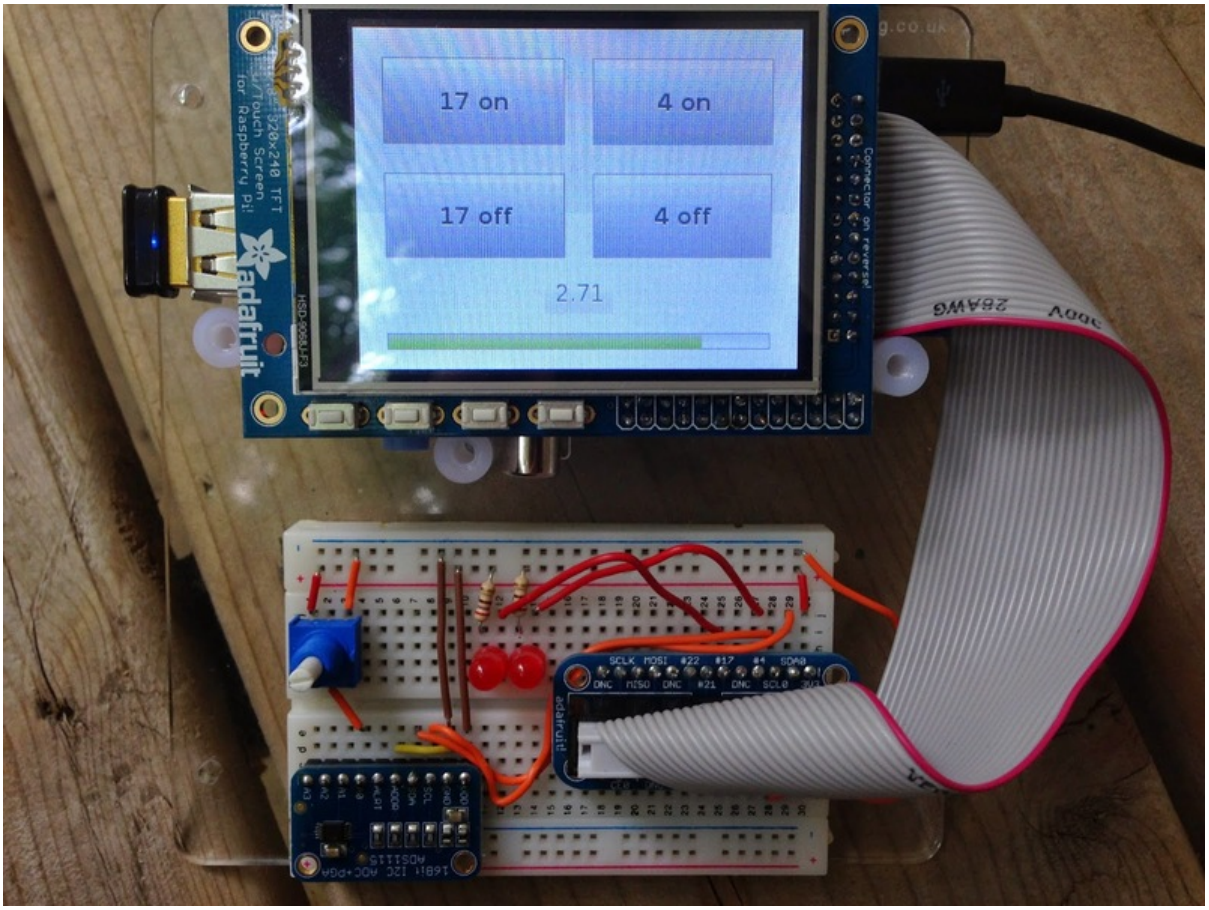
Raspberry Pi Pygame UI basics

Created by Jeremy Blythe



https://learn.adafruit.com/raspberry-pi-pygame-ui-basics

Last updated on 2024-06-03 01:51:42 PM EDT

Table of Contents

Overview

Before diving in to this tutorial, first follow the setup instructions for your PiTFT. Make sure to choose the correct guide particularly for resistive (https://adafru.it/dDK) vs capacitive (https://adafru.it/l6d) displays and carefully follow the section on PiTFT Pygame Tips (https://adafru.it/l6e) - in particular you really need SDL 1.2!

