CircuitPython Display Support Using displayio

Created by Carter Nelson

Last updated on 2019-12-14 10:15:32 PM UTC
With the introduction of CircuitPython 4.0 (https://adafruit.it/EFq) comes a new native library - displayio (https://adafruit.it/EFr). This library provides the support needed for drawing to graphical displays. It allows for some common tasks like displaying bitmap images, drawing text with fonts, etc. However, there are also some fancy additional features that provide the framework for creating extended functionality.

This guide will go over the main aspects of the new displayio library and describe how they are used. It will explain how all the bits work together to finally create colored pixels on your display. Additionally, a set of examples to demonstrate some typical use cases are provided.

Let's get started...
Library Overview

The official documentation for the `displayio` library can be found here:

https://adafruit.it/EFr

You'll want to go there for detailed information about using the `displayio` library. This guide is meant to be a compliment to that information.

We start with an overview of what all the parts do.

Image Related Things

Graphics means images, right? Pretty much. These are the items that relate to essentially that.

- **Bitmap** (https://adafruit.it/EFs) - This is pretty much what you expect, a 2D array of pixels. Each pixel contains an index into a "pixel shader", typically a **Palette**, which is where the actual color information comes from.
- **OnDiskBitmap** (https://adafruit.it/EFt) - This creates a Bitmap image (picture) from a file stored on a disk, like `omg_cute_kitteh.bmp`. It must also be used in conjunction with a pixel shader, typically **ColorConverter**, to provide the color information.
- **Palette** (https://adafruit.it/EFu) - This is a simple list of colors. A Bitmap's pixel value is an index into this list.
- **ColorConverter** (https://adafruit.it/EFv) - Use to convert between color formats.

Collection Related Things

Bitmaps are not displayed directly. Instead, they are added to a set of nested collection like classes which ultimately get shown on the display.

- **TileGrid** (https://adafruit.it/EFw) - This uses a Bitmap and a pixel shader (Palette) to draw actual pixels. It must be added to a **Group**.
• **Group** - This is a collection of one or more TileGrids. It can also contain other Groups. The contents of a Group are ultimately what get shown on a display.

**Display Hardware Things**

This sets up the actual display hardware and how it is connected to the microcontroller.

• **Display** - This is the actual display. It must be connected to the host controller via a "display bus".
• **FourWire** - A SPI based display bus.
• **ParallelBus** - An 8-bit parallel display bus.

**Coordinate System**

Two dimensional (2D) information is used throughout the displayio library. The 2D objects have an associated width and height, usually in units of pixels. Locating things, like pixels, within these 2D areas is done using x and y coordinates. Here's an example with width=4 and height=3:

![Coordinate System Diagram]

Note the following:

• the origin is in the upper left hand corner
• y is positive in the down direction
• the first pixel is at (0, 0) and the last pixel is at (3, 2), which corresponds to (width - 1, height - 1).
Bitmap and Palette

The **Bitmap** and **Palette** classes work together to generate colored pixels. So let's discuss them together.

**Bitmap**

This one is nice and easy. It's a 2D array of pixels. Each bitmap is `width` pixels wide and `height` pixels tall. Each pixel contains a `value` and you specify the maximum number of possible values with `value_count`. You can think of this as the total number of colors if you want.

Here is how you would create a bitmap 320 pixels wide, 240 pixels high, with each pixel having 3 possible values:

```python
bitmap = displayio.Bitmap(320, 240, 3)
```

Here is how you would set the pixel at `(x, y) = (23, 42)` to a value of 2:

```python
bitmap[23, 42] = 2
```

Note that the maximum `x` value is `width - 1`, the maximum `y` value is `height - 1` and the maximum pixel `value` is `value_count - 1`. This is due to the zero based indexing. Similarly, the first pixel and color value are all at 0.

**Palette**

This is also pretty straightforward. It is a simple list of color values. You specify the total number of colors with `color_count`.

Here is how you would create a palette with 3 total colors:

```python
palette = displayio.Palette(3)
```

Here is how you would specify the color for each entry:

```python
palette[0] = 0xFF0000 # red
palette[1] = 0x00FF00 # green
palette[2] = 0x0000FF # blue
```

Note how the last entry is at `color_count - 1`.

**Bitmap + Palette**

Think of the **Bitmap** and **Palette** working together like this:
Bitmap

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Palette

\[ \begin{array}{c}
\text{Bitmap} \quad + \quad \text{Palette} \\
\end{array} \]

\[ \begin{array}{c}
\text{Bitmap} \quad + \\
\end{array} \quad \begin{array}{c}
\text{Palette} \\
\end{array} \]

\[ \text{Result} \]

\[ \begin{array}{c}
\text{Result} \\
\end{array} \]

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
TileGrid

If all we wanted to do was display an entire bitmap image onto our display, we could probably stop here. We could just do something like display.show(bitmap) and our bitmap would show up. However, the CircuitPython displayio library adds a few extra layers to the mix. This is done for good reason (spoiler alert = games), but it may initially seem overly complex and confusing. Hopefully we can help clear that up here.

Let's start with the first item, the TileGrid class.

