Lab 2

Outline

In Lab2, we will go through the following steps:

- Add external buttons to the RPi setup to expand the control functions of video_control.py from Lab1
- Compare performance of video_test.py in lab 1 and modified versions of this program.
- Develop general methods for checking performance of applications on the Pi with an eye towards performance tuning.
- Develop some test programs using the PyGame and expand these programs into control panels for video_test.py

Lab safety

We have all been in lab before and handled electronic components. Please apply all the cautions that you have learned including the items below:
Integrated electronics are REALLY susceptible to static discharge.

- Avoid walking around while holding bare components
- Confine assembly to the grounded mats on the lab bench
- Make sure you ground yourself by staying in contact with the mat.

Personal safety

- Safety goggles are REQUIRED for soldering

Experimental Safety

- If something on your board is
- Really hot
- Smoking
- Just doesn't look right
- Immediately unplug power
- Train yourself so that this is your first reaction – know where the power switch/cutoff is for your experiment.

Experimental assembly

- Before adding any hardware to the system, power should be OFF
- Before powering up the equipment, please have staff check your added circuit

video_control.py application from Lab1

A few more buttons

In the final step of Lab 1 (step 7), you created the video_control.py to control buttons on the piTFT. This section includes some modifications to this program.
As a first step, make sure the original program, as defined in Lab 1, is working as designed. A short description includes the button functions:
On the piTFT:

- Pause
- Fast-forward 10 seconds
- Rewind 10 seconds
- Quit

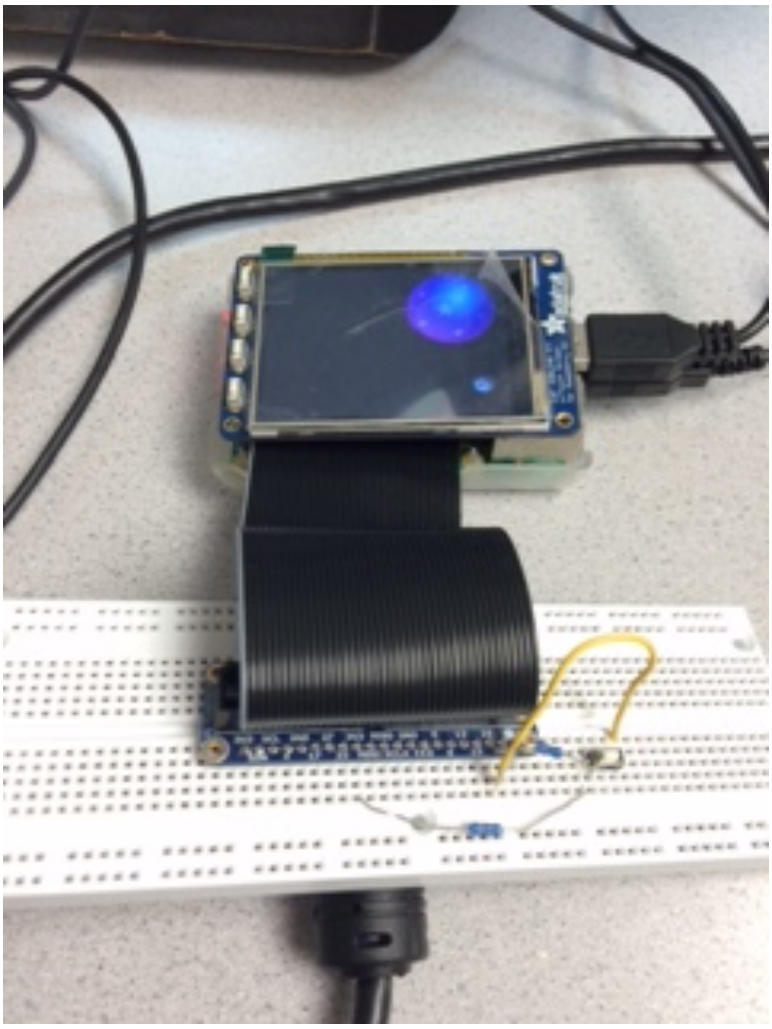
Reference step 6 through step 8 in Lab 1 for additional details.

