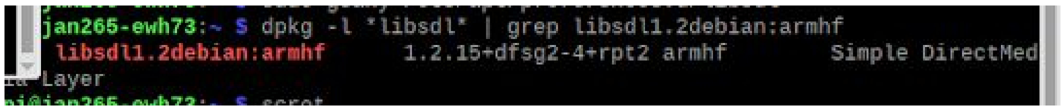
At the end of this section, an SD card backup is done using Apple Pi-Baker. This is very important to allow us to come back to back-up if anything goes wrong in the future.

PiTFT Touch Control:

The next part of the lab is to implement the PiTFT Touch Control to the system. Unfortunately, the touch control does not work too well with the current RPi kernel version, so our RPi needs some functions from the Wheezy kernel version. This is done by enabling Wheezy package sources by writing to a new file "`deb http://legacy.raspbian.org/raspbian wheezy main`" after running "`sudo geany /etc/apt/sources.list.d/wheezy.list`" to open the new file. Then, the Wheezy changes is set as a stable default package source by writing "`APT::Default-release "stable"`" after running "`sudo geany /etc/apt/apt.conf.d/10defaultRelease`" to open the new file. Finally, the libsdl from Wheezy is set as a higher priority in the file “`sudo vim /etc/apt/preferences.d/libsdl`”. This is done by adding the following lines:

```
Package: libsdl1.2debian
Pin: release n=buster
Pin-Priority: -10
Package:libsdl1.2debian
Pin: release n=wheezy
Pin-Priority: 900
```

The state of the sdl library is checked before applying the changes:



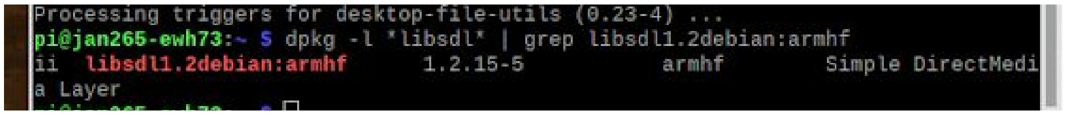
```
jan265-ewh73:~ $ dpkg -l *libsdl* | grep libsdl1.2debian:armhf
 libsdl1.2debian:armhf      1.2.15+dfsg2-4+rpt2 armhf       Simple DirectMed
ia Layer
```

Figure 13 - libsdl result from dpkg before Wheezy changes.

We are searching for the name “`libsdl1.2debian:armhf`”.
Then, the changes that were made are installed by running:

```
sudo apt-get update
sudo apt-get -y --allow-downgrades install libsdl1.2debian/wheezy
```

The state of the sdl library is again checked to see the changes:



```
Processing triggers for desktop-file-utils (0.23-4) ...
pi@jan265-ewh73:~ $ dpkg -l *libsdl* | grep libsdl1.2debian:armhf
ii  libsdl1.2debian:armhf      1.2.15-5         armhf       Simple DirectMedi
a Layer
```

Figure 14 - libsdl result from dpkg after Wheezy changes.

From the above, we can see the differences of before and after the Wheezy changes. Now that the change is confirmed, PiTFT touch should be working. This section is rather intimidating as a slight typo would actually damage a lot of things. We are being extremely careful in this section of the lab.

Since PiTFT touch will now be used, our code would require the following to allow program to be displayed on PiTFT:

```
os.putenv('SDL_VIDEODRIVER', 'fbcon') # Display on piTFT
os.putenv('SDL_FBDEV', '/dev/fb1')
```

The program would also require the following to initialize the mouse driver:

```
os.putenv('SDL_MOUSEDRV', 'TSLIB') # Track mouse clicks on piTFT
os.putenv('SDL_MOUSEDEV', '/dev/input/touchscreen')
```

Lastly, the following function allows us to choose on whether the mouse cursor is to be displayed or not by passing in *True* or *False*:

```
pygame.mouse.set_visible()
```

Quit Button:

The next part of the lab is to create a “quit_button.py” program. This implementation is similar to the way we display the ball animation, however we will be displaying a button. The button is first created as a dictionary, which allows us to add more buttons. Then, a font variable is also created, which allows us to choose the appropriate font size:

```
my_font = pygame.font.Font(None, 50)
my_buttons = {'quit':(240,180)} # Make quit button
```

