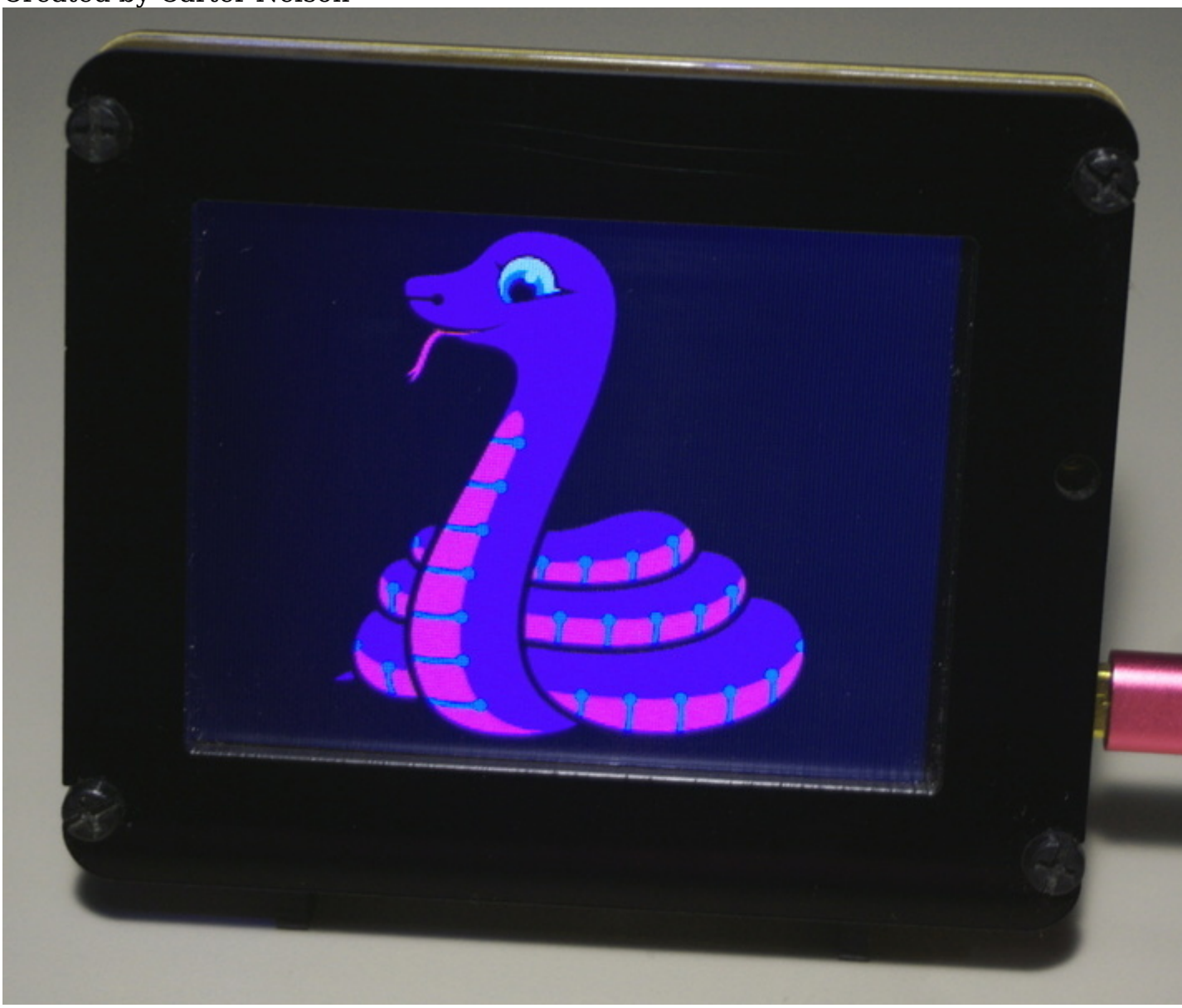
```
Error: Can't find stylesheet to import.
  ╷
4 │ @import "gist";
  │         ^^^^^^
  ╵
  app/assets/stylesheets/application.pdf.scss 4:9  root stylesheet
```

CircuitPython Display Support Using displayio

Created by Carter Nelson



https://learn.adafruit.com/circuitpython-display-support-using-displayio
Last updated on 2024-03-28 07:55:30 PM EDT

Table of Contents

- Changing Text

Display a Bitmap

- BMP File Format
- OnDiskBitmap
- ImageLoad

Draw Pixels

Sprite Sheet

- Sprite Sheet Example
- Change The Scale!
- Change The Location!

Multiple TileGrids

- A Sprite and Its Castle
- Order Matters
- Using Different Scale
- Change The Sprite
- Change Sprite Location

External Display

- The Hard Way
- The Easy Way

Manual Refresh

- Turning Auto Refresh Off
- Example Usage
- Turning Auto Refresh On

UI Quickstart

- Referencing the Display
- Groups
- Shapes
- Fonts
- Label
- Button
- Images
- Calculator UI Elements

Helper Libraries

- The List

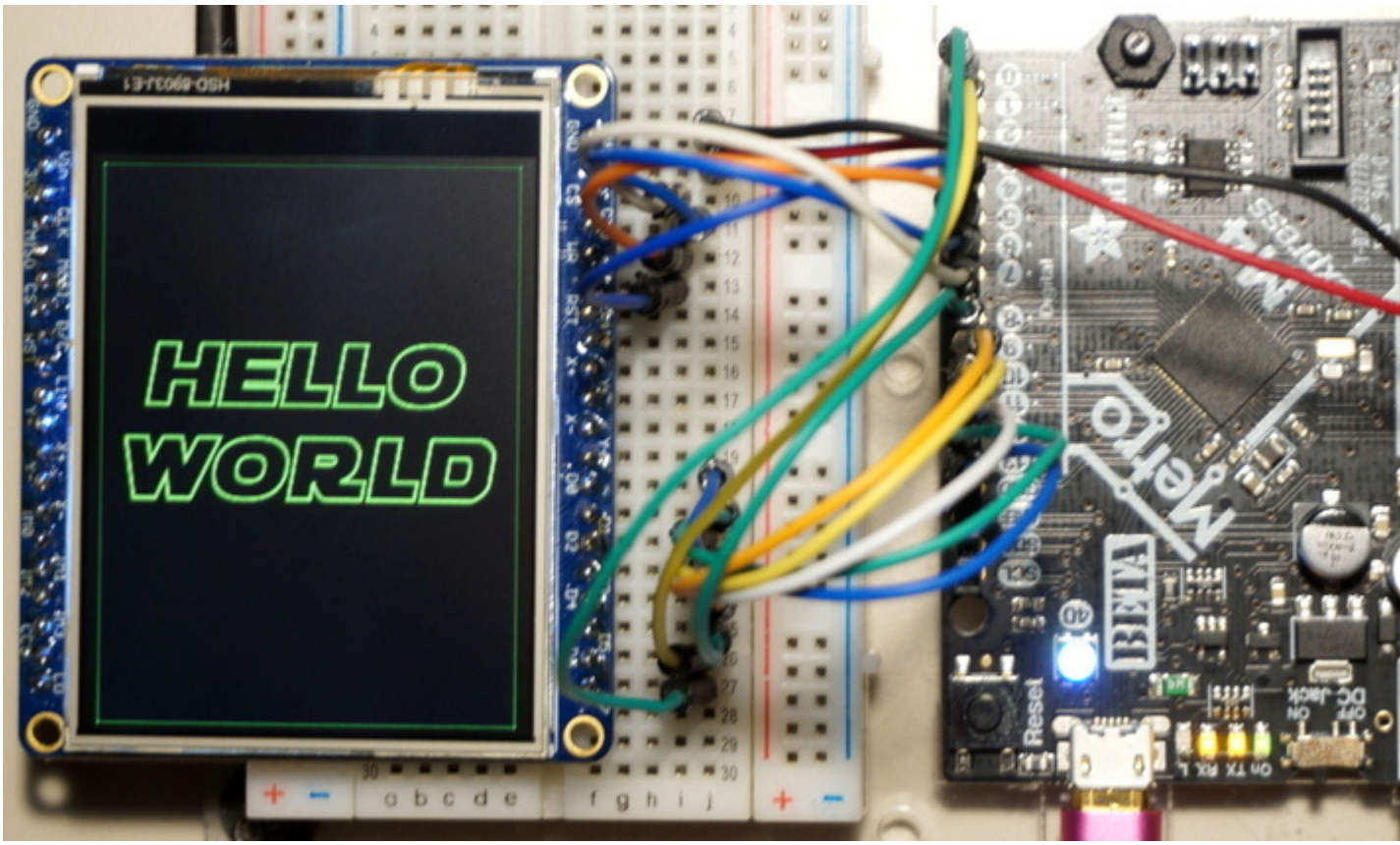
FAQs

Introduction

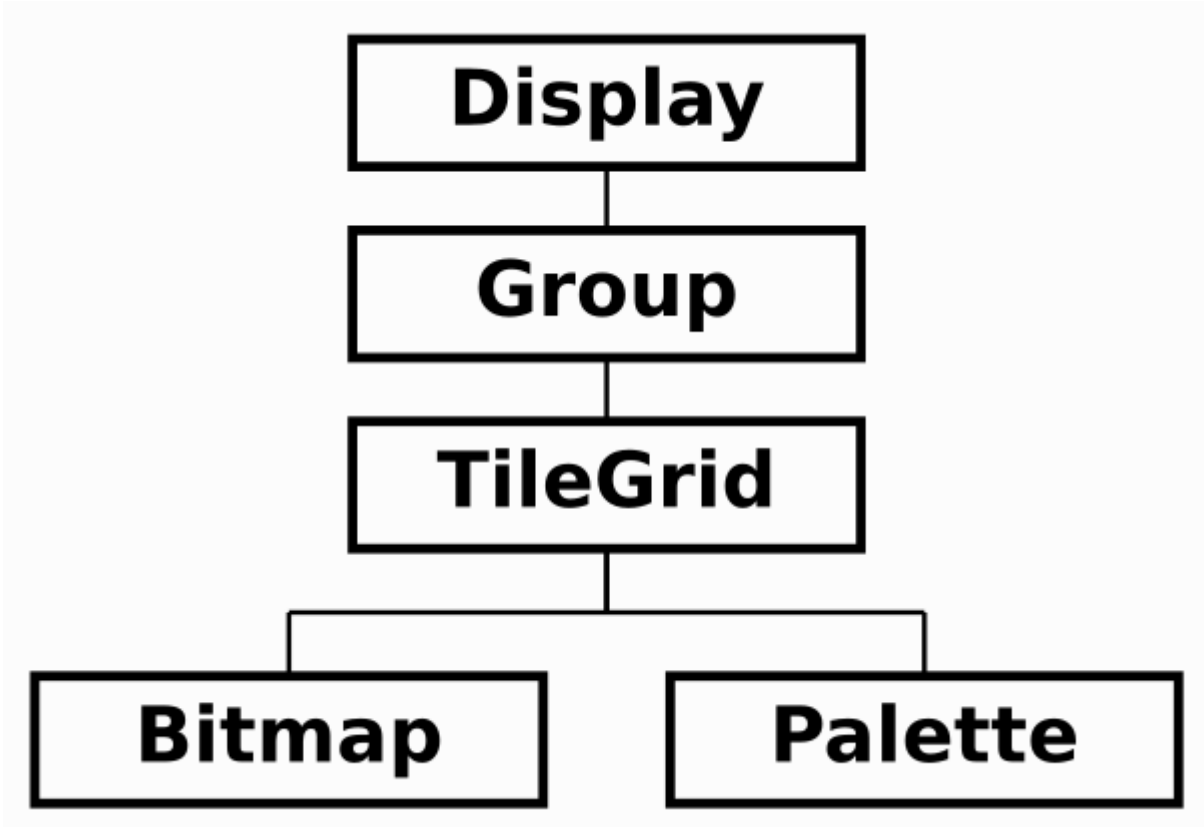
CircuitPython has native support for displays with the **displayio (https://adafru.it/EFr)** built-in module This library provides the support needed for drawing to graphical displays. It allows for some common tasks like displaying bitmap images, drawing text with fonts, etc. However, there are also some fancy additional features that provide the framework for creating extended functionality.