Using the breakout cable ('Pi-cobbler'), add two additional buttons to the RPi setup used for playing the video on the PiTFT.
Consider which GPIO pins to use, based on what free pins are available

- Once the cable is connected to the protoboard, check that the polarity of the plug is correct. Using a voltmeter, check that the +5 pins and several of the +3.3 pins are in the correct locations (according to the indications on the cable header). If not, or if you are unsure, please check in with the TA.
- Use the safe development techniques discussed in class, so as not to 'cook' the GPIO pins if something goes wrong.
- Once the buttons are wired correctly;

  STOP and please check with the TA to insure correct connections and GPIO selection.

A photo of a setup connected correctly:



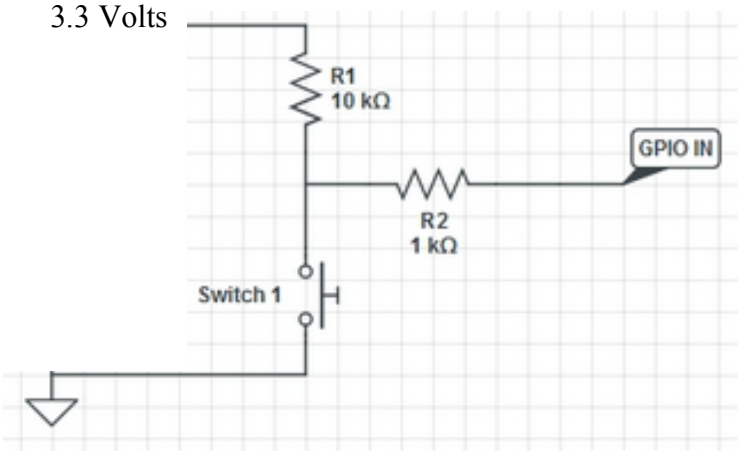
Notes:

- The white stripe on the piCobbler breakout cable is on the same side as the piTFT buttons
- On the protoboard, you can read the pin assignments (starting from the left) 3.3, SDA, SCL, 4, Gnd, etc….
- Note that the numbering on the piCobbler corresponds to BCM numbering (for example pin ‘26’ on the piCobbler is GPIO26
- This example has a single button wired into GPIO26

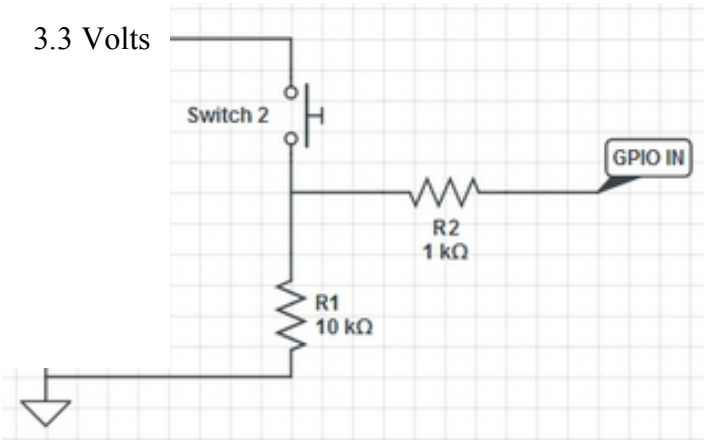
Some references:

GPIO#	2nd func	pin#	pin#	2nd func	GPIO#
N/A	+3V3	1	2	+5V	N/A
GPIO2	SDA1 (I2C)	3	4	+5V	N/A
GPIO3	SCL1 (I2C)	5	6	GND	N/A
GPIO4	GCLK	7	8	TXD0 (UART)	GPIO14
N/A	GND	9	10	RXD0 (UART)	GPIO15
GPIO17	GEN0	11	12	GEN1	GPIO18
GPIO27	GEN2	13	14	GND	N/A
GPIO22	GEN3	15	16	GEN4	GPIO23
N/A	+3V3	17	18	GEN5	GPIO24
GPIO10	MOSI (SPI)	19	20	GND	N/A
GPIO9	MISO (SPI)	21	22	GEN6	GPIO25
GPIO11	SCLK (SPI)	23	24	CE0_N (SPI)	GPIO8
N/A	GND	25	26	CE1_N (SPI)	GPIO7
(Models A and B stop here)					
EEPROM	ID_SD	27	28	ID_SC	EEPROM
GPIO5	N/A	29	30	GND	N/A
GPIO6	N/A	31	32	-	GPIO12
GPIO13	N/A	33	34	GND	N/A
GPIO19	N/A	35	36	N/A	GPIO16
GPIO26	N/A	37	38	Digital IN	GPIO20
N/A	GND	39	40	Digital OUT	GPIO21

3.3 Volts



3.3 Volts



- Once checked, power up and extend four_buttons.py to create ‘six_buttons.py’ to make sure you can correctly read the existing four, and two new, buttons
- Extend ‘video_control.py’ to create ‘more_video_control.py’ to add two new functions to video control; fast forward 30 seconds and rewind 30 seconds
- Modify the ‘start_video’ bash script to use the the new code, more_video_control.py.