TileGrid

The TileGrid class slices up a source bitmap into multiple rectangular regions called tiles. You can have one or more tiles arranged in a 2D array called a grid. Thus the name TileGrid.

You specify the source bitmap with bitmap and you also need an associated pixel_shader to generate the pixel colors.

Then, you specify how many tiles the TileGrid will have using width and height. These are the number of tiles, not the number of pixels. The size of each tile will be the same and is specified by tile_width and tile_height. These are in units of pixels. Furthermore, the number must evenly divide into the source bitmap's dimensions. So you can't just specify anything. You can specify the initial contents of the tiles using default_tile. This is an index into the source bitmap's tiles and it can be changed later for each individual tile.

Finally, as we will see later, a TileGrid will be added to a Group. You specify the 2D location of the TileGrid relative to the Group with x and y.

Changing a Tile

In the example above, each tile has the default index of 0. This index refers to the tile index in the Source Bitmap. You can reassign any tile in the TileGrid to any of the available indices from the Source Bitmap. To do so, use the syntax:

\[ \text{tile_grid[tile_grid_index]} = \text{source_index} \]

The tile_grid_index can either be the integer number for the tile index or an (x, y) tuple - both notations are shown in the TileGrid example above. The source_index is the integer number from the Source Bitmap - also shown in the example above.
Using our example from above, if we did something like this:

```python
tile_grid[0] = 9
tile_grid[1] = 5
tile_grid[2] = 3
tile_grid[3] = 0
```

or this, which uses the other notation:

```python
tile_grid[0, 0] = 9
tile_grid[1, 0] = 5
tile_grid[0, 1] = 3
tile_grid[1, 1] = 0
```

We would end up with something like this:

Note that nothing is changing in the Source Bitmap. Only the **TileGrid** is changed. You can do this over and over as many times as you want.

Also note that only the **TileGrid** will eventually be shown on the display. The Source **Bitmap** just lives in memory and serves up the graphical data used by the **TileGrid**.
OK, can we please draw something on the display now? Not just yet. Sorry. We're close. Very close. Just one more item to talk about - the Group.

Bitmap and Palette work together to actually make colored pixels. They both get sent to a TileGrid, which allows for some fancy slicing and dicing of the bitmap (if you want to). You can have more than one TileGrid. To collect them all together for final display, you put them into a Group. You can even add a Group to a Group. This allows for some really fancy nesting and drawing.

So let's talk about the Group class. It's actually not that complex.

Group

The Group class is pretty simple. It's just a collection of TileGrids that you have created. It also allows for nesting other Groups (subgroups) within a Group.

The maximum number of items the Group can contain is specified with max_size. Once created, you add items to the Group using append() or insert(). You can change an item using index notation group[index] = tilegrid. You can remove an item using pop(), or the built in del command.

The Group will appear on the screen rooted at the location you specify with x and y. All the items in the Group are positioned relative to this root location (remember TileGrid has x and y also). You can also scale the entire contents of the Group using scale. This is a simple integer scaling factor. 1 is normal, 2 is twice as big, etc.

The Group is what we will finally show on our Display. The end result looks something like this:

![Diagram showing Group and TileGrids](https://learn.adafruit.com/circuitpython-display-support-using-displayio)

This shows a notional Group located at (x, y) on the Display. It contains 3 TileGrids of differing size, shape, and location. Note that the TileGrid locations, specified by their own (x, y) values, are relative to the Group. The values can be negative, like the x value for [1]. Also note how the TileGrid stored in Group index [2] overlaps and is shown above the TileGrid stored in index [0]. This is how "z ordering" works within a Group.
OK, let's setup an actual display so we can start showing stuff. There are two parts to this - the display itself, called Display, and how it is connected to the host controller via some "display bus" like FourWire or ParallelBus.

You first setup the display bus specific to your setup, be it FourWire, ParallelBus, etc. Then, when you setup your Display, you will pass this in so it can be used.

FourWire

The FourWire class is used to talk to displays over a spi_bus using the typical four pins associated with SPI - SCK, MOSI, MISO, and CS (aka, chip_select). One additional pin needed for the display is a pin to indicate if the information being sent over the bus is "data" (image information) or "command" (display control). This is done with the D/C pin specified via the command parameter.

To setup a FourWire bus, you would first create a spi_bus object in the normal way. You would then pass that in, along with specifications for the command and chip_select pins to use.

Here's the basic usage example for hardware SPI:

```python
display_bus = displayio.FourWire(board.SPI(),
       command=board.D10,
       chip_select=board.D9)
```

ParallelBus

A parallel bus is fast, but it takes a lot of pins. You'll need 8 pins for the main data, and they need to be in consecutive order on one of the microcontroller's ports and the first pin has to be on port number 0, 7, 15, or 23 (so we can write the byte in a single DMA command). Then you specify the first pin for data0 and the rest (the other 7) are inferred.

Then you need 4 more digital pins that can be used for command, chip_select, write, and read. Oof. That's 12 pins.

The biggest road block will be finding a microcontroller with all those pins AND with 8 consecutive pins on the same port. What does "port" mean? It refers to something lower level that you may not generally worry about. Think of it as a group of pins that can be collectively manipulated quickly via commands that operate on the entire port.