Since we do not need our screen to be reset/blacked out, the button can just be displayed once to both the workspace and PiTFT display, thus this mechanism can be done outside our main loop. A for loop will be used to render the text based on a chosen font, create a rectangle for it, and to display it on the workspace, for all buttons on the dictionary. It is done by the following code:

```
            for my_text, text_pos in my_buttons.items():
                  text_surface = my_font.render(my_text, True, WHITE)
                  rect = text_surface.get_rect(center=text_pos)
                  screen.blit(text_surface, rect)
            pygame.display.flip()
```

Now that the buttons are displayed, it is time to code how the touch is detected. The touch will be detected as a pygame event. The pygame is able to detect mouse button clicks, and there are both mouse down and mouse up detection. This also translates to touch in PiTFT touchscreen capabilities. We determined that the button selection will be done when our hand/mouse has lifted off it, so it will be on mouse up. The following code is implemented:

```
for event in pygame.event.get():
            if (event.type is MOUSEBUTTONDOWN):
                  pos = pygame.mouse.get_pos()
            elif (event.type is MOUSEBUTTONUP):
                  pos = pygame.mouse.get_pos()
                  x,y = pos
                  if y > 120:
                        if x > 160:
                              print("Quit button is pressed")
                              code_run = False
```

During `MOUSEBUTTONUP` event, the position of the mouse or touch is stored. Using if conditional, we will be checking if the position is in the area of our button, and it is then the program will stop by stopping the main loop of the program, which will then quit pygame. This button is not too accurate yet, as it covers the entire quarter of the bottom right display. This will be further calibrated in the next sections of the labs. In order to prevent RPi to get stuck on the program during testing, a physical quit button and a timeout is added to the program to allow us to quit.

The program is tested on PiTFT and it works really well. The quit button works well and the program will immediately quit upon the quit button press.

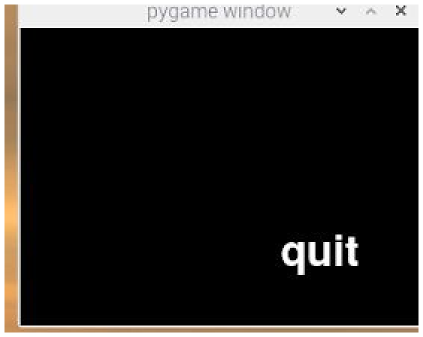


Figure 15 - “quit_button.py” running on startx.

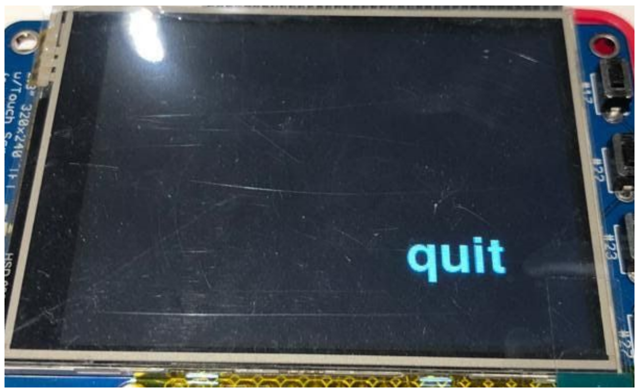


Figure 16 - “quit_button.py” running on PiTFT.

This section taught us about how to create text to be displayed on the screen and how mouse clicks or touchscreen is detected by the program. This prepares us for more complex programs with more buttons and animations displayed. We did not encounter any difficulties in this section.

Screen Coordinates program:

In the next section, a screen coordinate program “`screen_coordinates.py`” will be created where a screen coordinate will be displayed on the screen to show where mouse clicks or touches are made. This is built based on the previous quit button program, but with the additional coordinates being displayed.

Since the screen will now have to be updated on every mouse click or touch, the button will now need to be displayed in the main loop. Additionally, there will be coordinates being displayed on the main loop. However, we are still having the buttons to be initially displayed before the main loop so that the button will be displayed before anything is pressed. In the main loop, the screen will be blacked out and the entire button dictionary will be re-rendered and displayed after every `MOUSEBUTTONUP` is detected. Right after that, a string containing the x, y coordinates is being rendered and displayed using a similar method to buttons. These coordinates are displayed in the middle of the screen. Another part of this section is to store the output of 20 consecutive touch coordinates, and the result is being saved in a txt for future reference. This is done by opening a txt file (or creating one if the file does not exist yet), then writing to it and then closing it at the end, using the open(), write(), and close() functions. Note that the file should be opened as writable.