Step2: interrupt callbacks

Modification of video_control.py

Once more_video_control.py is running, copy the verified file into a new file, more_video_control_cb.py. In this code, modify the button presses to use threaded callback interrupt routines for button presses. Note:

- Most (all) of the buttons can be modified to use callback routines
- The polling loop used in more_video_control.py will require alteration
- The quit function may be a special case.
- For each button, you will need to:
    - Setup a callback routine
    - Define and event that accesses the callback routine

Once this routine has been redesigned, verify that it operates correctly in a bash script, similar to the operation of start_video from lab1. Name this new bash script start_video_cb. Confirm that the operation of all buttons in start_video_cb is identical to the function in the original start_video script.

Take a backup of your SD card at this point in the lab.

Performance measurement with ‘perf’ utilities

This section explores the Linux ‘perf’ utility to measure performance of applications on the R-Pi. We will be looking into the performance difference between polling loop and interrupt versions of programs implemented in Python during lab.

Step1: Install Perf with the following commands:

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

Try the command:

```
perf –help
```

And it should fail. The message will tell you that your system is missing the correct version of perf. You can remedy this by running:

```
sudo apt-get install linux-perf-4.9
```

Step2:

Test that perf is operating as designed by running:

```
perf --help
perf list
perf --version
```

Also, note the different hardware and software events perf is able to track. Keep in mind that not all of these measurements are available on all platforms.

Step3: Create a python file named cal_v1.py containing the following statements:

```
import time
time.sleep(0.2)  # sleep
```

Step4: Run the following:

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

This will show the following performance statistics:

Task-clock: program execution time in milliseconds
Context-switches: how many times the Linux process scheduler switched control between running processes
cpu-migration: how many times process was moved to a different cpu (or core)
page-faults: How many times a part of the processes virtual memory was copied to physical memory
cycles: how many fetch-decode-execute instruction cycles were run
instructions: How many instructions were actually run for this process.

This test gives a baseline set of statistics for a simple python code. Record the results.

Performance measurement of video_control.py

This section includes a series of tests to begin to characterize performance of the polling and interrupt versions of the video_control routines. Use python2 for all runs for consistency of measurement. Plan to record results from all runs for comparison.

In order to fairly compare the polling and interrupt versions of video_control, consider how you might need to modify the codes to perform for a fixed amount of time; that is, all codes in the experiment should run for a fixed time of 10 seconds (for example), then quit. The idea is to determine how much overhead may be present in the programs associated with button control; not the functions called by the button presses but, rather, the logic used to establish the control for the buttons.

Step1: Modify more_video_control.py to introduce fixed timing prior to the perf run. Run the perf tools, using the measures described above, for more_video_control.py. In the polling loop, make sure to start with a 'sleep' value of 200 milliseconds (time.sleep(0.2)).

Step2: Modify more_video_control_cb.py to introduce a fixed run-time prior to running the perf tool. Once modified, run the perf tools, again with the same measurements, for more_video_control_cb.py

Step3: make successive runs of more_video_control.py, changing polling loop times to 20 milliseconds, 2 milliseconds, 200 microseconds, 20 microseconds, and finally, with no sleep statement at all

Step4: Note changes to the perf measurements and deduce impacts of changes and compare the results between successive runs of more_video_control.py and with more_video_control_cb.py. Include discussion in the Lab report.

PyGame: Bounce Program

1. On your Raspberry Pi, implement the 'Bounce' code in python using pygame (you can find pygame documentation in the Blackboard 'References' section).
2. Extend this code to include 2 balls, two_bounce.py, where each ball moves on the screen at a different speed. Note that the two bouncing balls are designed to be 'transparent' to one another. That is, although a ball bounces off the 'walls' at the edge of the screen, balls may pass over each other when they intersect. Design this code to run on the monitor (or /dev/fb0).
3. Expand two_bouce.py to create two_collide.py, so that the balls alter their trajectories as they collide with one another. There are many techniques for collision of pygame objects. Note that you can check the 'Dynamics' section in: http://people.ece.cornell.edu/land/courses/ece4760/labs/f2016/lab3_particle_beam.html For suggestions on how to alter velocity after collisions. [1]
4. Implement a version of two_collide.py that runs on the piTFT screen. Design this code to run on the piTFT screen and to stop when one of the buttons is hit.