How do you find 8 consecutive port pins? We'll, if you're starting from scratch, it'll take a bit on investigating. Here's one example. Take a look at the Metro M4 Express schematic (https://adafru.it/EFB) and look in the general area where pin
D13 is shown:

<table>
<thead>
<tr>
<th>Pin</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>D13</td>
</tr>
<tr>
<td>36</td>
<td>D12</td>
</tr>
<tr>
<td>37</td>
<td>D10</td>
</tr>
<tr>
<td>38</td>
<td>D11</td>
</tr>
<tr>
<td>41</td>
<td>D9</td>
</tr>
<tr>
<td>42</td>
<td>D8</td>
</tr>
<tr>
<td>43</td>
<td>D1</td>
</tr>
<tr>
<td>44</td>
<td>D0</td>
</tr>
<tr>
<td>PA16/I2C/I0/SERCOM1.0+3.1</td>
<td>35</td>
</tr>
<tr>
<td>PA17/I2C/I1/SERCOM1.1+3.0</td>
<td>36</td>
</tr>
<tr>
<td>PA18/I2/SERCOM1+3.2</td>
<td>37</td>
</tr>
<tr>
<td>PA19/I3/SERCOM1+3.3</td>
<td>38</td>
</tr>
<tr>
<td>PA20/I4/SERCOM3+5.2/I2SFS0</td>
<td>41</td>
</tr>
<tr>
<td>PA21/I5/SERCOM3+5.3/I2SDO</td>
<td>42</td>
</tr>
<tr>
<td>PA22/I2C/16/SERCOM3.0+5.1/I2SDI</td>
<td>43</td>
</tr>
<tr>
<td>PA23/I2C/I7/SERCOM3.1+5.0/SOF/I2SFS1</td>
<td>44</td>
</tr>
</tbody>
</table>

For example, D13 is wired to physical pin 35 which has several functions internally. The important one to note is PA16. This refers to the digital I/O on Port A at 16. Note that the pins below D13 go consecutively from PA16 to PA23. That’s 8 pins on Port A we can use!

So, for a Metro M4 Express, you could use pins D13, D12, D10, D11, D9, D8, D1, and D0 for your 8 data pins. Then just pick any other 4 for the others.

```python
display_bus = displayio.ParallelBus(data0=board.D13,  
    command=board.D7,  
    chip_select=board.D6,  
    write=board.D5,  
    read=board.D4)
```

Display

To setup a `Display` you need four things:

- A display bus (`display_bus`) for actually talking to the display.
- An initialization sequence (`init_sequence`) to be used to setup the display for initial use.
- The `width` and...
- The `height` of the display in pixels.

In general, you’ll know the `width` and `height` for whatever display you are working with. The `display_bus` is one of the available display buses setup as described above. The `init_sequence` takes a bit of work to come up with. Typically it comes from reading datasheets or other sources. We’ll talk more about this below. For now, just assume you have it created in something called `INIT_SEQUENCE`.

The basic `Display` setup would then look like this:

```python
display = displayio.Display(display_bus,  
    INIT_SEQUENCE,  
    width=320,  
    height=240)
```

Display Drivers

The init sequence (`init_sequence`) is a bit of a cryptic mess. We’ve worked it out for some displays and have created some light weight drivers that take care of the boiler plate. Instead of creating a `Display` object from scratch, you can
use these drivers (and maybe more, check the guide for your display):

- ILI9341 (https://adafru.it/EFC) - color TFTs
- SSD1331 (https://adafru.it/EFD) - color OLEDs
- ST7789 (https://adafru.it/EFE) - wide angle color TFT
- HX8357 (https://adafru.it/EFF) - color TFT
- ST7735R (https://adafru.it/EFG) - color TFT

Then you would create your display like this:

```python
display = adafruit_ili9341.ILI9341(display_bus,
           width=320,
           height=240)
```

Note that it's basically the same as using Display, just without the `init_sequence`. That's taken care of for you.

Boards with Built In Displays

If you have a board like a HalloWing (https://adafru.it/CmY) or a PyPortal (https://adafru.it/pyportal) that already has a display attached, then all this work has been done for you - both the setting up of the display bus and the display itself. The CircuitPython firmware build for these boards has the display ready to go. It is available via the `DISPLAY` object found in the `board` module. All you need to do is:

```python
import board
display = board.DISPLAY
```

Using a Display

Once a Display is setup, use `show` to specify the `Group` to use for displaying items on the screen. Creating a `Group` and the associated `TileGrid`s and `Bitmap`s and `Palette`s has been covered previously in this guide. Once you have your `Group` setup and ready to go, it's just a matter of calling:

```python
display.show(group)
```
Examples

Make sure you have the latest CircuitPython firmware loaded for your board to use displayio. You can download it from the link below.

https://adafruit.it/Em8
https://adafruit.it/Em8

These are some basic examples that cover some common use cases. They are intentionally crude and simple so that just the functional aspects of the displayio library can be seen. A fun thing to do would be to take one of these examples and modify it to try and add something new. Change the text color, make a sprite move around, etc.
What about text? How do you print "Hello World" to the display? Where is text support in `displayio`?!