You can also grab all the tutorial source code from git:

```
git clone https://github.com/jerbly/tutorials.git
```

To test the display you can run through these steps in the python console:

```
pi@raspberrypi ~ $ sudo python
Python 2.7.3 (default, Mar 18 2014, 05:13:23)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
&gt;&gt;&gt; import pygame
&gt;&gt;&gt; import os
&gt;&gt;&gt; os.putenv('SDL_FBDEV', '/dev/fb1')
&gt;&gt;&gt; pygame.init()
(6, 0)
&gt;&gt;&gt; lcd = pygame.display.set_mode((320, 240))
&gt;&gt;&gt; lcd.fill((255,0,0))
&lt;rect(0, 0, 320, 240)&gt;
&gt;&gt;&gt; pygame.display.update()
&gt;&gt;&gt; pygame.mouse.set_visible(False)
1
&gt;&gt;&gt; lcd.fill((0,0,0))
&lt;rect(0, 0, 320, 240)&gt;
&gt;&gt;&gt; pygame.display.update()
```

you may also need to try

`os.putenv('SDL_VIDEODRIVER', 'fbcon') # Force PyGame to PiTFT`

in the beginning, if pygame isn't appearing

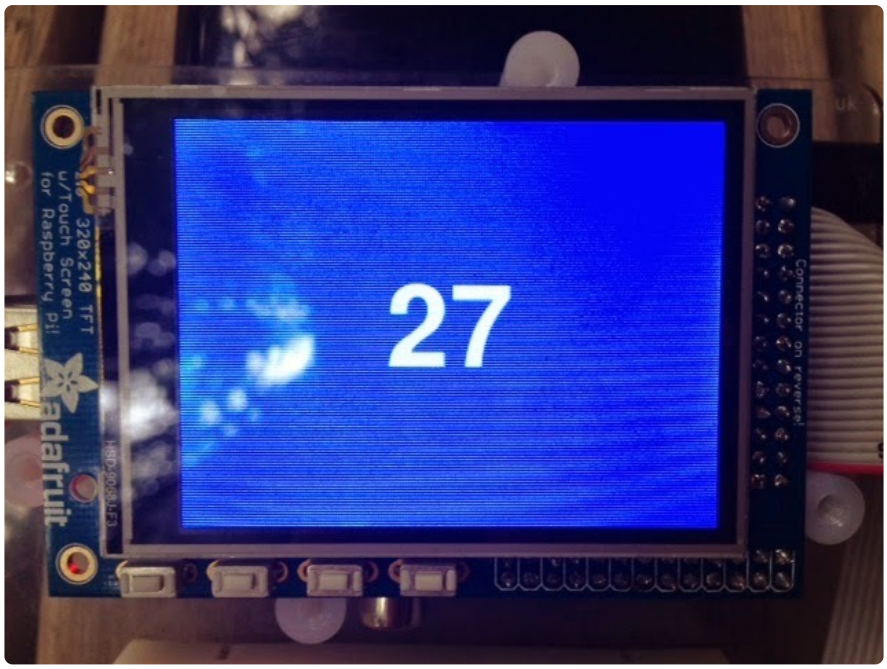
You can also run this test (with a one second sleep) from the pygamelcd project:

```
sudo python test1.py
```

OK now you're ready!

From GPIO to screen

So, we can paint colours on the screen - let's do this from GPIs!



We'll use the four tactile buttons along the bottom of the screen to draw the GPIO number and a coloured background. From left to right the buttons correspond to GPIO #23, #22, #27, and #18.

(**Note**: If you have a revision 1 board then #27 is #21 - you'll just have to change the code a little)

Startup

The code is split into two sections: **startup** and **main loop**. Let's take a look at what's happening during startup.

This first section below imports a few things and then defines a datastructure for the four buttons. It's a simple map from GPIO number to RGB tuple. RGB tuples are used in a lot of pygame calls to set colors for fonts, backgrounds etc.

With the button map in place we can now loop through this and setup all the GPIOs. Each one needs to be set to an input with a pull-up since the buttons are connected to ground.