Below is a code snippet of the `MOUSEBUTTONUP` event, which shows how button press and coordinates are detected:

```
elif (event.type is MOUSEBUTTONUP):
      pos = pygame.mouse.get_pos()
      x,y = pos
      if y > 160 and x > 190:
            print("Quit button is pressed")
            code_run = False

      # Re-display buttons and message every touch
      screen.fill(BLACK)

      # Re-display buttons
      for my_text, text_pos in my_buttons.items():
            text_surface = my_font.render(my_text, True, WHITE)
            rect = text_surface.get_rect(center=text_pos)
            screen.blit(text_surface, rect)

      msg = "touch at " + str(x) + ", " + str(y)
      print(msg)
      text_file.write(msg + "\n")
      text_surface_msg = my_font.render(msg, True, WHITE)
      rect_msg = text_surface.get_rect(center= (100, 80))
      screen.blit(text_surface_msg, rect_msg)
```

The result of the program:

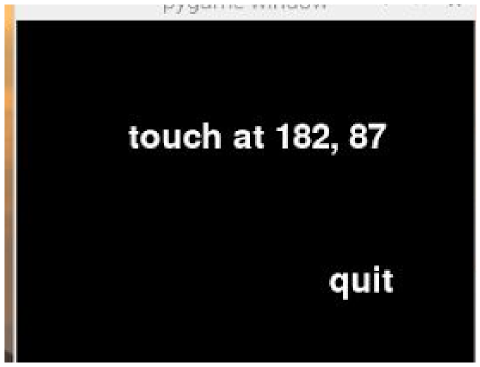


Figure 17 - “screen_coordinates.py” running on startx.

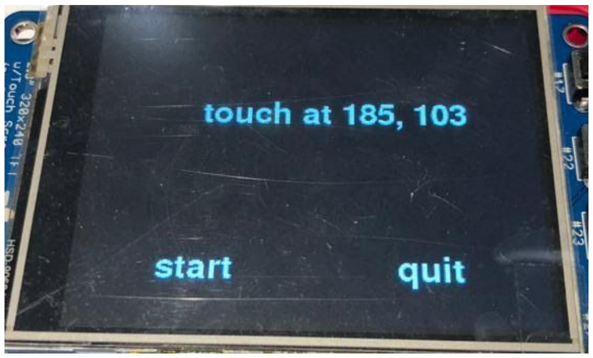


Figure 18 - “screen_coordinates.py” running on PiTFT.

From the result, we can see that the program works well. We can see that the quit button is initially displayed. Then, if any touches or clicks are made, then the coordinate will be displayed. The program will then quit when the quit button is pressed or a timeout is reached. This program is more complicated than the previous ones. We met a few difficulties when deciding on how the buttons and message should be displayed. What happened was that when the coordinate is displayed, the message would stack on top of the previous message, as the screen was not properly reset. Another thing that happened was that the button was not displayed before the screen was touched. In the end, we are able to get the program working.

Two buttons program:

The next part of the lab is to create a “`two_button.py`” program. This program will have two buttons “start” and “quit”. The start button will have the two ball colliding animation from the week1 lab. The program begins by only displaying the two buttons, and it will display the coordinates of the last touch or mouse click. If the start button is pressed, the ball animation will then start. An important part of the program is that the button should not interfere with the animation, so the button will be designed to stay afloat/above the animation.

The first thing that is done is that code snippets from “`two_collide.py`” program are being copied over. This includes the codes where the balls and their rectangles are created, their radius is calculated, and a clock being set to control the frame speed. These things are done before the main loop. The “start” button is also added to the button dictionary that was previously set. An “animate” variable is created to store the information on whether the start button has been pressed, which would then start the ball animation in the main loop.

The challenging part is what’s in the main loop. The loop starts with the event check, which will now check for two regions of touch for the two buttons. The x and y range is calibrated to only cover the small area of the button themselves. When quit is pressed, the main loop will stop. When the start button is pressed, the animate variable will be set to true. In the event check, we also have the coordinate string to store new coordinates when available.