It's not actually in the core library. Instead, it is provided by a set of external libraries:

- CircuitPython Display Text ([https://adafru.it/EFH](https://adafru.it/EFH)) - for displaying text
- CircuitPython Bitmap Font Library ([https://adafru.it/DZl](https://adafru.it/DZl)) - for bitmap font support

Basic Text with Built in Font

The workhorse item is the `Label`, which is essentially a `Group` containing all the characters of the text. So once it's created, you use it like you would a `Group`.

Creating a `Label` is pretty straightforward - you give it the `text`, `font`, and `color` to use. You specify the text location using `x` and `y`.

In general, you always need to specify a `font` to use for the `text`. The next section shows how to load custom font files. However, a simple built in font is provided so that you can display text without needing a font file. It is available from `terminalio.FONT`.

Here's a basic Hello World example:

```python
import board
import displayio
import terminalio
from adafruit_display_text import label

display = board.DISPLAY

# Set text, font, and color
text = "HELLO WORLD"
font = terminalio.FONT
color = 0x00FF00

# Create the text label
text_area = label.Label(font, text="HELLO WORLD", color=0x00FF00)

# Set the location
text_area.x = 100
text_area.y = 80

# Show it
display.show(text_area)
```
Using Bitmap Fonts

You can also load fonts from external files in Bitmap Distribution Format (.bdf). Check out the Custom Fonts for CircuitPython Displays (https://adafruit.it/EFI) guide for more information about how to create your own BDF font files.

Once you have the BDF font file, copy it to your CIRCUITPY folder somewhere. You then load it using `load_font()` as shown below. The rest is the same as the example above.

```python
import board
importdisplayio
from adafruit_bitmap_font import bitmap_font
from adafruit_display_text import label

display = board.DISPLAY

# Set text, font, and color
text = "HELLO WORLD"
font = bitmap_font.load_font("/Helvetica-Bold-16.bdf")
color = 0xFF00FF

# Create the text label
text_area = label.Label(font, text=text, color=color)

# Set the location
text_area.x = 20
text_area.y = 20

# Show it
display.show(text_area)
```

To run the example above, you'll need this font file:

https://adafruit.it/EFJ

Note: the font file should be on the board’s CIRCUITPY flash drive. In the example above, the main (root) directory `/` is used, but in some tutorials, a new subdirectory `/font` is created for font files. Just be sure your `load_font` has the correct directory to the font file in your project.

Text Origin

When setting text location, you are setting the origin point. It is located relative to the text as shown below.
Display a Bitmap

Make sure you have the latest CircuitPython firmware loaded for your board to use displayio. You can download it from the link below.

https://adafruit.it/Em8

This example shows how to load and display a bitmap (.bmp) file. We will still need to create all the pieces - a TileGrid, a Group, etc. But they can be used in their most simple way.

OnDiskBitmap

First, let's use OnDiskBitmap to source the bitmap image directly from flash memory storage. This is like reading the image from disk instead of loading it into memory first (we'll do that next). The trade off here is the reduced use of memory for potentially slower pixel draw times.

We'll use a 320x240 pixel image. Here's the image:

https://adafruit.it/EFK

Here's the code:
import board
import displayio

display = board.DISPLAY

# Open the file
with open("/purple.bmp", "rb") as bitmap_file:
    # Setup the file as the bitmap data source
    bitmap = displayio.OnDiskBitmap(bitmap_file)

    # Create a TileGrid to hold the bitmap
    tile_grid = displayio.TileGrid(bitmap, pixel_shader=displayio.ColorConverter())

    # Create a Group to hold the TileGrid
    group = displayio.Group()

    # Add the TileGrid to the Group
    group.append(tile_grid)

    # Add the Group to the Display
    display.show(group)

    # Loop forever so you can enjoy your image
    while True:
        pass

ImageLoad

This approach uses the CircuitPython Image Load (https://adafruit.it/EFL) library to load the image into memory and then display it. Using the same image from above, here's the code:
import board
import displayio
import adafruit_imageload

display = board.DISPLAY

bitmap, palette = adafruit_imageload.load("/purple.bmp",
    bitmap=displayio.Bitmap,
    palette=displayio.Palette)

# Create a TileGrid to hold the bitmap
tile_grid = displayio.TileGrid(bitmap, pixel_shader=palette)

# Create a Group to hold the TileGrid
group = displayio.Group()

# Add the TileGrid to the Group
group.append(tile_grid)

# Add the Group to the Display
display.show(group)

# Loop forever so you can enjoy your image
while True:
    pass
Do you want to just set a specific pixel to a specific color? Here's how. Most of the code is the setup of necessary parts - the **TileGrid**, **Palette**, and **Group**. But once everything is setup, you can access pixels with the simple syntax:

```
bitmap[x, y] = color_value
```

Remember that `color_value` is not an actual color, but a reference to the associated **Palette**.