Finally in this startup section we initialise pygame. The `os.putenv` call here is setting up an environment variable for SDL to tell it which frame buffer device we want to use. We then initialise pygame, hide the mouse pointer, set the display size and fill the background in black.

```
import pygame
import os
```

```
from time import sleep
import RPi.GPIO as GPIO

#Note #21 changed to #27 for rev2 Pi
button_map = {23:(255,0,0), 22:(0,255,0), 27:(0,0,255), 18:(0,0,0)}

#Setup the GPIOs as inputs with Pull Ups since the buttons are connected to GND
GPIO.setmode(GPIO.BCM)
for k in button_map.keys():
    GPIO.setup(k, GPIO.IN, pull_up_down=GPIO.PUD_UP)

#Colours
WHITE = (255,255,255)

os.putenv('SDL_FBDEV', '/dev/fb1')
pygame.init()
pygame.mouse.set_visible(False)
lcd = pygame.display.set_mode((320, 240))
lcd.fill((0,0,0))
pygame.display.update()

font_big = pygame.font.Font(None, 100)
```

Main loop

Here we scan through the GPIOs to see if the buttons are pressed. We simply loop over the map we created earlier pulling out the GPIO number and the RGB tuple into k and v. If the GPIO is set False then the button is down. In which case we fill the background with the color referenced by v and then draw the text of the GPIO number.

Note the `sleep(0.1)` call at the end of the loop. This simply ensures that our program is yielding and not running at 100% CPU usage.

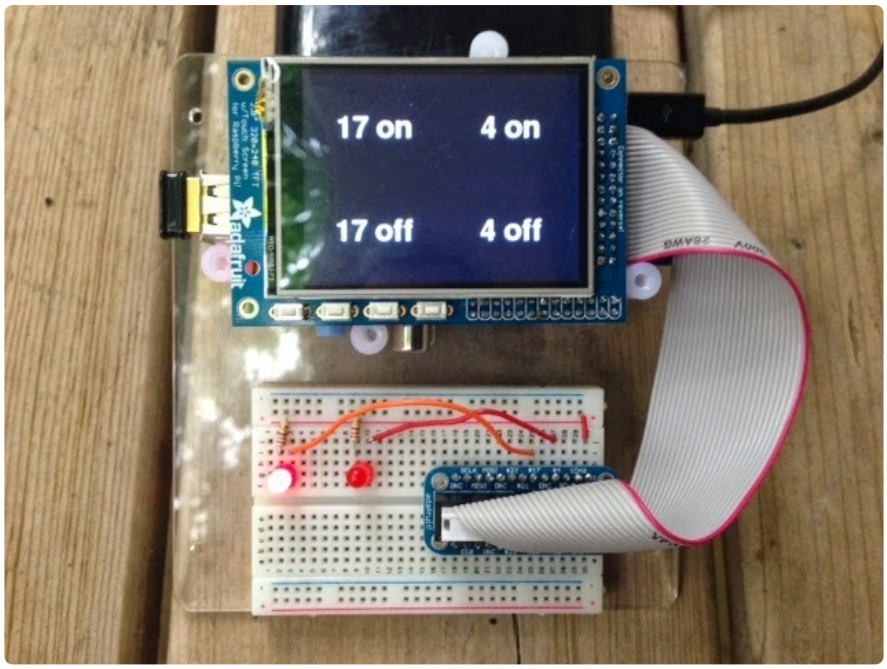
```
while True:
    # Scan the buttons
    for (k,v) in button_map.items():
        if GPIO.input(k) == False:
            lcd.fill(v)
            text_surface = font_big.render('%d'%k, True, WHITE)
            rect = text_surface.get_rect(center=(160,120))
            lcd.blit(text_surface, rect)
            pygame.display.update()
    sleep(0.1)
```

You can also run this from the pygamelcd project:

```
sudo python test2.py
```

From screen to GPIO

Let's see how we get input from the touchscreen. We'll use this to light some LEDs on the breadboard.



With the PiTFT installed with the 4 tactile buttons there aren't many GPIs left on the model B Raspberry Pi. So wire up #17 and #4. The software renders 4 labels on the screen and then looks for mouse events in the four quarters

Startup

In this section we're now setting the two GPIOs as outputs so we can control the LEDs. We also now have a couple more environment variables defined for SDL via `os.putenv` so we can use the touchscreen.

```
import pygame
from pygame.locals import *
import os
from time import sleep
import RPi.GPIO as GPIO

#Setup the GPIOs as outputs - only 4 and 17 are available
GPIO.setmode(GPIO.BCM)
GPIO.setup(4, GPIO.OUT)
GPIO.setup(17, GPIO.OUT)

#Colours
WHITE = (255,255,255)

os.putenv('SDL_FBDEV', '/dev/fb1')
os.putenv('SDL_MOUSEDRV', 'TSLIB')
os.putenv('SDL_MOUSEDEV', '/dev/input/touchscreen')

pygame.init()
pygame.mouse.set_visible(False)
lcd = pygame.display.set_mode((320, 240))
lcd.fill((0,0,0))
pygame.display.update()

font_big = pygame.font.Font(None, 50)
```