Outside the event check, there is a block of code for the ball animation which will perform ball movement and check for ball collisions to the wall or another ball. However, this will only happen if the animate variable is set to True. Next, we have a section of “`screen.blit()`” which will display the balls, buttons, and text to the screen, along with the rendering of the text for both the buttons and coordinate message. The order is extremely important as the last called “`screen.blit()`” will be displayed topmost. The screen is first reset to black, then “`screen.blit()`” is called in the order for coordinates, balls, then buttons. Keep in mind that the balls are only “blit” when animate variable is True. With the order above, the buttons will be displayed above the ball animation, while the coordinates will be below the ball animation. These are all the code needed for the program and it is ready to be tested.

Below is a code snippet on the order of how things are displayed in the workspace. This order is extremely important to achieve the goal of this program (i.e we want the buttons to not interfere with the ball and touch coordinates need to be displayed during animation):

```
# Clear workspace
screen.fill(BLACK)

# Display coordinates
text_surface_msg = my_font.render(msg, True, WHITE)
rect_msg = text_surface.get_rect(center= (100, 80))
```

```
        screen.blit(text_surface_msg, rect_msg)

        if animate == True:
              # Dislay ball animation
              screen.blit(ball, ballrect)
              screen.blit(ball2, ball2rect)

        # Re-display buttons
        for my_text, text_pos in my_buttons.items():
              text_surface = my_font.render(my_text, True, WHITE)
              rect = text_surface.get_rect(center=text_pos)
              screen.blit(text_surface, rect)

        pygame.display.flip()
```

The testing is done on both PiTFT and startx. When the program first starts, we are welcomed by the display of the start and quit button. When any part of the screen is touched, the coordinate will be displayed. When the start button is pressed, the ball animation will start as expected, with the buttons not interfering with the animation at all. The touch coordinates will still be displayed and updated. The program will quit when the quit button is pressed or the timeout happens. This program is able to run as expected.

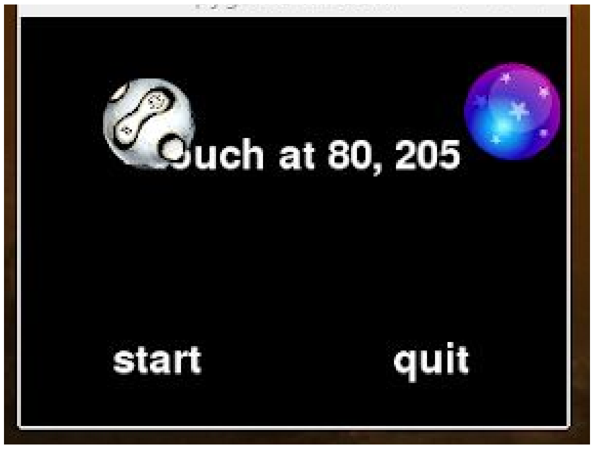


Figure 19 - “two_button.py” running on startx.

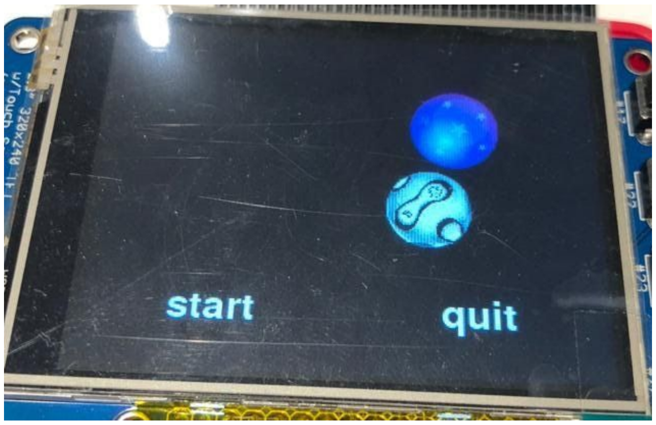


Figure 20 - “two_button.py” running on PiTFT.

This section of the lab is pretty challenging. A lot of thinking needs to be done on how the ball animation can be displayed together with the buttons. A lot of trial and errors are done to make sure that the animation, buttons, and coordinate message can be displayed together. There are instances where the animation is not displayed at all, or when the buttons are not responsive. There is not really a major setback that happened for us, but it just takes a while to experiment with the right algorithm to program. In the end, it is really cool to have an animation running with touchscreen buttons to control it.