This guide will go over the main aspects of the **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:

displayio API Documentation
https://adafru.it/19ej

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://adafru.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://adafru.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://adafru.it/EFu) - This is a simple list of colors. A **Bitmap**'s pixel value is an index into this list.
- **ColorConverter** (https://adafru.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://adafru.it/EFw) - This uses a **Bitmap** and a pixel shader (**Palette**) to draw actual pixels. It must be added to a **Group**.
- **Group** (https://adafru.it/EFx) - This is a collection of one or more **TileGrid**s. It can also contain other **Group**s.

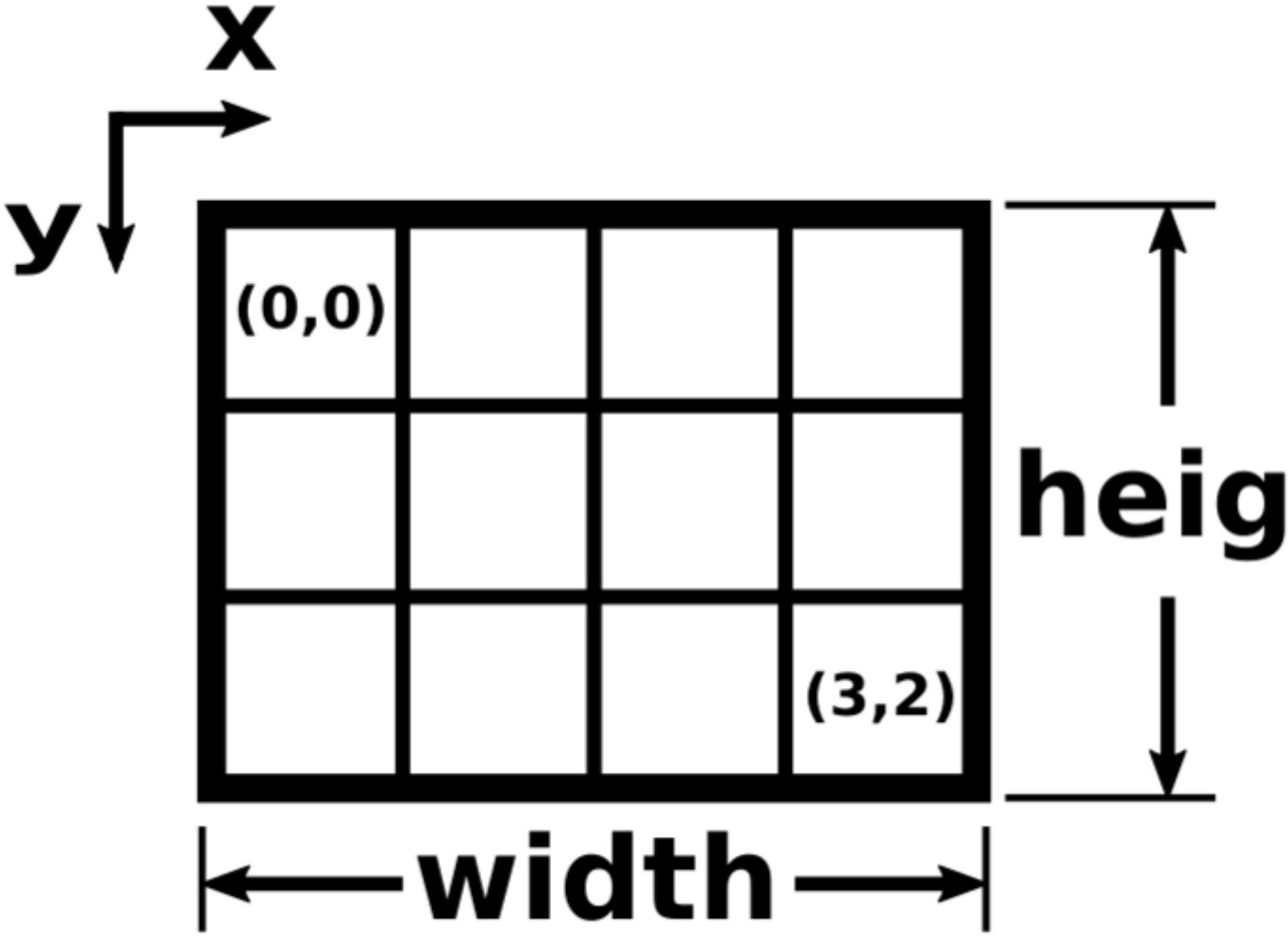
Display Hardware Things

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

- **BusDisplay** (https://adafru.it/19IB)- This is the actual display. It must be connected to the host controller via a "display bus".
- **FourWire** (https://adafru.it/19IC) - A SPI based display bus.
- **ParallelBus** (https://adafru.it/19ek) - An 8-bit parallel display bus.
- **I2CDisplayBus** (https://adafru.it/19ID) - An I2C based display bus.
- **EPaperDisplay** (https://adafru.it/19IE) - An EPaper or E-Ink style display.

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`:



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)`.

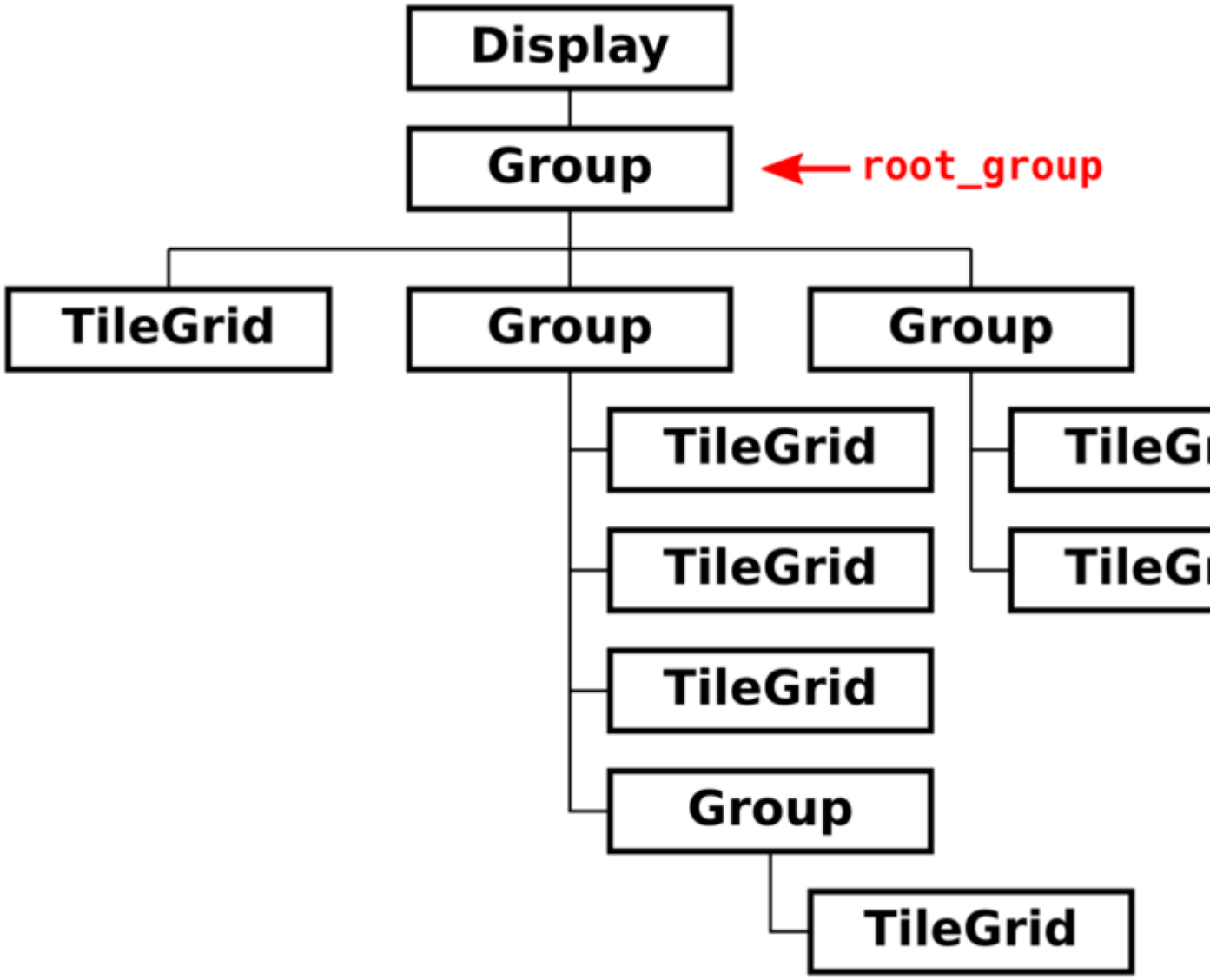
Hierarchy and Nesting

The image at the top of this page is an example of the most simple arrangement of **displayio** objects. It's a good visual reference for the general hierarchy. However, much more complex arrangements are possible. Keep in mind these general rules:

- A **Display** can only show a **Group**.

- A **Display** can only show one **Group** at any time. This is called the **root group**.
- A **Group** can contain one or more **TileGrid**s as well as one or more **Group**s.

So, for example, you could have something like the arrangement shown below. The **Bitmap** and **Palette** associated with each **TileGrid** have been left out for simplicity.



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.