UI definition

The `touch_buttons` map simply holds the button text and center position (x,y with top left as 0,0) for this text on the display. We then loop through this map and draw each "button".

```
touch_buttons = {'17 on':(80,60), '4 on':(240,60), '17 off':(80,180), '4 off':
(240,180)}

for k,v in touch_buttons.items():
    text_surface = font_big.render('%s'%k, True, WHITE)
    rect = text_surface.get_rect(center=v)
    lcd.blit(text_surface, rect)

pygame.display.update()
```

Main loop

Here we introduce pygame events (https://adafru.it/eMx). Whenever there's some user interaction an event is generated on a queue for our program to read. It's the responsibility of our program to read this queue quickly enough to avoid events getting thrown away. Of course, we want to do this anyway to have a reasonably responsive program.

We're only really interested in the `MOUSEBUTTONUP` event, this will be added to the queue when you lift your finger off the display. We then simply get the current mouse position (this is also availble in the event itself) and determine which quadrant of the screen was selected.

```
while True:
    # Scan touchscreen events
    for event in pygame.event.get():
        if(event.type is MOUSEBUTTONDOWN):
            pos = pygame.mouse.get_pos()
            print pos
        elif(event.type is MOUSEBUTTONUP):
            pos = pygame.mouse.get_pos()
            print pos
            #Find which quarter of the screen we're in
            x,y = pos
            if y &lt; 120:
                if x &lt; 160:
                    GPIO.output(17, False)
                else:
                    GPIO.output(4, False)
            else:
                if x &lt; 160:
                    GPIO.output(17, True)
                else:
                    GPIO.output(4, True)
    sleep(0.1)
```

This is certainly not a perfect UI experience: you could touch the screen on one button, drag to another and then lift off - which button did you hit? The complexities of event handling for simple things like clicking on buttons is one of the reasons why UI frameworks exist. We'll start using one in the next section.

Again, you can run this from the pygamelcd project:

```
sudo python test3.py
```

PygameUI GPIOs

We're now going to improve the UI by introducing a widget framework PygameUI (https://adafru.it/mER).

Update your version of distribute:

```
sudo easy_install -U distribute
```

Install PygameUI:

```
sudo pip install pygameui
```

PygameUI by default uses, IMO, a rather ugly font. If you want to change this you can simply copy a couple of True Type fonts into the pygameui resources directory.

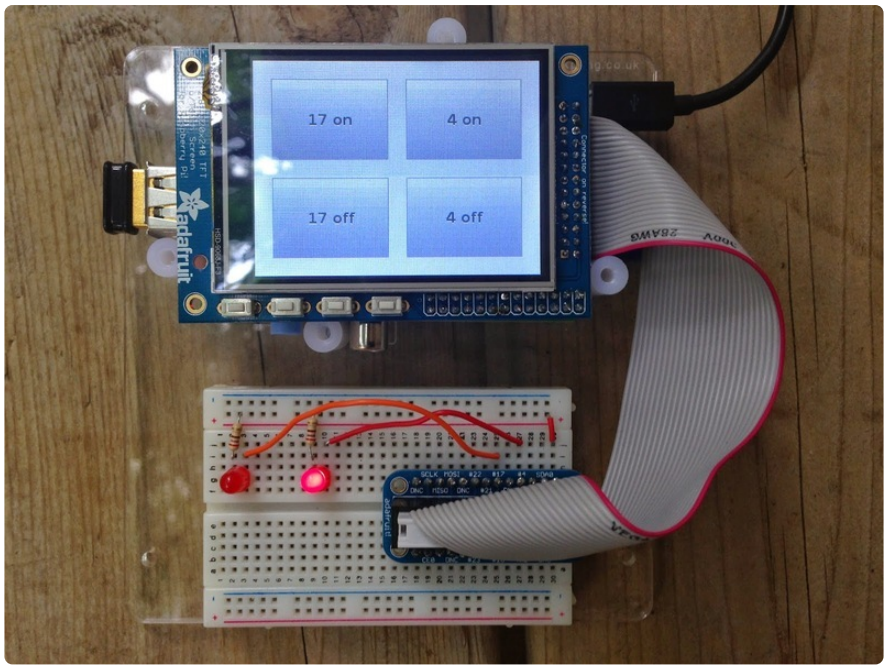
You can find all the ttf files already on your Raspberry Pi with this command:

```
sudo find / -type f  -name '*.ttf'
```