Control two collide program:

The final section of this lab is to create the “`control_two_collide.py`” program. This program will have 2 menus. First menu is the “level 1” menu with the “start” and “quit” button on the display and touch coordinates will be displayed if any location except the buttons is touched. Second menu is the “level 2” menu where animation of the ball will be displayed along with four buttons to control the animation. The buttons consist of “pause” to pause and play the animation, “fast” to make the ball animation go faster, “slow” to make the animation go slower, and “back” to go to “level 1” menu. This program is built off the previous programs and several changes will need to be made.

The first focus is to create the “level 1” menu. This is rather simple because it is very similar to the “two_button.py” program. “Quit” button is programmed to quit the main loop and the “start” button is used to set the “animate” variable to “True” and head to the “level 2” menu.

The difference starts at “level 2”. Instead of having the same set of buttons, we are moving to a different set consisting of 4 buttons. A second set of button dictionary is created:

```
my_buttons2 = {'Pause':(50,200), 'Fast':(130,200), 'Slow':(200,200),
'Back':(280,200)} # In animation button
```
So with two different button dictionaries, "`my_buttons`" is displayed when we are in "level 1" and "animate" is "False", while "`my_buttons2`" is displayed when we are in "level 2" and "animate" is "True". The switch of button dictionaries is made using if statements on the section of the main loop where we are rendering and and perform "`screen.blit()`" the button text.

Now that we have the two button dictionaries set up, a change needs to be made on the "`MOUSEBUTTONUP`" event handling. Previously, there was only one button set and the program is set to detect touch coordinates that are within the range of the two buttons. Now that there is a second set, this part of the program is set. During a "`MOUSEBUTTONUP`" event, we would check for the touch position. Then, we would check using if statements for the "animate" variable on whether we are at "level 1" or "level 2". Our previous code would only serve "level 1". When we are at "level 2", we will be checking on button presses using the following code:
```
if y > 180 and x > 240: # back button
        animate = False
  elif y > 180 and x > 160: # slow button
        if frame_speed > 20:
              frame_speed = frame_speed - 20
  elif y > 180 and x > 80: # fast button
        frame_speed = frame_speed + 20
  elif y > 180 and x > 0:
        if pause == False:
              pause = True
        else:
              pause = False
```

The y coordinates range is kept the same as it would detect the lower range of the screen. The x range would then be divided into 4 sections for the 4 different buttons. It will check from the rightmost button then to the leftmost. When the "back" button is pressed, "animate" is set to "False" and the program will go back to "level 1" menu. When the "slow" button is pressed, the "frame_speed" variable value will be decreased by 20. "Frame_speed" is a variable created to have the integer passed for "clock.tick()" function in the main loop, which would set the frame per second of the animation. Notice that "slow" will only change the "frame_speed" if it has not reached the lower bound of the allowed value. Next, the program checks for "fast" button speed which will increase the value of "frame_speed". Lastly, the "pause" button will change the value of "pause" variable to either "True" or "False", reversing the previous state. "Pause" variable is another variable created to store the current state of ball animation, either currently playing or paused.

The next change to be made in the main loop is in the section executed only when “animate” is True. The code used is very similar to the previous program “two_button.py”. The only change is that the ball rectangles will only move when “pause” is set to “False”. This would allow the animation ball to pause and unpause by controlling the “pause” button. The last change comes from the button displayed, where the set of button dictionary that is displayed depends on the current menu level.

Below is a code snippet on the order of how things are displayed in the workspace. This order is extremely important to achieve the goal of this program, such as how different button menu is displayed in different levels:

```
# Clear workspace
      screen.fill(BLACK)

      # Display coordinates
      if animate == False:
            text_surface_msg = my_font.render(msg, True, WHITE)
            rect_msg = text_surface.get_rect(center= (100, 80))
            screen.blit(text_surface_msg, rect_msg)

      if animate == True:
            # Dislay ball animation
            screen.blit(ball, ballrect)
            screen.blit(ball2, ball2rect)

      # Re-display buttons
      if animate == True:
            buttons_dict = my_buttons2
      else:
            buttons_dict = my_buttons
      for my_text, text_pos in buttons_dict.items():
            text_surface = my_font.render(my_text, True, WHITE)
            rect = text_surface.get_rect(center=text_pos)
            screen.blit(text_surface, rect)

      pygame.display.flip()
```

With all of the changes above, our final program is ready to be tested:

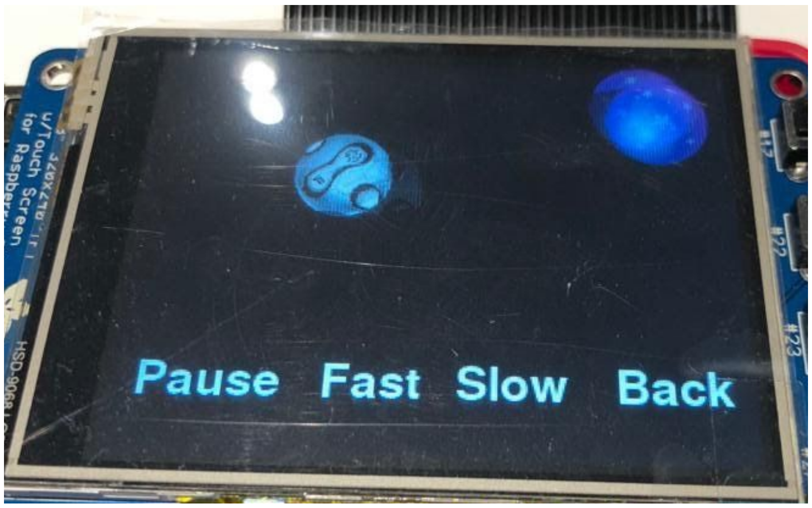


Figure 21 - Level 2 menu of “control_two_collide.py” running on PiTFT.

When the program is tested in PiTFT, we are welcomed by the “level 1” menu, which is a display of “start” and “quit” buttons similar to the previous programs (see Figure 18). The touch coordinates message will also be displayed when part of the screen is touched. When the “start” button is pressed, we will shift to “level 2” menu as shown in Figure 21. The ball animation will not interfere with the buttons and will go under it. Pressing the “fast” and “slow” button will correctly change the ball movement speed. “Pause” button can be used to pause and play the animation. “Back” button will allow us to go back to “level 1” menu and stop the animation. The program will quit when the “quit” button in “level 1” menu or “button 17” is pressed, or when timeout is reached.

The program is able to run smoothly. This program is quite challenging as it has a lot of different components to it. However, since a lot of the things are based on the previous program, the main focus is on how to create two different menus. This involves a lot of trials and errors but we are able to complete the program without too much problems. At the end, we performed a back-up using Apple PiBaker to allow us to restore it if anything unfortunate happens in the future.

Results:

This lab was a bit time consuming and a little difficult due to the fact that we are remote. We hit a few bumps with two_collide.py and the wheezy downgrade, but other than that the lab went smoothly. Only part of our lab group was able to check off during the lab section, but the rest of us were able to check-off during office hours later in the week. Despite this delay, we were able to complete the lab after a bit of work and one of us getting a clean wheezy kernel.

The most difficult parts of the lab were the two_collide python script and the wheezy downgrade. The two_collide code was difficult to implement because there are so many factors that can affect how the balls interact with one another and each other, especially when those actions occur at the same time. Getting help from the TAs and making sure that we complete all ball movement calculations before moving was key. The wheezy downgrade was difficult for one of us because after, the TFT touchscreen would not record the correct coordinates when pressed. Even when our lab partner reloaded the previous saved version of the SD card and re-did the downgrade, he still ended up with the same issue. We therefore believe that this might have been caused by something from lab 1 that we were unaware of. To solve this, Professor Skovira gave us and some other lab groups a clean version of the wheezy kernel to install and this helped resolve the TFT issues.

One piece of advice I would include is that if you have to revert your SD card back to an older version in the middle of a lab, scp whatever code you'd like to save over to the ECE5725 server. This allows us to quickly save code on another system without doing another backup.

Conclusion

Overall, the lab went pretty smoothly. There are challenges that we have faced, such as the Wheezy downgrade causing the TFT touchscreen to not work and when experimenting how to display buttons and animation in PyGame. We are able to gain a lot of learnings from this lab. We learned about adding external buttons to RPi via the GPIO. We learned about how to implement a callback interrupt to detect events like button presses and compared its performance to a polling loop that we have previously been using. We also explored PyGame, which allows us to display animations and buttons, and at the same time integrate the touchscreen capabilities of the PiTFT to create an interactive environment. The lab instructions are very clear and there are no suggestions on what can be improved. All of our code has been uploaded to the ece5725 server:

```
/home/Lab2/M_jan265_ewh73_edk52_2
```