Example of a two_bounce.py screen:

Backup your SD card – Important as next section has some detailed changes.

Demonstrate your work to the TA; note that all demonstrations should be on the RPi and PiTFT. Demonstrate all Codes to the TA including:

- six_buttons.py
- more_video_control.py and modified start_video script
- more_video_control_cb.py and start_video_cb bash script
- demonstrate perf runs with modified and more_video_control.py more_video_control_cb.py
- bounce.py, two_bounce.py and two_collide.py

Week 2:

piTFT touch control

If you haven't done it yet, backup your SD card….

The move from wheezy to Jessie has disrupted the control of the touch screen a bit. In short, some of the SDL calls are broken in Jessie; a fix that seems to work reverts to some wheezy functions for these calls. In order to proceed, please issue the following commands to enable the correct operation of the touch function of the piTFT.

The following instructions were created by the Raspbian user community and ultimately included into Adafruit instructions for using the PiTFT. As a reference, please have a look at the link:

https://learn.adafruit.com/adafruit-pitft-28-inch-resistive-touchscreen-display-raspberry-pi/pitft-pygame-tips

Note on the following instructions: If you are cutting and pasting these instructions using a console window, I had some issues with the 'dash' character, '-'. This character did not seem to copy correctly. If the entries are typed into the console window (rather than using a cut and paste), everything works correctly.

On to the commands:

1) Enable wheezy package sources by editing the file:

```
sudo vim /etc/apt/sources.list.d/wheezy.list
```

and adding the line:

```
deb http://archive.raspbian.org/raspbian wheezy main
```

Save and close the file

Note that the above commands will create a new file.

2) Set stable as default package source (for the wheezy changes) by editing the file:

```
sudo vim /etc/apt/apt.conf.d/10defaultRelease
```

and adding the line:

```
APT::Default-release "stable";
```

Save and close the file

Note that the above commands will create a new file.

3) Set the priority for libsdl from wheezy higher then the Jessie package by editing the file:

```
sudo vim /etc/apt/preferences.d/libsdl
```

and adding the lines:

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

Save and close the file

The above commands also create a new file.

4) Install the changes by running the commands:

```
sudo apt-get update
sudo apt-get –y --force-yes install libsdl1.2debian/wheezy
```

5) Once these changes have been completed, run the 'Automatic Calibration' procedure for the piTFT as described in the AdaFruit link (on Blackboard, see the Lab2 folder, piTFT calibrate link).

Once the above changes have been completed, work may proceed on using touch screen controls. The idea for the following codes will be to use touch as a mouse click to press on-screen buttons controlling your software. It is MUCH easier to begin this process by displaying the small screen on the monitor (rather than the piTFT). Using the monitor, you will be able to observe the impact of code changes and any debug/log information and using a console window. In addition, you can add print statements to a running program to indicate response to events ('pause button hit' for example).

Control for monitor use is determined by the environment variables:

```
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')
```

If you add these into your python program, comment them out to display the piTFT screen output in a small window on the monitor. You will also be using:

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

Which will turn off the mouse cursor. When debugging on the monitor, set this to True so you can easily use the mouse to click the on-screen buttons.

Once you have your code debugged and running on the monitor, you can switch to the actual piTFT. Un-comment the 4 environment variable settings and set the mouse to invisible. If you now launch from an ssh window (on your laptop) the application should run on the piTFT. You may also be able to start the piTFT code from a console window in startx however, there may be a Linux problem that prevents this method from working