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

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

```
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 straight forward. 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:

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

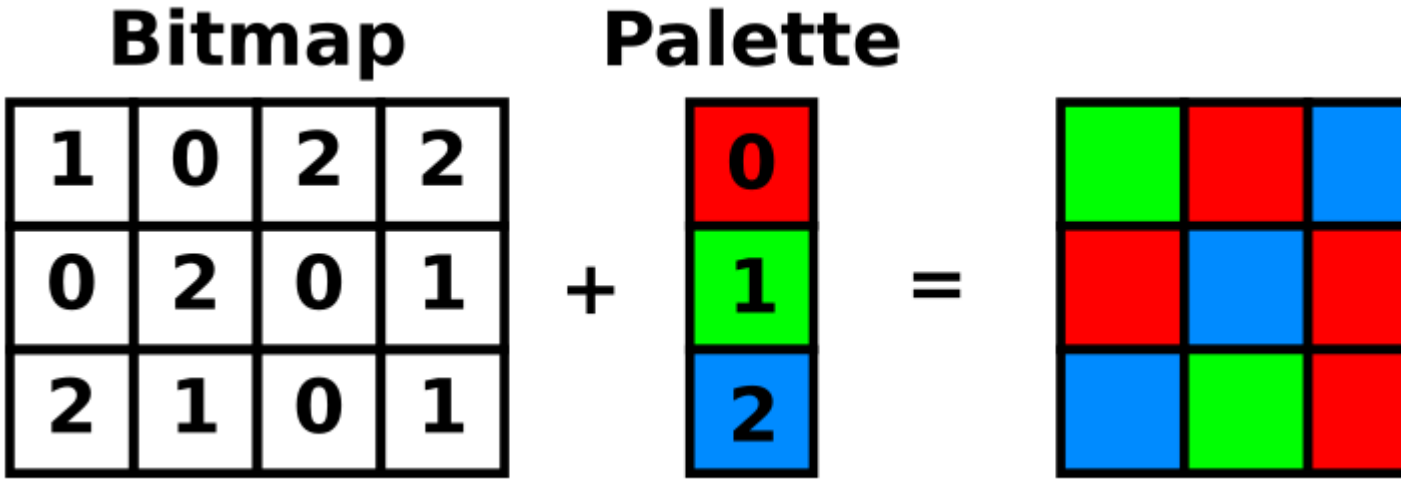
Here is how you would specify the color for each entry:

```
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:



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.root_group = 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.

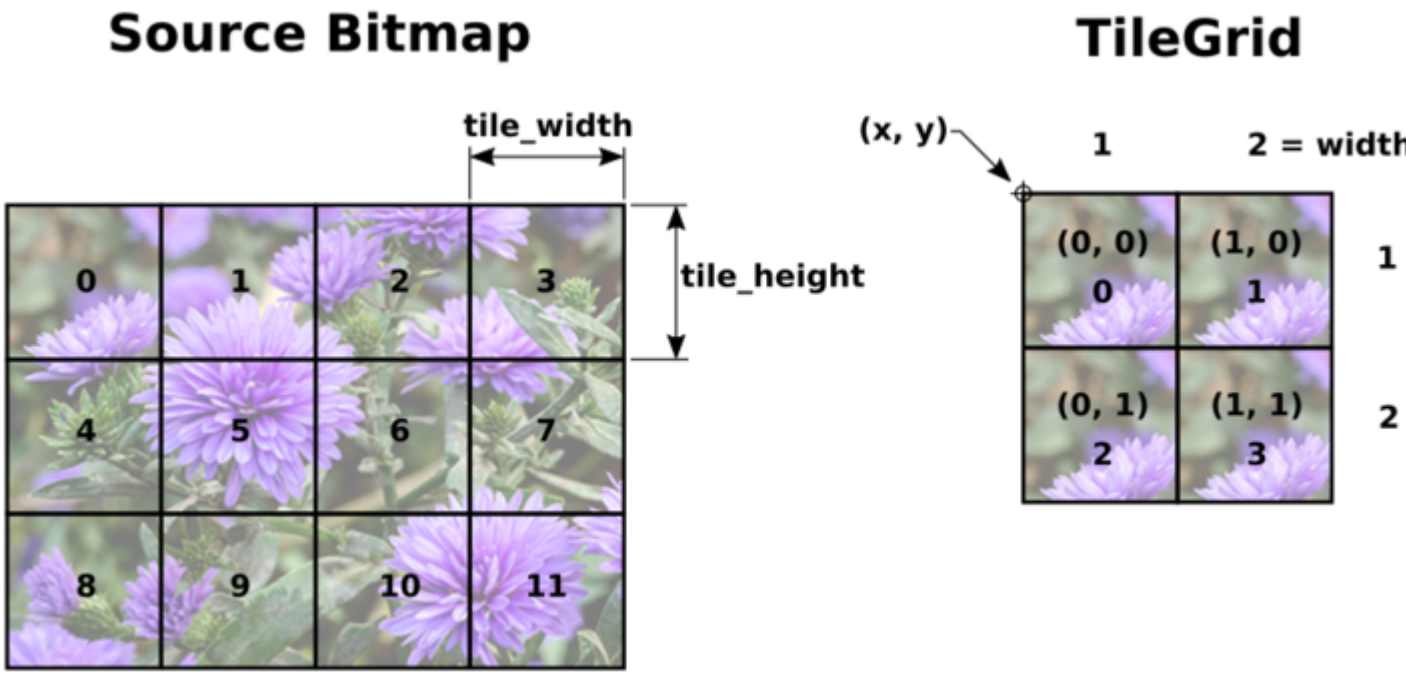
TileGrid

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:

```
tile_grid[tile_grid_index] = 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.

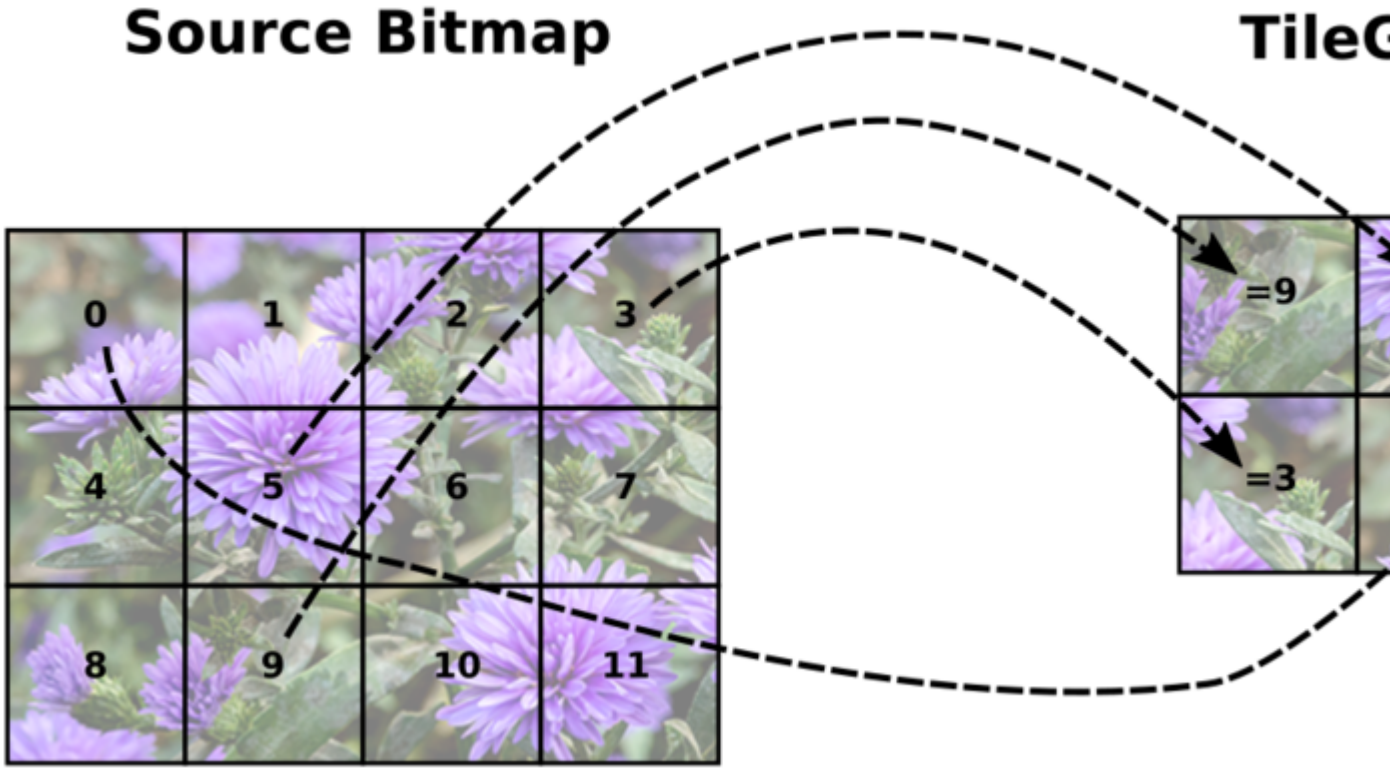
Using our example from above, if we did something like this:

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

or this, which uses the other notation:

```
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**.

Group

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 Adding/Removing

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

Add items to the **Group** using `append()` or `insert()`. The `append()` command adds the item to the end while the `insert()` command allows specifying an index location. You can also change an item directly using index notation, ex: `group[index] = tilegrid`.

Remove items using `pop()` or `remove()`. The `pop()` command can take an index to pop the ith item, otherwise it removes the last one. The `remove()` command allows specifying the specific item within the **Group** to remove.

Group Position

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).

Group Scale

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.

Group Visibility

The `hidden` property of the **Group** can be used to set whether the contents of the Group are to be shown or not. Set `True` to hide (won't be shown) or `False` to show. By default, this is set to `False` so items are shown.