Here's a full example:

```python
import board
import displayio

display = board.DISPLAY

# Create a bitmap with two colors
bitmap = displayio.Bitmap(display.width, display.height, 2)

# Create a two color palette
palette = displayio.Palette(2)
palette[0] = 0x000000
palette[1] = 0xffffff

# Create a TileGrid using the Bitmap and Palette
tile_grid = displayio.TileGrid(bitmap, pixel_shader=palette)

# Create a Group
group = displayio.Group()

# Add the TileGrid to the Group
group.append(tile_grid)

# Add the Group to the Display
display.show(group)

# Draw a pixel
bitmap[80, 50] = 1

# Draw even more pixels
for x in range(150, 170):
    for y in range(100, 110):
        bitmap[x, y] = 1
```
This example is simple, but shows a basic usage of what makes TileGrid so cool. While you can create a TileGrid with multiple tiles, you can also create a TileGrid with just a single tile. This special case is often referred to as a "sprite". You still need a source Bitmap for the TileGrid. So we use one that contains several sprites all arranged nicely. This is called a "sprite sheet".

Here's a nice little sprite sheet bitmap we will use for this example:

It's super tiny! If you zoom in, it looks like this:

Let's add a grid overlay to better show each individual pixel:
You can see how there are 6 total sprites. Each sprite is 16 pixels wide by 16 pixels high. We want to slice it up like this:

Remember, this is not the TileGrid. This is just how the source bitmap is sliced up to provide source tiles for the TileGrid.

We will create a TileGrid with only one tile (sprite). We can then change the index of this TileGrid to be whichever of these 6 characters from the source bitmap (sprite sheet) we want to show. And we can change it again to show a different one, etc.

We have everything we need:

- A source Bitmap - the sprite sheet
- We know each sprite is 16 pixels by 16 pixels
  - tile_width = 16
- tile_height = 16

- We know we want a TileGrid that is only 1 wide by 1 high
  - width = 1
  - height = 1

Here is what creating the TileGrid would look like to set this up:

```python
sprite = displayio.TileGrid(sprite_sheet, pixel_shader=palette,
    width = 1,
    height = 1,
    tile_width = 16,
    tile_height = 16)
```

By default, the source index will be 0 to start with. So the sprite is set to show Blinka (the purple snake). You can change it to any of the other 6 sprites using the syntax:

```python
sprite[0] = 1
```

Now the sprite is set to show Adabot - index 1. Note that the sprite index is [0], which is the only index that applies in this case, since there's only 1 tile in the TileGrid.

Want Sparky? Do this:

```python
sprite[0] = 4
```

And so on.

**Sprite Sheet Example**

Here's the full code. It loads the source Bitmap, sets up the TileGrid, adds it to a Group, which is then added to the Display so it's finally shown. It then cycles through each of the sprites.
import time
import board
import displayio
import adafruit_imageload

display = board.DISPLAY

# Load the sprite sheet (bitmap)
sprite_sheet, palette = adafruit_imageload.load("/cp_sprite_sheet.bmp",
                                bitmap=displayio.Bitmap,
                                palette=displayio.Palette)

# Create a sprite (tilegrid)
sprite = displayio.TileGrid(sprite_sheet, pixel_shader=palette,
                                  width = 1,
                                  height = 1,
                                  tile_width = 16,
                                  tile_height = 16)

# Create a Group to hold the sprite
group = displayio.Group(scale=1)

# Add the sprite to the Group
group.append(sprite)

# Add the Group to the Display
display.show(group)

# Set sprite location
group.x = 120
group.y = 80

# Loop through each sprite in the sprite sheet
source_index = 0
while True:
    sprite[0] = source_index % 6
    source_index += 1
    time.sleep(2)

Change The Scale!

The sprites are pretty small. The default scale is 1, so each pixel of the sprite is a pixel on the display. You can change this using the `scale` parameter which is passed in when creating the `Group`.

Try changing that to something like 2 or 4 and running the code again. It's this line of code:

```python
group = displayio.Group(scale=1)
```

Now the sprites should show up much larger!

Change The Location!

Want the sprites to show up in a different location? You can do so by changing these lines and setting new values for `x` and `y`.
group.x = 120
group.y = 80
This example builds on the Sprite Sheet example to show a more sophisticated usage of **TileGrid**. We'll show how you can have more than one TileGrid and that a TileGrid can be more than just one tile.

The castle wall tiles used in this example were borrowed from this excellent tilesheet: [dungeon_tileset-ii](https://adafruit.it/EFN)

**A Sprite and Its Castle**

Our first TileGrid will be another sprite - so a TileGrid with a single tile. This is the same as was done in the Sprite Sheet example. Our second TileGrid will be a little more interesting. It will have more than one tile and will be used to generate the walls and floor of a 2D castle for our sprite to live in.

The idea is to generate the walls and floors by reusing the same source tile over and over. For example, we can create something that looks like this:

![TileGrid Example](image)

You can kind of already see the grid like repetitive pattern. Let's put a reference grid over the top:
This grid is 6 tiles wide by 5 tiles high. You can see how the floor is just the same tile over and over. The walls can similarly be created by reusing the same source tile. So we just need a source bitmap that has each of these basic building pieces. It can come from the same bitmap we'll use for our sprite. Let's do that - here's our new sprite sheet we will work with:

https://adafru.it/EFO
https://adafru.it/EFO

Another super tiny BMP! Here's what it looks like more blown up:
There are a couple of characters we can use for our sprite at the top. But there's also the basic building blocks needed for our castle.