Now just copy a regular and a bold font over the existing ones in the resource directory. You may want to backup the originals just in case.

```
cd /usr/local/lib/python2.7/dist-packages/pygameui/resources/fonts
sudo mv bold.ttf bold.old
sudo mv regular.ttf regular.old
sudo cp /usr/share/fonts/truetype/freefont/FreeSans.ttf regular.ttf
sudo cp /usr/share/fonts/truetype/freefont/FreeSansBold.ttf bold.ttf
```

This example controls GPIO #17 and #4 as before but now we're using the new framework.



The widget rendering and touchscreen events are handled by PygameUI. The PiTft class defines the buttons to draw on screen and the click event to be fired when a button is pressed.

Startup

At the top here we've now imported the pygameui library. We use it quite a bit later on so `ui` is a nice short alias for it.

The other key addition here is logging (https://adafru.it/mES). When using more libraries and frameworks it's very useful to set up logging so you can see the output in a file or, in this case, on the console. We're also able to configure the logger into `DEBUG` mode so we can really see what's going on underneath our code and troubleshoot if needs be.

```
import pygame
import os
import pygameui as ui
import logging
import RPi.GPIO as GPIO
```

```
#Setup the GPIOs as outputs - only 4 and 17 are available
GPIO.setmode(GPIO.BCM)
GPIO.setup(4, GPIO.OUT)
GPIO.setup(17, GPIO.OUT)

log_format = '%(asctime)-6s: %(name)s - %(levelname)s - %(message)s'
console_handler = logging.StreamHandler()
console_handler.setFormatter(logging.Formatter(log_format))
logger = logging.getLogger()
logger.setLevel(logging.DEBUG)
logger.addHandler(console_handler)

os.putenv('SDL_FBDEV', '/dev/fb1')
os.putenv('SDL_MOUSEDRV', 'TSLIB')
os.putenv('SDL_MOUSEDEV', '/dev/input/touchscreen')
```

UI definition

For some hints on how to use widgets within the framework take a look at the kitchen sink (https://adafru.it/mET) script.

The basic principle is that you define a class that inherits from `ui.Scene` and in the `__init__` method you define all the widgets. A Scene is the full window, in our case the entire display (320,240).

Each button's position and size is defined using a `ui.Rect` and a text label is also supplied. An `on_clicked` handler is then assigned - this handler is called when the button is clicked. In our case this is the `gpi_button` method.

The `gpi_button` method simply looks at the label of the button that's been clicked to determine which GPIO to set on or off.

```
MARGIN = 20

class PiTft(ui.Scene):
    def __init__(self):
        ui.Scene.__init__(self)

        self.on17_button = ui.Button(ui.Rect(MARGIN, MARGIN, 130, 90), '17 on')
        self.on17_button.on_clicked.connect(self.gpi_button)
        self.add_child(self.on17_button)

        self.on4_button = ui.Button(ui.Rect(170, MARGIN, 130, 90), '4 on')
        self.on4_button.on_clicked.connect(self.gpi_button)
        self.add_child(self.on4_button)

        self.off17_button = ui.Button(ui.Rect(MARGIN, 130, 130, 90), '17 off')
        self.off17_button.on_clicked.connect(self.gpi_button)
        self.add_child(self.off17_button)

        self.off4_button = ui.Button(ui.Rect(170, 130, 130, 90), '4 off')
        self.off4_button.on_clicked.connect(self.gpi_button)
        self.add_child(self.off4_button)

    def gpi_button(self, btn, mbtn):
        logger.info(btn.text)
```

```
        if btn.text == '17 on':
            GPIO.output(17, False)
        elif btn.text == '4 on':
            GPIO.output(4, False)
        elif btn.text == '17 off':
            GPIO.output(17, True)
        elif btn.text == '4 off':
            GPIO.output(4, True)

ui.init('Raspberry Pi UI', (320, 240))
pygame.mouse.set_visible(False)
ui.scene.push(PiTft())
```

Main loop

This is a key point. The main loop of you program is now taken care of by pygameui. The control has been inverted (https://adafru.it/mEU) - we've supplied pygameui with code to call when certain user actions occur.