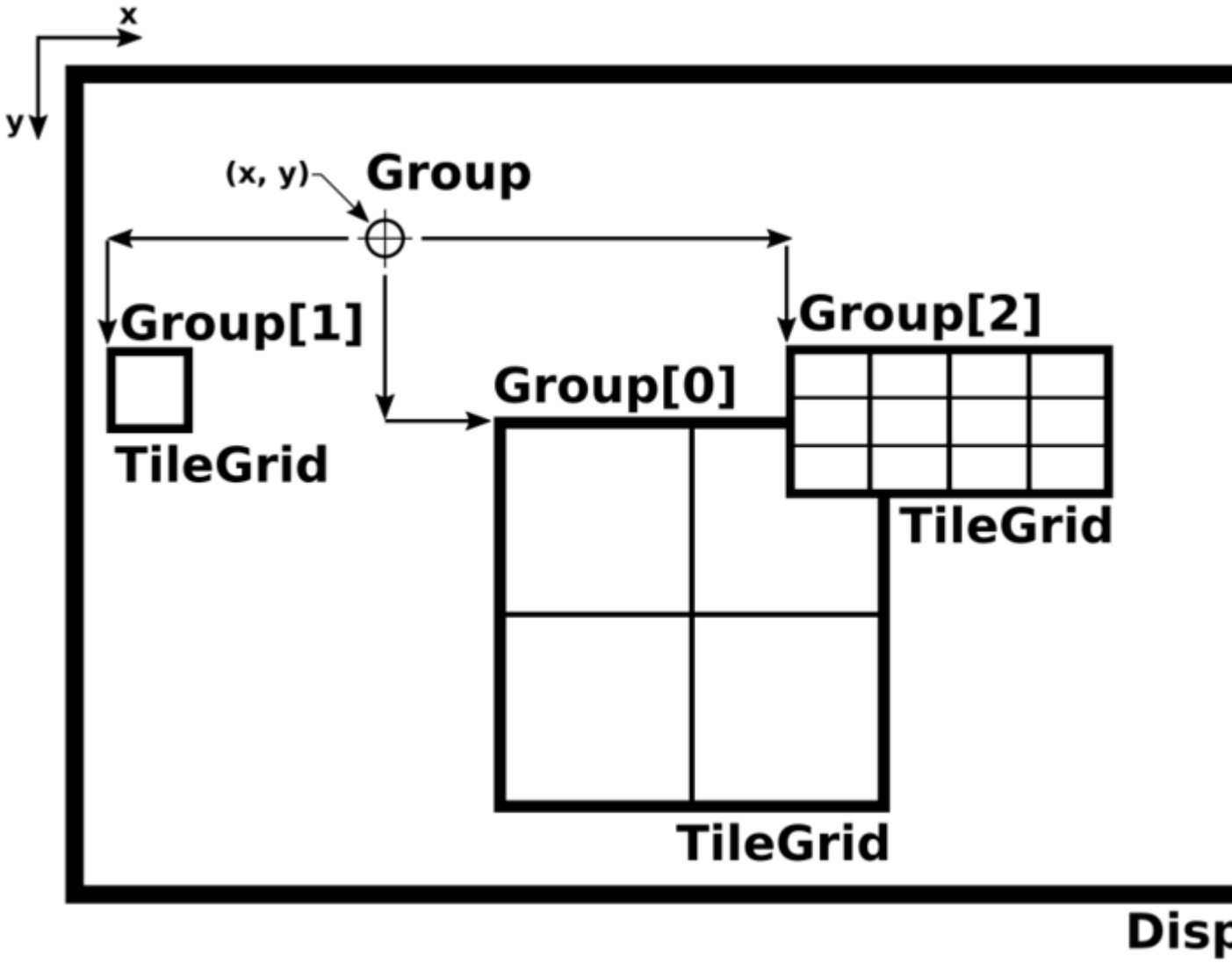
Group Content Limits

In older versions of CircuitPython **Group** was restricted to a maximum number of items. That number was specified by the user code so enough memory space would be created.

However, newer releases allow updating the group to grow as new items are appended so there isn't a specific maximum any longer.

Summary

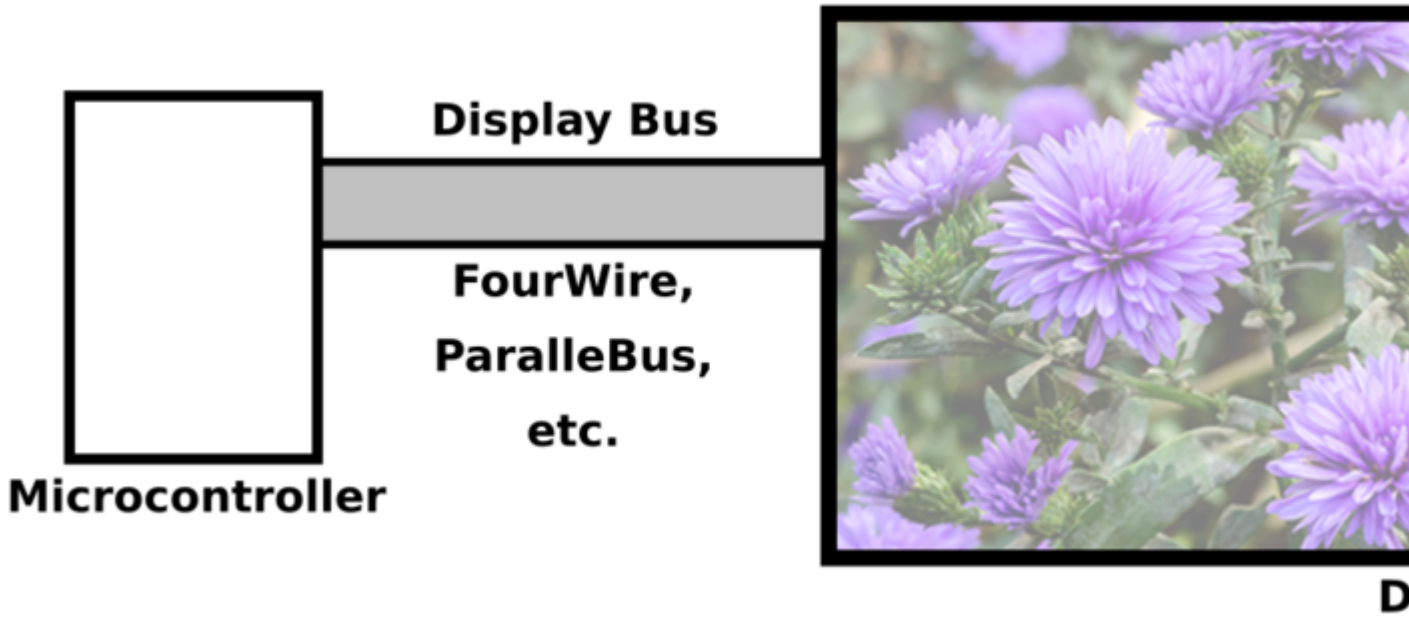
The **Group** is what we will finally show on our **Display**. The end result looks something like this:



This shows a notional **Group** located at **(x, y)** on the **Display**. It contains 3 **TileGrid**s 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**.

Display and Display Bus

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**, **ParallelBus**. etc.



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.

The display bus classes were moved from displayio to separate classes in CircuitPython 9. See https://learn.adafruit.com/circuitpython-display-support-using-displayio/faqs#faq-3167831 for details.

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:

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

I2CDisplay

The I2CDisplay class is used to talk to displays over an `i2c_bus`. You specify the display `device_address` like you typically would for an I2C device. An optional `reset` pin can also be specified.

To setup an I2CDisplay bus, you would first create an `i2c_bus` object and then pass that in along with the `device_address`. Here's the basic usage example:

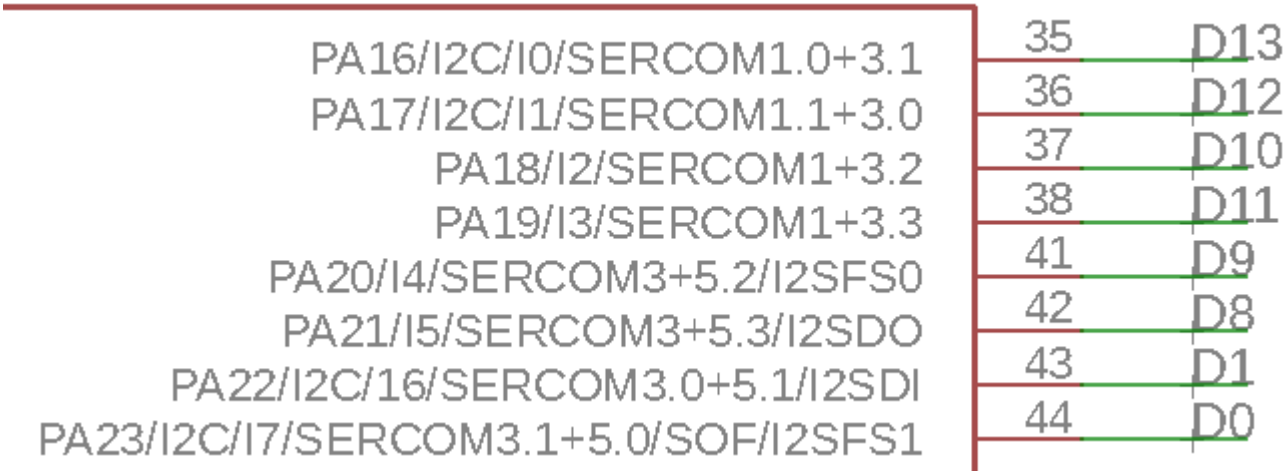
```
display_bus = displayio.I2CDisplay(
    board.I2C(),
    device_address=0x3D,
)
```

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:



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.

```
display_bus = paralleldisplaybus.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:

```
display = dispalyio.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/19el) - color TFT

Then you would create your display like this:

```
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 (http://adafru.it/3900), PyPortal (http://adafru.it/4116), CLUE (http://adafru.it/4500), etc. 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:

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

Boards without Built In Displays

If you have a more generic main board, like a Feather (https://adafru.it/Dij) or Matrix Portal (https://adafru.it/NDR), then you will need to create the display manually as described above. This means there will **not** be a **board.DISPLAY** available in the Circuit Python firmware.

- For OLED and TFT FeatherWings or breakouts, see examples in their associated library.
- For RGB matrices with the Matrix Portal, checkout the MatrixPortal Library (https://adafru.it/OCK).

Using a Display

Once a **Display** is setup, use the `root_group` property 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:

```
display.root_group = group
```

For CircuitPython versions prior to 8.0, use display.show(group) instead.

Keep in mind that while a **Display** can only show one **Group** (the so called **root group**), multiple **Groups** can be nested within the **root group** for more complex layouts.

Releasing Displays

Once you've created your display instance, the CircuitPython firmware will remember the setup between soft resets. This helps facilitate showing the serial output on the display, which can be useful for seeing error messages, etc.. Because of this behavior, you may run into an issue similar to what is shown below:

```
Adafruit CircuitPython 5.3.1 on 2020-07-13; Adafruit ItsyBitsy M4 Expre
amd51g19
>>>
soft reboot

Auto-reload is on. Simply save files over USB to run them or enter REPL
le.
code.py output:
Traceback (most recent call last):
  File "code.py", line 4, in <module>
RuntimeError: Too many display busses
```

To avoid this issue, you can use the `release_displays()` command in displayio (https://adafru.it/MAr). **Call this before creating your display bus.** You can tuck this call in somewhere up near the top of your code. For example:

```
import board
import i2cdisplaybus
import adafruit_ssd1327

# release any currently configured displays
displayio.release_displays()

# go through display setup as normal
display_bus = displayio.I2CDisplay(board.I2C(), device_address=0x3D)
display = adafruit_ssd1327.SSD1327(display_bus, width=128, height=128)

#
# use display as you wish
#
```

Now the code will run without errors with each soft reset.

You do not need to do this for boards with built in displays. For those cases, the release gets taken care of for you as part of the internal display creation which provides the `board.DISPLAY` instance.

EPaperDisplay

The **EPaperDisplay** class is similar to the **Display** class discussed previously, but is specific to electronic paper display (aka EPD, eInk, epaper, etc.) hardware.

Also much like **Display**, you rarely, if ever, will use the **EPaperDisplay** class directly. Instead, you will use a library which takes care of display specific setup for the many available EPD breakouts and boards. Some examples include:

- SSD1608 (https://adafru.it/ZaU)
- SSD1675 (https://adafru.it/ZaV)
- IL91874 (https://adafru.it/ZaW)
- IL0398 (https://adafru.it/ZaX)
- IL0373 (https://adafru.it/ZaY)

If you have one of those EPDs, you can just use the corresponding library. The code there can be useful as examples for other EPDs.

Boards with Built in EPDs

If you are using a board with a built in EPD, like the Adafruit MagTag (http://adafru.it/4800), then an **EPaperDisplay** will already be setup for you in the CircuitPython firmware. You can access it simply with:

```
import board
epd = board.DISPLAY
```

EPD Usage

The general usage of **EPaperDisplay** is much like regular **Display**. Use `display.root_group` to establish what **Group** will be shown. There are `width` and `height` properties available to query display size. There is also a `rotation` property that can be used to query / set rotation.

To actually display the **Group**, you call `refresh()`. However, there are EPD specific things which must be taken into account. We discuss those next.

EPD Specific Behavior

EPDs are fundamentally different hardware than other displays like TFTs. The main difference is that they are slow to display and are limited in how often they can be refreshed. Therefore **EPDs do not auto refresh**.

EPaperDisplays do not auto refresh. You must call refresh() unless a library does that for you.

To refresh the display, you simply call `refresh()`. However, to take care of the slowness and refresh limit, these additional properties are important:

- `time_to_refresh` - This is the time, in **seconds**, until you can refresh the display. If you call `refresh()` too soon, you will throw an exception.
- `busy` - This is `True` while the display is in the process of refreshing. Use this if you want to make sure the display refresh is complete before doing something else.

Here is a simple example of how these properties might get used:

```
# wait until we can actually refresh
time.sleep(epd.time_to_refresh)
# refresh the display
epd.refresh()
# (optional) wait until display is fully updated
while epd.busy:
    pass
# display is now updated
```

The exact usage would depend on your specific application. For example, you are free to move on to other things without querying `busy`. However, if you did something like immediately enter deep sleep after calling `refresh()`, you would want make sure the display is fully updated before doing so.

Also realize that you most likely can not call `refresh()` again immediately after `busy` is complete. You still need to use `time_to_refresh` appropriately. For example, the display may update in 5 seconds (`busy` becomes False) but can only be refreshed once every 60 seconds (`time_to_refresh` is > 0).

Examples

CircuitPython Firmware
https://adafru.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.

A Note on Display Setup

Most of these examples assume a board with a built in display. See the Display and Display Bus (https://adafru.it/Uam) section for information on built in vs. external displays. If you are not using a board with a built in display, then this line:

```
display = board.DISPLAY
```

in the examples will not work. That would need to be substituted for lines that configure an external display.

Text

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, each which takes care of a certain aspect.

Display Text

The CircuitPython Display Text Library (https://adafru.it/FiA) is used to create text elements you can then display. We cover the basics here, but checkout this guide for more info:

CircuitPython Display_Text Library
https://adafru.it/Rof

Bitmap Fonts

The CircuitPython Bitmap Font Library (https://adafru.it/DZl) provides support for using custom fonts. We show a simple example here, but checkout this guide for more info:

Custom Fonts for CircuitPython Displays
https://adafru.it/EFI

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 straight forward - 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:

Example assumes board with a built in display.

```
# SPDX-FileCopyrightText: 2019 Carter Nelson for Adafruit Industries
#
# SPDX-License-Identifier: MIT
```

```
import board
import terminalio
from adafruit_display_text import label

display = board.DISPLAY

# Set text, font, and color
text = "HELLO WORLD"
font = terminalio.FONT
color = 0x0000FF

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

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

# Show it
display.root_group = text_area

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

Using Bitmap Fonts

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

Here we show a basic BDF example. First, copy the font file to your CIRCUITPY folder somewhere. You then load it using `load_font()` as shown below. The rest is the same as the example above.

```
# SPDX-FileCopyrightText: 2019 Carter Nelson for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import board
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 tet label
text_area = label.Label(font, text=text, color=color)

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

# Show it
display.root_group = text_area

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

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

Helvetica-Bold-16.bdf
https://adafru.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 Positioning

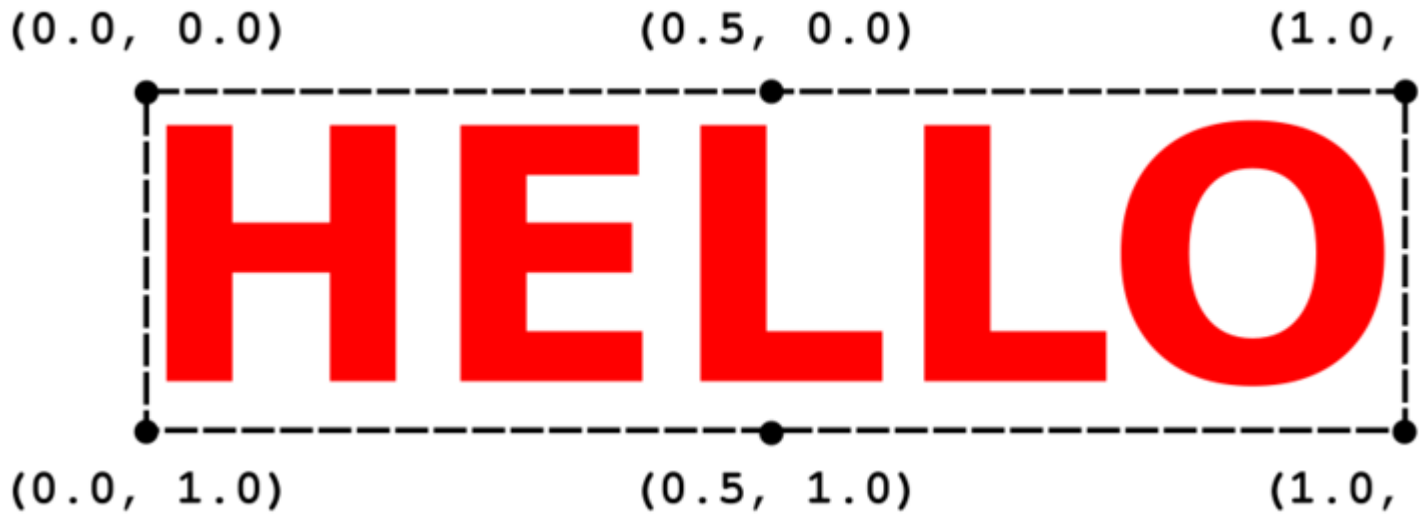
When setting text location using the `x` and `y` properties, you are setting the origin point. It is located relative to the text as shown below.

Alternatively, thanks to more recent updates (https://adafru.it/IUe) to the CircuitPython Display Text (https://adafru.it/FiA) library, you can now change the anchor point used to locate the text label. You do so by using the `anchor_point` property of the `label` and giving it an `(x, y)` tuple, like this:

```
label.anchor_point = (0.1, 0.8)
```

The values range from 0 to 1 with `x` being the horizontal and `y` being the vertical. The origin is in the upper left corner. A value of 0 is at the origin. A value of 1 is all the way to the right/down.

Here are some example locations:



You can then set the position of the label on the display using the `anchored_position` property and also specifying an `(x, y)` tuple. But this time, the values are actual screen coordinates in pixels, like this:

```
label.anchored_position = (120, 85)
```

See the example program linked below for lots of common example use cases:

[display_text_anchored_position.py](https://adafru.it/PEa)
https://adafru.it/PEa

Changing Text

If you ever want to change the text of a label, you can do this:

```
text_area.text = "NEW TEXT"
```

However, the new text length can not be longer than what was originally specified when the label was created. So you have to know the expected max number of characters ahead of time. One easy way to do this when creating the label is with something like:

```
text_area = label.Label(font, text=" "*20)
```

where `20` is the character count, i.e. the maximum number of characters the label can display.

Display a Bitmap

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.

BMP File Format

Not all BMP file formats are supported. You will need to make sure you have an **indexed BMP** file. Follow the link below for some good info on how to convert or create such a BMP file:

Indexed BMP Graphics
https://adafru.it/MbZ

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:

purple.bmp
https://adafru.it/EFK

Here's the code:

Example assumes board with a built in display.

```
# SPDX-FileCopyrightText: 2019 Carter Nelson for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import board
import displayio

display = board.DISPLAY

# Setup the file as the bitmap data source
bitmap = displayio.OnDiskBitmap("/purple.bmp")

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

# 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.root_group = group

# Loop forever so you can enjoy your image
```

```
while True:
    pass
```

ImageLoad

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

```
# SPDX-FileCopyrightText: 2019 Carter Nelson for Adafruit Industries
#
# SPDX-License-Identifier: MIT

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.root_group = group

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

Draw Pixels

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:

Example assumes board with a built in display.

```
# SPDX-FileCopyrightText: 2019 Carter Nelson for Adafruit Industries
#
# SPDX-License-Identifier: MIT

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.root_group = 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

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

Sprite Sheet

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

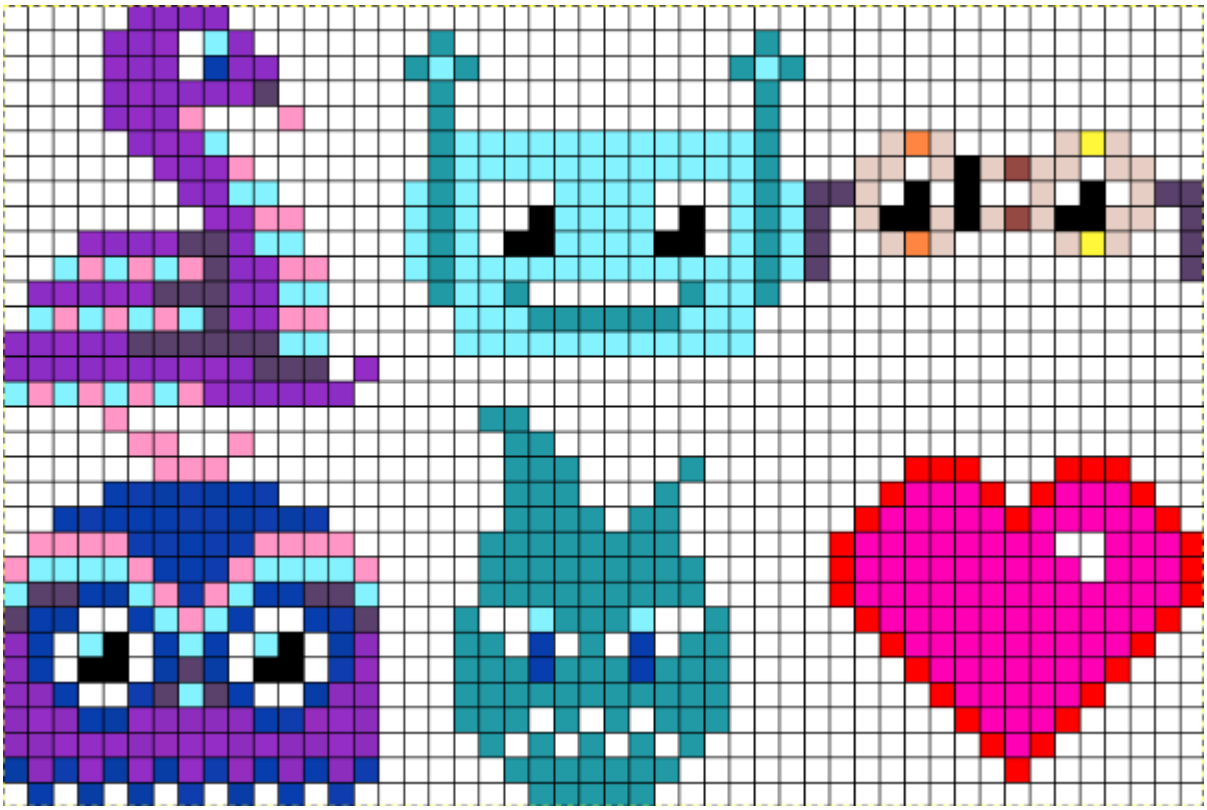
that contains several sprites all arranged nicely. This is called a "sprite sheet".

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

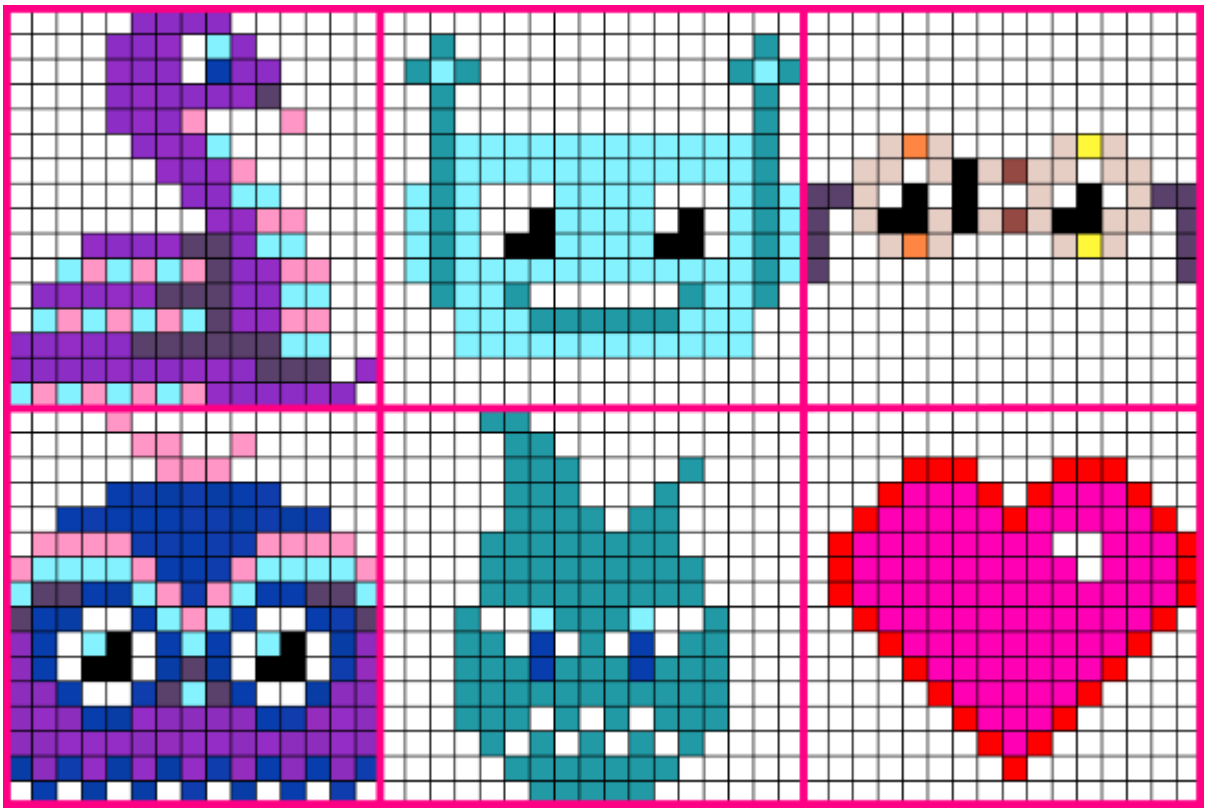
cp_sprite_sheet.bmp
https://adafru.it/EFM

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:

```
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:

```
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:

```
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.

Example assumes board with a built in display.

```
# SPDX-FileCopyrightText: 2019 Carter Nelson for Adafruit Industries
#
# SPDX-License-Identifier: MIT

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.root_group = 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:

```
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
```

Multiple TileGrids

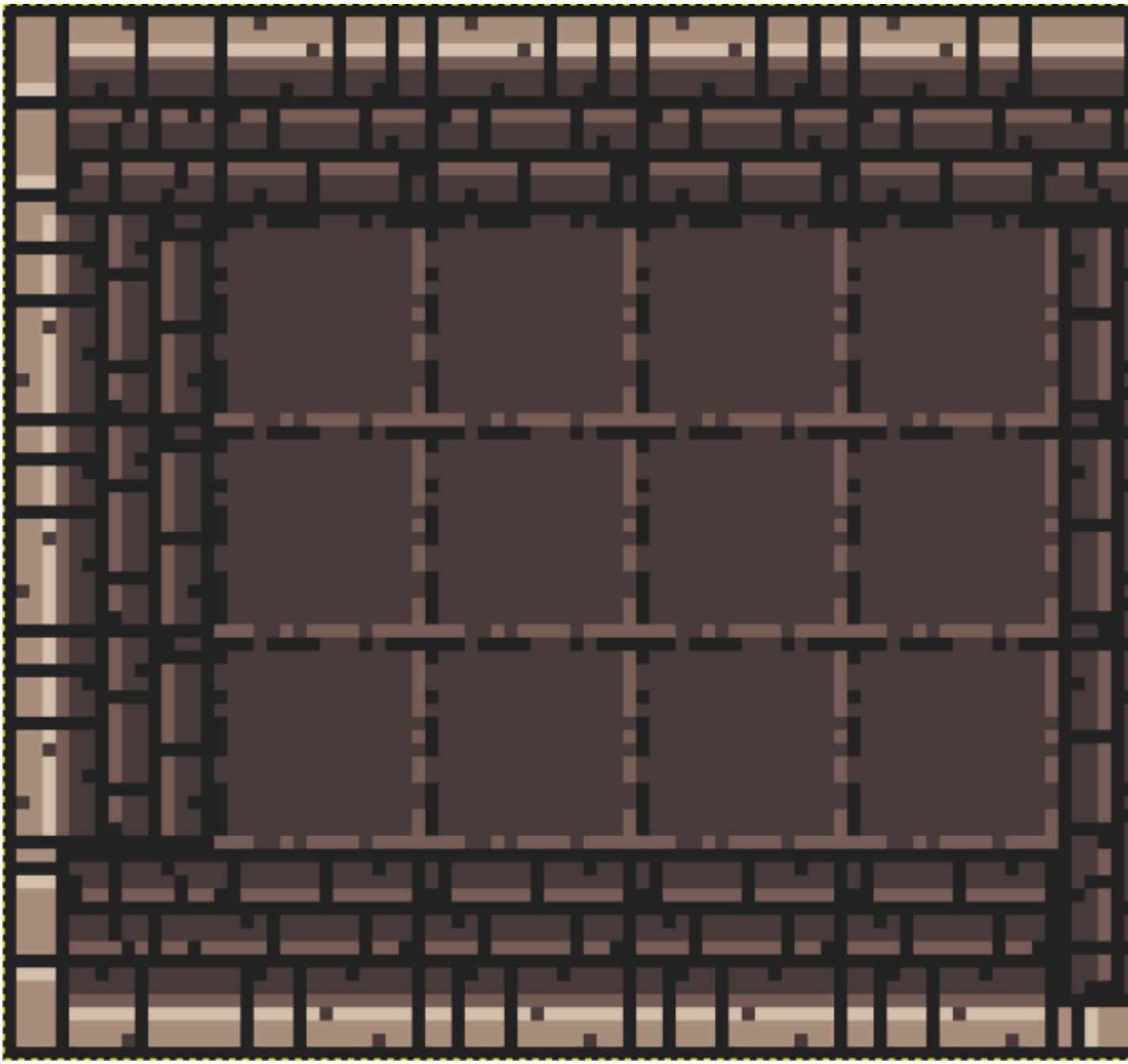
This example builds on the Sprite Sheet example to show a more sophisticated usage of **TileGrid**. We'll show how you can have more than one **TileGrid** and that a **TileGrid** can be more than just one tile.

The castle wall tiles used in this example were borrowed from this excellent tilesheet: dungeontileset-ii (https://adafru.it/EFN)

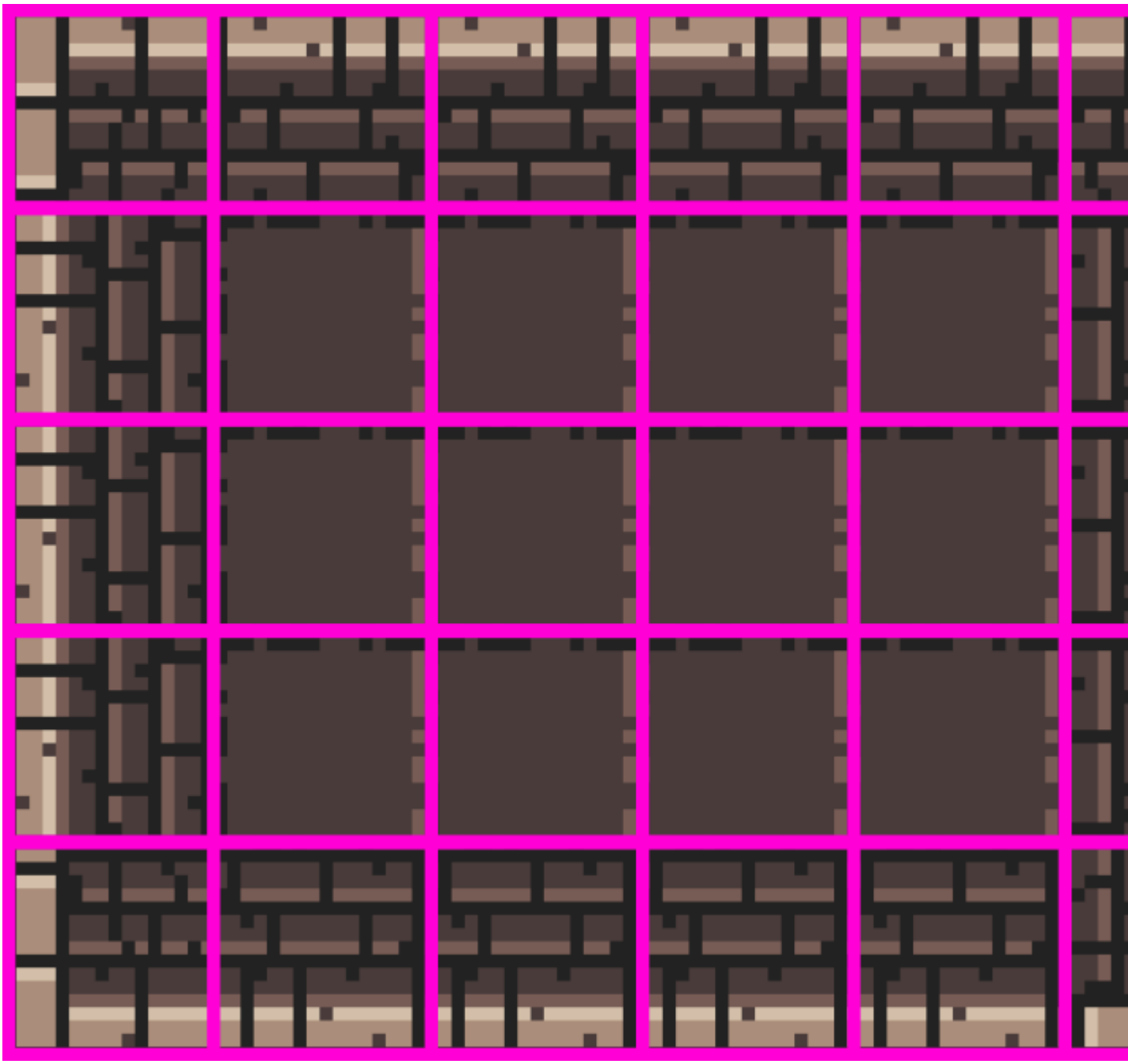
A Sprite and Its Castle

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:



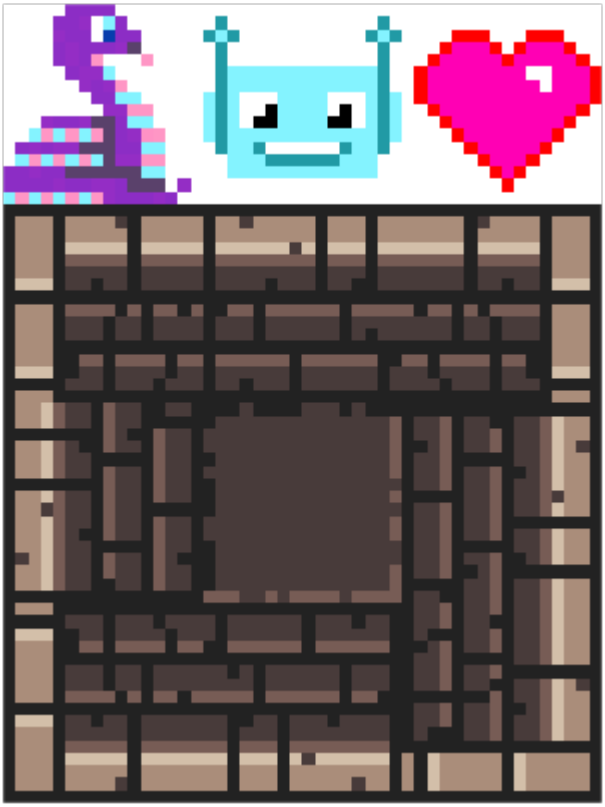
You can kind of already see the grid like repetitive pattern. Let's put a reference grid over the top:



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:

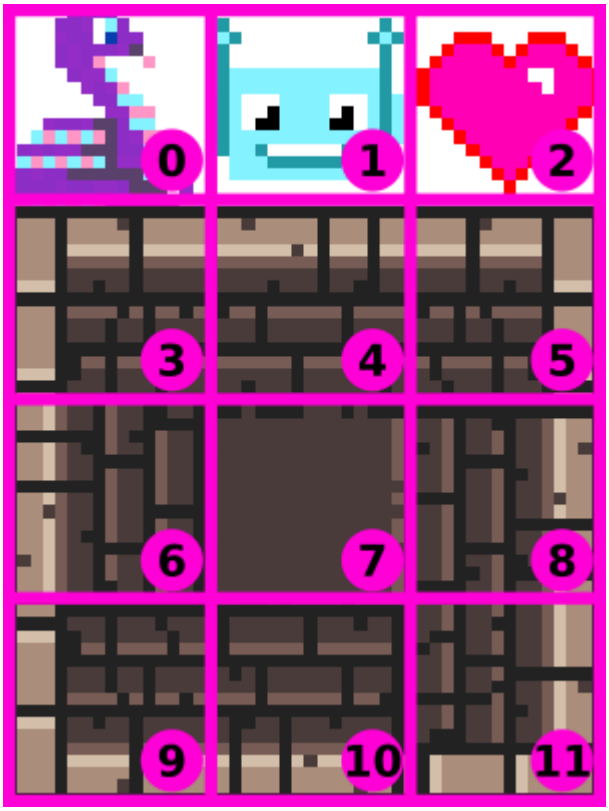
castle_sprite_sheet.bmp
https://adafru.it/EFO

Another super tiny BMP! Here's what it looks like more blown up:



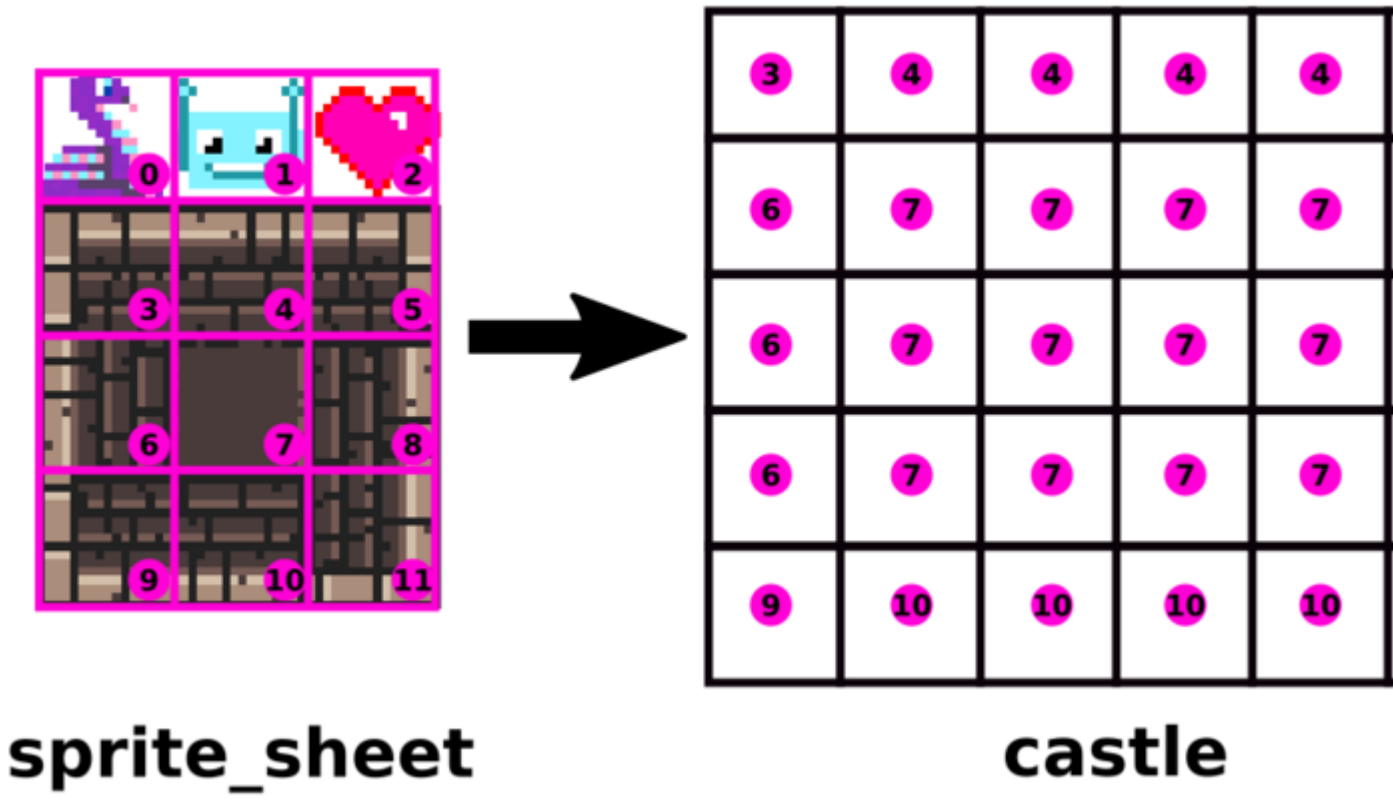
There are a couple of characters we can use for our sprite at the top. But there's also the basic building blocks needed for our castle.

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



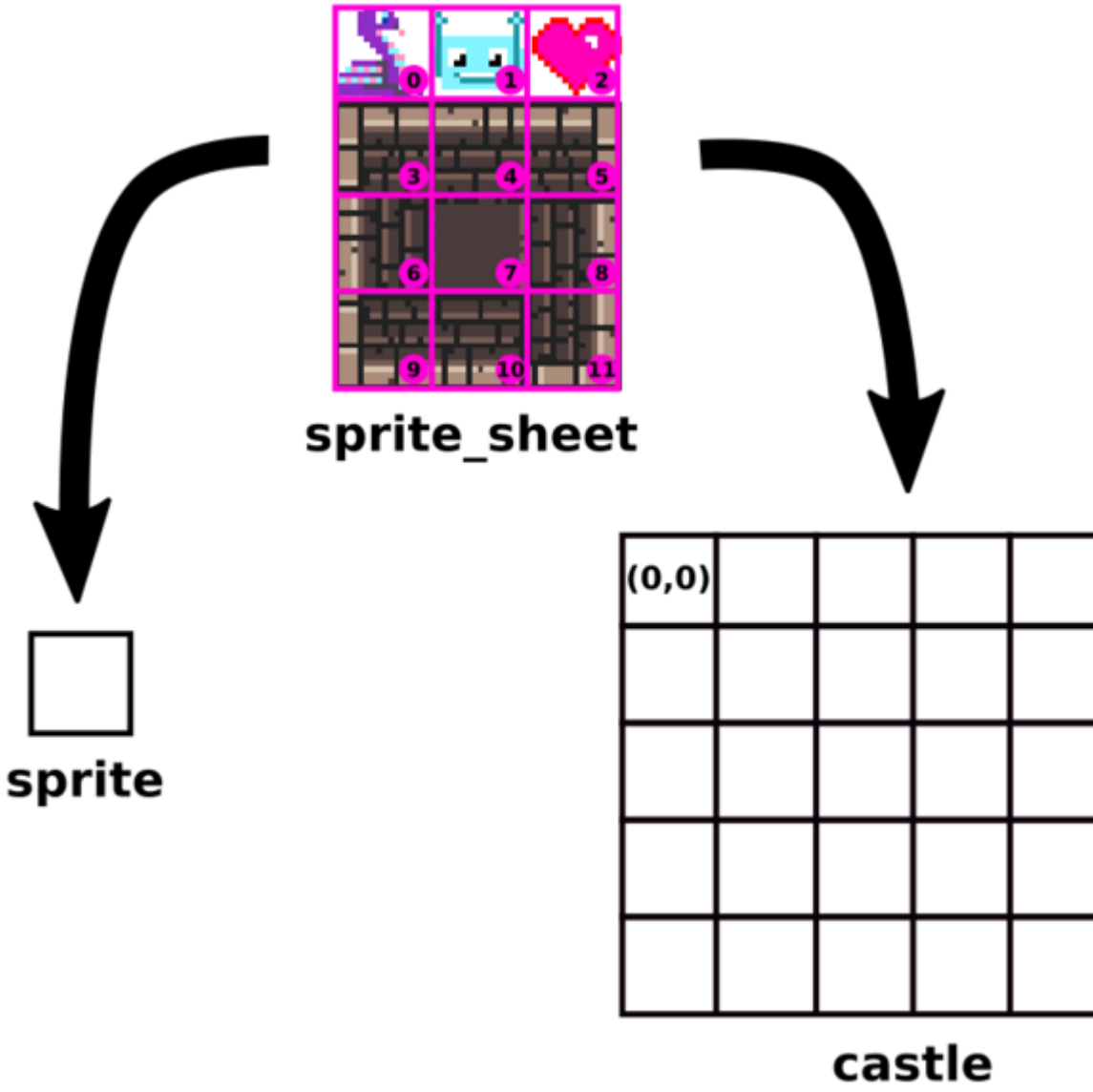
As mentioned above, our castle is 6 tiles wide by 5 tiles high. So that will be the size of the **TileGrid** we'll create to generate the castle. Then, each tile in the castle **TileGrid** just needs to be set to the correct index from the source bitmap.

Here's what that would look like:



But keep in mind this is only one of the **TileGrid**s 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:

Example assumes board with a built in display.