For this example, each item is 16 pixels by 16 pixels. So we'll end up carving up the sprite sheet like this:

As mentioned above, our castle is 6 tiles wide by 5 tiles high. So that will be the size of the `TileGrid` we'll create to generate the castle. Then, each tile in the castle `TileGrid` just needs to be set to the correct index from the source bitmap.

Here's what that would look like:
But keep in mind this is only one of the TileGrids we'll create. The other is our simple single tile TileGrid - the sprite. It comes from the same sprite sheet.

Think of it working like this:

And we can assign the single tile of sprite or any of the tiles of castle to any of the indices from sprite_sheet. The (x, y) notation for a couple of tiles in castle are shown as a helpful reminded of how they are accessed.

For example, to set the lower right corner located at (5, 4) of the castle to the "lower right corner" graphic found at index 11 in the sprite_sheet, do this:

```
castle[5, 4] = 11
```

But of course we need to set all of the tiles. This just ends up being more lines of code.
Here's the full code:

```python
import board
import displayio
import adafruit_imageload

display = board.DISPLAY

# Load the sprite sheet (bitmap)
sprite_sheet, palette = adafruit_imageload.load("/castle_sprite_sheet.bmp",
                                              bitmap=displayio.Bitmap,
                                              palette=displayio.Palette)

# Create the sprite TileGrid
sprite = displayio.TileGrid(sprite_sheet, pixel_shader=palette,
                             width = 1,
                             height = 1,
                             tile_width = 16,
                             tile_height = 16,
                             default_tile = 0)

# Create the castle TileGrid
castle = displayio.TileGrid(sprite_sheet, pixel_shader=palette,
                             width = 6,
                             height = 5,
                             tile_width = 16,
                             tile_height = 16)

# Create a Group to hold the sprite and add it
sprite_group = displayio.Group()
sprite_group.append(sprite)

# Create a Group to hold the castle and add it
castle_group = displayio.Group(scale=3)
castle_group.append(castle)

castle_group.append(sprite_group)

# Castle tile assignments
# corners
castle[0, 0] = 3  # upper left
castle[5, 0] = 5  # upper right
castle[0, 4] = 9  # lower left
castle[5, 4] = 11 # lower right
# top / bottom walls
for x in range(1, 5):
    castle[x, 0] = 4  # top
    castle[x, 4] = 10 # bottom
# left/ right walls
for y in range(1, 4):
    castle[0, y] = 6 # left
castle[5, y] = 8 # right
```

© Adafruit Industries  https://learn.adafruit.com/circuitpython-display-support-using-displayio
for x in range(1, 5):
    for y in range(1, 4):
        castle[x, y] = 7 # floor

# put the sprite somewhere in the castle
sprite.x = 110
sprite.y = 70

# Add the Group to the Display
display.show(group)

If you run that, you should end up with something like this:

![Image of a castle floor with a sprite]

Order Matters

Note the order in which the `sprite_group` and the `castle_group` were added to the main group that was finally shown on the display.

```python
group.append(castle_group)
group.append(sprite_group)
```

Think of it as building from the bottom up or outward from the display. Each new item will be shown above the previous items. Since we want our sprite to be seen above the castle, we add (append) it after we add the castle.

Using Different Scale

This example shows how you can mix different scales if you want. Since `scale` is used at the `Group` level and applies to everything in the `Group`, we created two separate Groups for the sprite and castle. That way we could set a different scale for the castle.
You don't have to do this. We could have just added the sprite and castle to the same Group. But this shows how there is flexibility in how you setup your collection of items that you send to the display.

Change The Sprite

Want Adabot to be in the castle instead of Blinka? All you need to do is change the source index for the sprite tile. There are two ways you could do this.

The first would be to use the default_tile parameter assignment when creating the TileGrid. In the code above, it was set to 0. If you wanted Adabot, you would change it to 1.

The second way would be to just set it after the TileGrid is created. That would look like this:

```python
sprite[0] = 1
```

Change Sprite Location

Want Blinka to be somewhere else in the castle? Simple, just change the x and y values here:

```python
sprite.x = 110
sprite.y = 70
```

Even more fun - write a loop with these changing inside the loop. Then Blinka will be moving around in the castle!
This example shows how to use a display on a breakout board using a SPI interface.

The Hard Way

Assuming you read through the datasheet(s) and somehow came up with the initialization sequence you needed for your display, you could do something like this.

This example was tested using a 2.4" TFT breakout wired to an Itsy Bitsy M4's hardware SPI pins. See here for display wiring information:

```python
import board
import displayio

# Release any previously configured displays
displayio.release_displays()

# Setup SPI bus
spi_bus = board.SPI()

# Digital pins to use
tft_cs = board.D10
tft_dc = board.D9

# Setup the display bus
display_bus = displayio.FourWire(spi_bus, command=tft_dc, chip_select=tft_cs)

# Setup the initialization sequence
# stolen from adafruit ili9341.py
INIT_SEQUENCE = (
    b'\x01\x80\x80'            # Software reset then delay 0x80 (128ms)
    b'\xEF\x03\x03\x80\x02'
    b'\xCF\x03\x00\xC1\x30'
    b'\xED\x04\x64\x03\x12\x81'
    b'\xE8\x03\x85\x00\x78'
    b'\xCB\x05\x39\x2C\x00\x34\x02'
    b'\xF7\x01\x20'
    b'\xEA\x02\x00\x00'
    b'\xC0\x01\x23'            # Power control VRH[5:0]
    b'\xC1\x01\x10'            # Power control SAP[2:0];BT[3:0]
    b'\xC5\x02\x3e\x28'        # VCM control
    b'
')
```

Make sure you have the latest CircuitPython firmware loaded for your board to use displayio. You can download it from the link below.