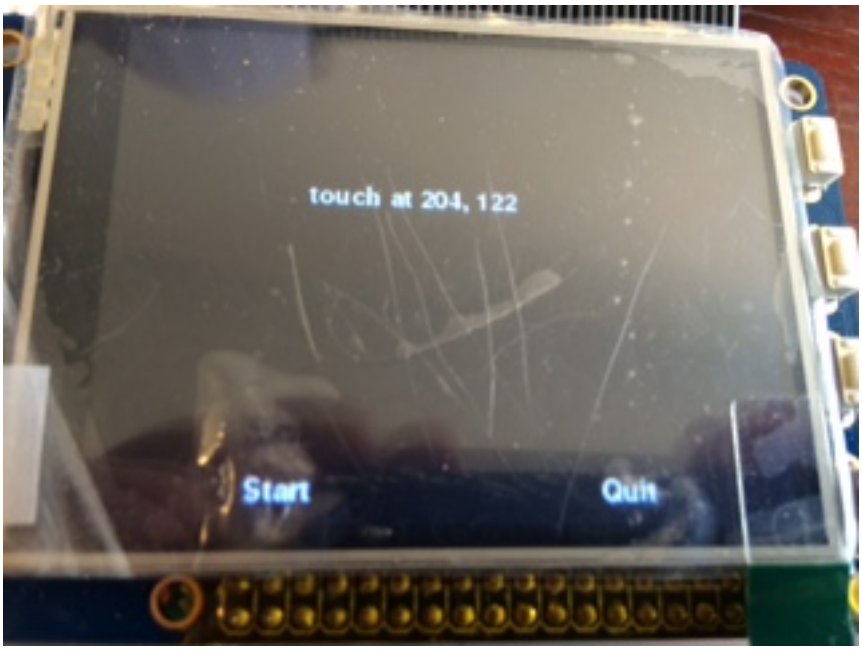
As a final test, the code should be started from the command line on the piTFT to show the embedded operation of the application. [2]

1. Using methods described in the PyGame libraries, implement a single function on the touch screen for two_collide.py. Enable a single button press which quits the program and returns to the console screen.

   Design a python application, quit_button.py, that displays a single 'quit' button on the lower edge of the screen. The program should be designed so that touching the 'quit' button ends the program and returns to the Linux console screen. Initially, the quit function may be implemented by touching at any location on the screen.

   Note that you might want to implement a timer and/or a physical button 'bail out' function in case on-screen button function is initially inoperative. This will prevent excessive power off/power on restarts of a hung RPi system.

2. Expand quit_button.py into a second python application, screen_coordinates.py.



The above example shows a screenshot of some test code to display button press (you do not need to implement the start button for this step….only quit)

The operation of screen_coordinates.py should be:

- Display a single quit button at the bottom of the screen
- Tapping any location on the screen still display 'Hit at x, y' where x, y show the screen coordinates of the hit. Note that the screen should be updated correctly for successive screen hits.
- Tapping the 'quit' button will exit the program. Note that you may need to iterate over several runs of screen_coordinates.py to refine the coordinates of the quit button.
- All hits should be displayed on the Linux console as well
- Plan to include a copy of 20 screen taps, ranging over the extent of the piTFT screen, in your lab report. Think about how best to collect these data on screen taps; what are some simple ways to save all twenty events?

3. Design a python program ‘two_button.py’ with the following functions:

- Two, on-screen buttons are displayed ‘start’ and ‘quit’
- Hitting ‘start’ begins playback of two_collide.py
- Hitting any other location on the screen has no effect
- Hitting ‘quit’ ends the program and returns to the Linux console screen
- The start and quit buttons should be displayed on the screen, and operate whenever they are displayed, during the entire time the program is running (including while the animation is playing)
- Note that button placement should be designed so as not to interfere with video playback.

4. Design a python program control_two_collide.py with the following functions:

Example screenshot showing implementation of second level touch controls

- As in 'two_button.py', start and quit are implemented with identical functions.
- Once the animation begins to play, a second level of button controls should be displayed including the following buttons and associated functions:
  - Pause/restart: pause a running animation. Restart a paused animation
  - Faster: speed up the animation by a fixed amount
  - Slower: slow the animation by a fixed amount
  - Back: stop the animation and return to the 'top' menu screen which implements the start and quit buttons.
- Note that button placement should not interfere with the running animation. Also, coordinates should be detected so as to separate button function (that is, avoid a single tap causing multiple buttons to be hit)

Backup your SD card.

Demonstrate all codes to the TA:

- Quit_button.py
- Screen_coordinates.py
- Two_button.py running with two_collide.py functions
- Control_two_collide.py running with two_collide.py functions

References:

[1] From ECE4760, Lab 3, Professor Bruce Land. Link suggested by Junyin Chen, ECE5725, Fall 2016 student, Cornell MEng 2016.

[2] This section on debugging with the monitor and piTFT paraphrases a note posted by Junyin Chen, ECE5725, Fall 2016 student, Cornell MEng 2016.