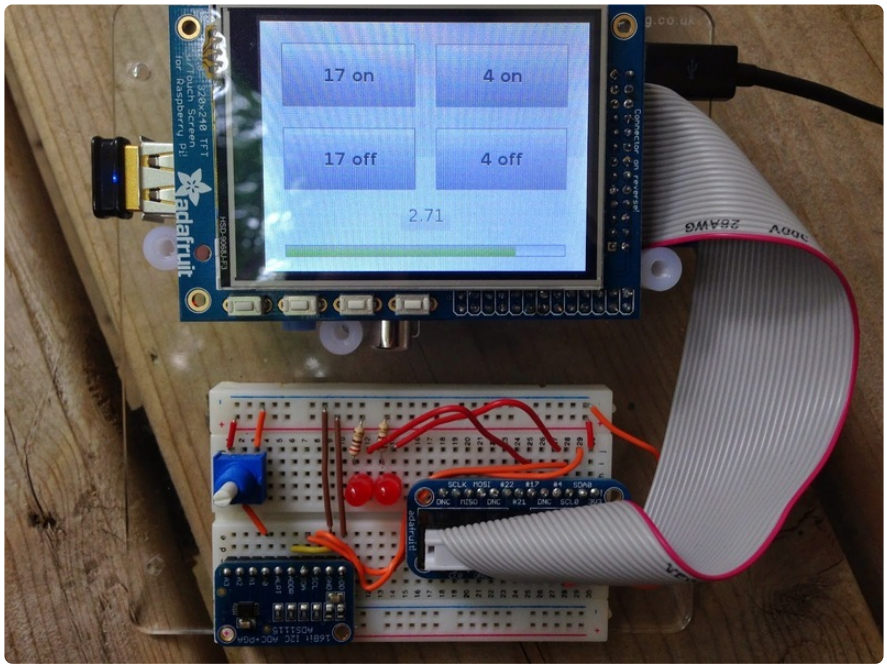
The pygameui main loop is started with this final line of code:

```
ui.run()
```

As usual, you can run this from the pygamelcd project:

```
sudo python test4.py
```

Analog input



This example uses a 10K potentiometer to provide a varying voltage. For analog to digital I normally use an MCP3008 over SPI (https://adafru.it/mEV). That's not possible here though because the PiTFT uses both SPI channels on the Pi. So I've switched to an I2C ADC: ADS1115 16-Bit ADC - 4 Channel with Programmable Gain Amplifier (http://adafru.it/1085).

Get the Adafruit Python library:

```
cd /home/pi
git clone https://github.com/adafruit/Adafruit-Raspberry-Pi-Python-Code.git
```

If you need to enable i2c follow this guide: Configuring I2C (https://adafru.it/dEO)

Startup

At the top of this script now we have some more imports for the threading and ADC.

The gain and sample rate are configured and the ADC initialized.

```
import sys
sys.path.append('/home/pi/Adafruit-Raspberry-Pi-Python-Code/Adafruit_ADS1x15')

import pygame
import os
import pygameui as ui
import logging
import RPi.GPIO as GPIO
import signal
from Adafruit_ADS1x15 import ADS1x15
import threading
import time
```

```
ADS1015 = 0x00  # 12-bit ADC
ADS1115 = 0x01  # 16-bit ADC

# Select the gain
# gain = 6144  # +/- 6.144V
gain = 4096  # +/- 4.096V
# gain = 2048  # +/- 2.048V
# gain = 1024  # +/- 1.024V
# gain = 512   # +/- 0.512V
# gain = 256   # +/- 0.256V

# Select the sample rate
sps = 8    # 8 samples per second
# sps = 16   # 16 samples per second
# sps = 32   # 32 samples per second
# sps = 64   # 64 samples per second
# sps = 128  # 128 samples per second
# sps = 250  # 250 samples per second
# sps = 475  # 475 samples per second
# sps = 860  # 860 samples per second

# Initialise the ADC using the default mode (use default I2C address)
# Set this to ADS1015 or ADS1115 depending on the ADC you are using!
adc = ADS1x15(ic=ADS1115)

#Setup the GPIOs as outputs - only 4 and 17 are available
GPIO.setmode(GPIO.BCM)
GPIO.setup(4, GPIO.OUT)
GPIO.setup(17, GPIO.OUT)

log_format = '%(asctime)-6s: %(name)s - %(levelname)s - %(message)s'
console_handler = logging.StreamHandler()
console_handler.setFormatter(logging.Formatter(log_format))
logger = logging.getLogger()
logger.setLevel(logging.DEBUG)
logger.addHandler(console_handler)

os.putenv('SDL_FBDEV', '/dev/fb1')
os.putenv('SDL_MOUSEDRV', 'TSLIB')
os.putenv('SDL_MOUSEDEV', '/dev/input/touchscreen')

MARGIN = 20
```