© Adafruit Industries  https://learn.adafruit.com/circuitpython-display-support-using-displayio  Page 35 of 46
The Easy Way

Use a driver instead. This will take care of the initialization sequence for you. Here we use the ILI9341 driver (https://adafru.it/EFC).
import board
import displayio
import adafruit_ili9341

# Release any previously configured displays
displayio.release_displays()

# Setup SPI bus
spi_bus = board.SPI()

# Digital pins to use
tft_cs = board.D10
tft_dc = board.D9

# Setup the display bus
display_bus = displayio.FourWire(spi_bus, command=tft_dc, chip_select=tft_cs)

# Setup the Display
display = adafruit_ili9341.ILI9341(display_bus, width=320, height=240)

# DONE - now you can use the display however you want

bitmap = displayio.Bitmap(320, 240, 2)
palette = displayio.Palette(2)
palette[0] = 0
palette[1] = 0xFFFFFF

for x in range(10, 20):
    for y in range(10, 20):
        bitmap[x, y] = 1

tile_grid = displayio.TileGrid(bitmap, pixel_shader=palette)

group = displayio.Group()
group.append(tile_grid)
display.show(group)
display.refresh_soon()

For displays in the Adafruit shop, there should be (or will be) a CircuitPython driver for each one. One driver might handle more than one product. They are designated by chipset number, like ILI9341. Look at the Adafruit product page to see which chipset the Adafruit product uses and then look for the corresponding Adafruit CircuitPython driver. Then you do not have to deal with the low level initialization.

If you have a non-Adafruit display, you might be able to use an existing CircuitPython driver if it uses the same chipset as one of the available drivers. This isn't guaranteed though as a manufacturer might have made changes to a board not supplied by Adafruit. So you might need to experiment more to see if the code may work. This isn't to get you to buy Adafruit's displays, more like a friendly note that more tinkering may be needed as things may not be tested out like Adafruit displays.

The CircuitPython team encourages contributors to add drivers for displays not currently handled in the CircuitPython library bundle. If you write a driver, it can be shared with others with the same display, contributing back to the community. Isn't Open Source helpful? We think so.
UI Quickstart

There are several User Interface (UI) elements available to use with displayio. You can use these together to create all kinds of fun applications such as a calculator.

Groups

Groups ([https://adafru.it/EFx](https://adafru.it/EFx)) are a way for displayio to keep track of all of the elements that it needs to draw. Subgroups can be inside of groups, but you must have at least one main group. For all of the elements on this page, you must first import displayio, so we'll start with making sure that's at the top of your file.

```python
import displayio
```

You must also create at least one group that will be used as your main group.

```python
my_display_group = displayio.Group(max_size=25)
board.DISPLAY.show(my_display_group)
```

Shapes

The shapes ([https://adafru.it/Fiu](https://adafru.it/Fiu)) are part of the adafruit_display_shapes ([https://adafru.it/Fiv](https://adafru.it/Fiv)) library. At the time of this writing, there are three shapes available. They work by generating a bitmap in the specific shape using displayio.

Rectangle

The rectangle is your most basic shape. It can either be filled, outlined, or both.
Rounded Rectangle
The rounded rectangle is a little more complex and is comprised of 4 lines and quarter circle corners.

Circle
The circle is based on the rounded rectangle and only draws the four corners without any width or height.

Triangle
The triangle allows you to supply three sets of coordinates and will either draw an outline between those vertices, fill it in, or both.

To use shapes, you first need to import the shapes you want to use at the top of your file. For instance if you wanted to separate import all the shapes, you would add something like this.

```python
from adafruit_display_shapes.rect import Rect
from adafruit_display_shapes.circle import Circle
```
from adafruit_display_shapes.roundrect import RoundRect
from adafruit_display_shapes.triangle import Triangle

Next, you can draw a rectangle (https://adafruit.it/Fiu) with something like:

```python
rect = Rect(0, 0, 80, 40, fill=0x00FF00)
```

For a circle (https://adafruit.it/Fiu), you can create it with something like:

```python
circle = Circle(100, 100, 20, fill=0x00FF00, outline=0xFF00FF)
```

For a triangle (https://adafruit.it/Fiu), you can create it with something like:

```python
triangle = Triangle(170, 50, 120, 140, 210, 160, fill=0x00FF00, outline=0xFF00FF)
```

Or you can draw a rounded rectangle (https://adafruit.it/Fiu) with something like:

```python
roundrect = RoundRect(50, 100, 40, 80, 10, fill=0x0, outline=0xFF00FF, stroke=3)
```

Finally, you can add all of these shapes to your group.

```python
my_display_group.append(rect)
my_display_group.append(circle)
my_display_group.append(triangle)
my_display_group.append(roundrect)
```

Fonts

For fonts, there are a couple options that you can use. You can create or provide a custom font file and use that for your label. If you don't want to provide a custom bitmap font, you can use the Built-in Terminal Font.

Built-in Terminal Font

The terminal font looks a little blocky, but at least you don't need a separate file.

This example comes from the PyBadger Event Badge (https://adafruit.it/Fiw) guide.

To use the Terminal Font, you first need to import `terminalio` by adding an import statement to the top of your file.

```python
import terminalio
```

Then you simply pass the terminalio font to the UI element. For instance, with the label, you would do something like:
my_label = Label(terminalio.FONT, text="My Label Text", color=BLACK)

Bitmap Fonts
The Bitmap font (https://adafru.it/Fix) uses the adafruit_bitmap_font (https://adafru.it/Fiv) library and requires a separate BDF (Bitmap Distribution Format) file, but looks nicer on the screen. It doesn't currently have anti-aliasing, so it still looks a little blocky on some fonts.

This example comes from the PyBadge Conference Badge With Unicode Fonts (https://adafru.it/Fy) guide.

To use a Bitmap Font, you first need to copy your custom file over to your CIRCUITPY drive. We like to place fonts into a /fonts folder. To find out more about creating your own custom fonts, be sure to check out our Custom Fonts for CircuitPython Displays (https://adafru.it/E7E) guide.

Next import bitmap_font by adding an import statement to the top of your file.

from adafruit_bitmap_font import bitmap_font

After that, you can create a font instance. For example, if you have a font file named Arial-12.bdf in the fonts folder, you would use the following line of code.

font = bitmap_font.load_font("/fonts/Arial-12.bdf")

Then you simply pass font instance to the UI element. For instance, with the label, you would do something like:

my_label = Label(font, text="My Label Text", color=BLACK)

Label
The label (https://adafru.it/Fiz) requires the adafruit_display_text (https://adafru.it/FiA) library. It requires a font to be passed in. This can either be the Terminal Font or a Custom Font. It allows you to display text and place it in your displayio group. The conference badge mentioned in the fonts section makes great use of labels.

To create a label, you first import the required library at the top of your file.

from adafruit_display_text.label import Label

Then you create the label.

my_label = Label(terminalio.FONT, text="My Label Text", color=BLACK)

Finally, add the label to a displayio group. This can either be your main group or a subgroup.

my_display_group.append(my_label)
Button

The button (https://adafruit.it/FiB) makes use of the adafruit_button (https://adafruit.it/FiC) library and builds on top of the adafruit_display_shapes, adafruit_label, and adafruit_touchscreen (https://adafruit.it/FiD) libraries. A button is basically a shape and label together which can also handle presses as well as color inversion.

To use the button, you need to add the required libraries to the top of your file.

```python
from adafruit_button import Button
import adafruit_touchscreen
```

Next create your button. There are lots of options and you can take a look at some of the examples provided in the button library to get an idea of the various things you can do.

```python
my_button = Button(x=20, y=20, width=80, height=40,
                   label="My Button", label_font=terminalio.FONT)
```

The font is required, but again, you can provide either the built-in font or a custom font. Finally add it to your group.

```python
my_display_group.append(my_button)
```

Here's what the simple test example looks like showing many different variations.

Images

Images are also available, although they are not used in this calculator project. There are a couple of different ways to display images with displayio.

ImageLoad Library

ImageLoad (https://adafruit.it/FiE) is the main class in the adafruit_imageload (https://adafruit.it/EFL) library provides an easy way to decode and display bitmaps. At the moment, it is only able to decode indexed 8-bit bitmaps. To use it, first
include the library at the top of your file.

```python
import adafruit_imageload
```

Second, Generate the Bitmap and Palette from the image:

```python
my_bitmap, my_palette = adafruit_imageload.load("/my_bitmap.bmp", bitmap=displayio.Bitmap,
    palette=displayio.Palette)
```

Third, create a TileGrid from the Bitmap and Palette:

```python
my_tilegrid = displayio.TileGrid(my_bitmap, pixel_shader=my_palette)
```

Finally add the TileGrid to your display group.

```python
my_display_group.append(my_tilegrid)
```

OnDiskBitmap

OnDiskBitmap ([https://adafru.it/EFI](https://adafru.it/EFI)) is available directly through displayio and is very easy to use. The first step is to open the image file with `read` and `binary` modes and create a bitmap. Because of its flexibility and low memory use, this is the recommended way.

```python
my_bitmap = displayio.OnDiskBitmap(open("/my_bitmap.bmp", "rb"))
```

The second step is to create a TileGrid from the image using the automatic color converter.

```python
my_tilegrid = displayio.TileGrid(my_bitmap, pixel_shader=displayio.ColorConverter())
```

Finally add the TileGrid to your display group.

```python
my_display_group.append(my_tilegrid)
```

Calculator UI Elements

The PyPortal Calculator makes use of the Rectangle, Label, and Button Elements.