Potentiometer thread

A thread is used to constantly read the potentiometer. If you take the reading in-line in the scene update method then you'll slow down the screen refresh rate.
This separate thread allows the screen to be redrawn in parallel with us reading a value from the ADC.

The class is defined as a callable - hence the `__call__` method. It takes a reference to the pitft class so we can update values on it. There's a simple termination scheme used by checking a flag on every loop. Later on you'll see how we cleanly close the program by setting this flag.

So we simply read the voltage value from the ADC and set the label and progress bar position on the pitft.

```
class PotReader():
    def __init__(self, pitft):
        self.pitft = pitft
        self.terminated = False

    def terminate(self):
        self.terminated = True

    def __call__(self):
        while not self.terminated:
            # Read channel 0 in single-ended mode using the settings above
            volts = adc.readADCSingleEnded(0, gain, sps) / 1000
            self.pitft.set_volts_label(volts)
            self.pitft.set_progress(volts / 3.3)
```

UI definition

We've added a couple more widgets to the scene now and resized the buttons to accommodate them. We're showing the voltage on a label and a progress widget. The `set_progress` and `set_volts_label` methods are called from the PotReader thread to update the values in these widgets.

```
class PiTft(ui.Scene):
    def __init__(self):
        ui.Scene.__init__(self)

        self.on17_button = ui.Button(ui.Rect(MARGIN, MARGIN, 130, 60), '17 on')
        self.on17_button.on_clicked.connect(self.gpi_button)
        self.add_child(self.on17_button)

        self.on4_button = ui.Button(ui.Rect(170, MARGIN, 130, 60), '4 on')
        self.on4_button.on_clicked.connect(self.gpi_button)
        self.add_child(self.on4_button)

        self.off17_button = ui.Button(ui.Rect(MARGIN, 100, 130, 60), '17 off')
        self.off17_button.on_clicked.connect(self.gpi_button)
        self.add_child(self.off17_button)

        self.off4_button = ui.Button(ui.Rect(170, 100, 130, 60), '4 off')
        self.off4_button.on_clicked.connect(self.gpi_button)
        self.add_child(self.off4_button)

        self.progress_view = ui.ProgressView(ui.Rect(MARGIN, 200, 280, 40))
        self.add_child(self.progress_view)

        self.volts_value = ui.Label(ui.Rect(135, 170, 50, 30), '')
        self.add_child(self.volts_value)

    def gpi_button(self, btn, mbtn):
        logger.info(btn.text)

        if btn.text == '17 on':
            GPIO.output(17, False)
        elif btn.text == '4 on':
            GPIO.output(4, False)
        elif btn.text == '17 off':
            GPIO.output(17, True)
        elif btn.text == '4 off':
            GPIO.output(4, True)

    def set_progress(self, percent):
        self.progress_view.progress = percent
```

```
    def set_volts_label(self, volts):
        self.volts_value.text = '%.2f' % volts

    def update(self, dt):
        ui.Scene.update(self, dt)
```

And finally...

To start everything going we initialise pygameui, construct the pitft class and then start the potreader thread with a reference to pitft.

Since we're running this extra thread we need a clean way to stop the program. A signal handler is used to trap ctrl+c and terminate the PotReader thread before calling sys.exit - otherwise the program will not close.

```
ui.init('Raspberry Pi UI', (320, 240))
pygame.mouse.set_visible(False)

pitft = PiTft()

# Start the thread running the callable
potreader = PotReader(pitft)
threading.Thread(target=potreader).start()

def signal_handler(signal, frame):
    print 'You pressed Ctrl+C!'
    potreader.terminate()
    sys.exit(0)

signal.signal(signal.SIGINT, signal_handler)

ui.scene.push(pitft)
ui.run()
```

You can run this final example from the pygamelcd project:

```
sudo python test5.py
```