```
# SPDX-FileCopyrightText: 2019 Carter Nelson for Adafruit Industries
#
```

```
# SPDX-License-Identifier: MIT

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)

# Create a Group to hold the sprite and castle
group = displayio.Group()

# Add the sprite and castle to the group
group.append(castle_group)
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
```

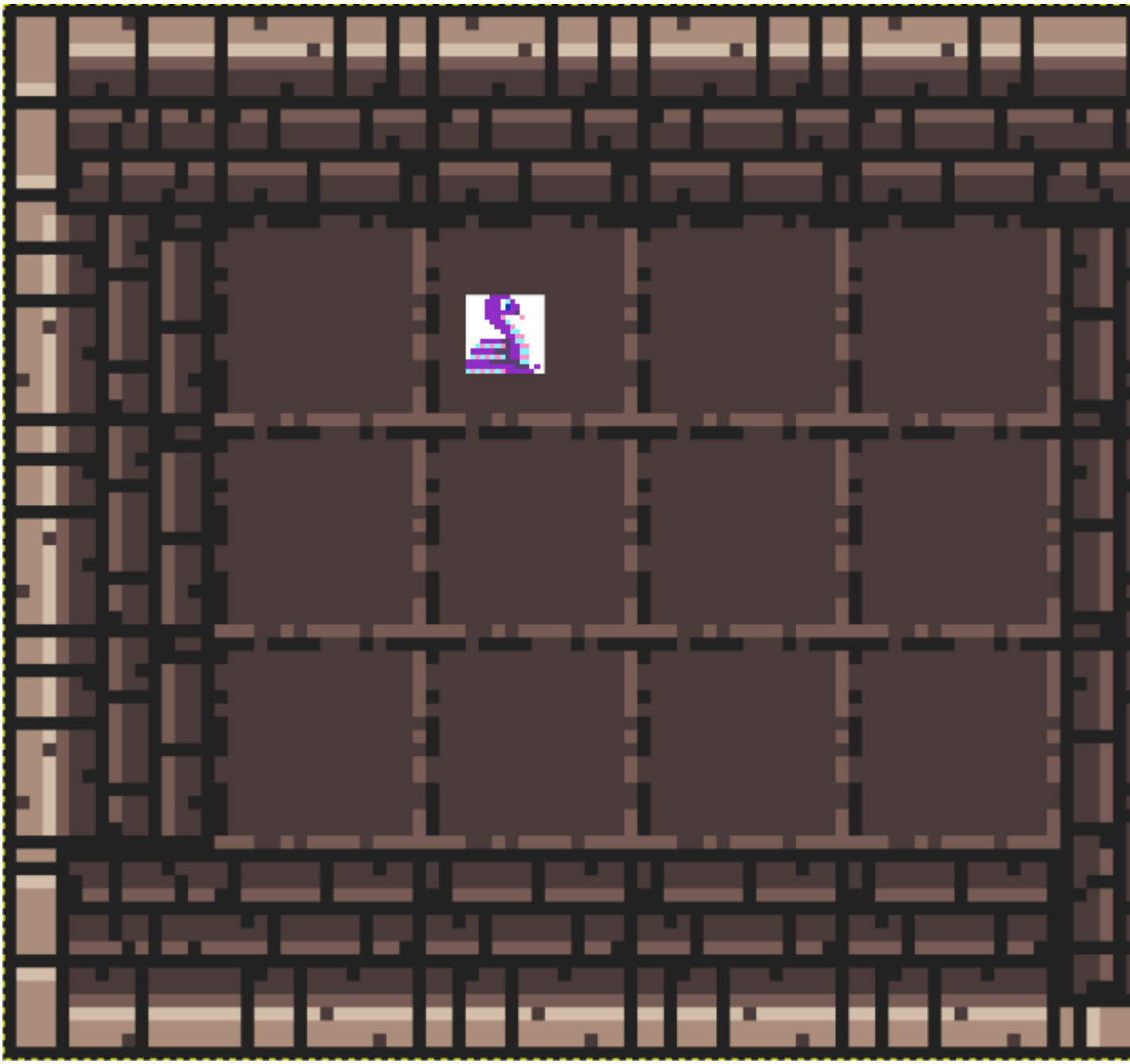
```
    castle[x, 4] = 10 # bottom
# left/ right walls
for y in range(1, 4):
    castle[0, y] = 6 # left
    castle[5, y] = 8 # right
# floor
for x in range(1, 5):
    for y in range(1, 4):
        castle[x, y] = 7 # floor

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

# Add the Group to the Display
display.root_group = group

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

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



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.

```
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:

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

Change Sprite Location

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

```
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!

External Display

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:

2.4" TFT SPI Wiring
https://adafru.it/EFP

```
# SPDX-FileCopyrightText: 2019 Carter Nelson for Adafruit Industries
#
# SPDX-License-Identifier: MIT

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_

# 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"\xc7\x01\x86"            # VCM control2
    b"\x36\x01\x38"            # Memory Access Control
    b"\x37\x01\x00"            # Vertical scroll zero
    b"\x3a\x01\x55"            # COLMOD: Pixel Format Set
    b"\xb1\x02\x00\x18"        # Frame Rate Control (In Normal Mode/Full C
    b"\xb6\x03\x08\x82\x27"    # Display Function Control
    b"\xF2\x01\x00"            # 3Gamma Function Disable
    b"\x26\x01\x01"            # Gamma curve selected
    b"\xe0\x0f\x0F\x31\x2B\x0C\x0E\x08\x4E\xF1\x37\x07\x10\x03\x0E\x09\x00
    b"\xe1\x0f\x00\x0E\x14\x03\x11\x07\x31\xC1\x48\x08\x0F\x0C\x31\x36\x0F
    b"\x11\x80\x78"            # Exit Sleep then delay 0x78 (120ms)
    b"\x29\x80\x78"            # Display on then delay 0x78 (120ms)
```

```
)

# Setup the Display
display = displayio.Display(display_bus, INIT_SEQUENCE, width=320, height=

#
# 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.root_group = group

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

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).

```
# SPDX-FileCopyrightText: 2019 Carter Nelson for Adafruit Industries
#
# SPDX-License-Identifier: MIT

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_

# 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.root_group = group

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

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.

Manual Refresh

Except for EPD (eink) displays, **displayio** by default automatically takes care of refreshing the display. This means you never need to call `refresh()` and things happen automagically. However, there are times when you may want to turn this feature off and instead manually refresh the display. One example might be if you are updating a lot graphical items and want to have the changes appear "all at once" on the display. Or maybe you want to be very exact about syncing the update with some other event.

Manually refreshing is pretty simple. The basic idea is to turn off the auto refresh behavior and then call `refresh()` as needed.

Turning Auto Refresh Off

There is an `auto_refresh` parameter you can use when creating your **Display** object. If you set this to `False`, the display will be created with auto refresh turned off. For example:

```
display = ST7789(display_bus, width=240, height=240, rowstart=80, auto_ref
```

The other way is to use the `auto_refresh` property after you've created the display. This not only returns the current state, but is also used to set the state. To check current state (if you wanted to), you could do something like:

```
if display.auto_refresh:
    print("Auto refresh is ON.")
else:
    print("Auto refresh is OFF.")
```

To turn auto refresh off, you simply set it `False`:

```
display.auto_refresh = False
```

Example Usage

Once you disable auto refresh, it is up to you to call `refresh()` as needed to show any updates. Exactly when and where you would do this is specific to your use case. But here's a simple boiler plate example:

```
# turn off auto refresh
display.auto_refresh = False

#
# your code here that changes the display
#

# when ready to show results, call refresh()
display.refresh()
```

Turning Auto Refresh On

If you ever want to revert back to the default behavior, simply re-enable auto refresh.

```
display.auto_refresh = True
```

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.

Referencing the Display

On boards that have a display, it is automatically initialized. to reference it, you simply need import `board` and assign it to the built-in display reference:

```
import board

display = board.DISPLAY
```

On board without a built-in display, we recommend taking a look at the CircuitPython pages regarding your specific display for how to initialize it. You would then assign display to the returned output of the display driver initialization. For instance, if you have an ILI9341 display, it would look something like this, though your pins may vary.

```
import board
import displayio
import adafruit_ili9341

displayio.release_displays()

spi = board.SPI()
tft_cs = board.D9
tft_dc = board.D10

display_bus = displayio.FourWire(
    spi, command=tft_dc, chip_select=tft_cs, reset=board.D6
)
display = adafruit_ili9341.ILI9341(display_bus, width=320, height=240)
```

Groups

Groups (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 - called the **root 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.

```
import displayio
```

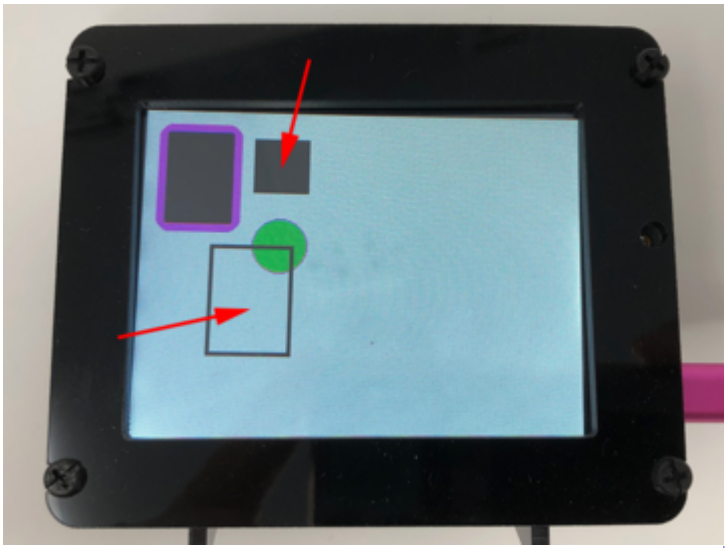
You must also create the group that will be used as your root group and set that:

```
my_display_group = displayio.Group()
display.root_group = my_display_group
```

For CircuitPython versions prior to 8.0, use display.show(my_display_group) instead.

Shapes

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

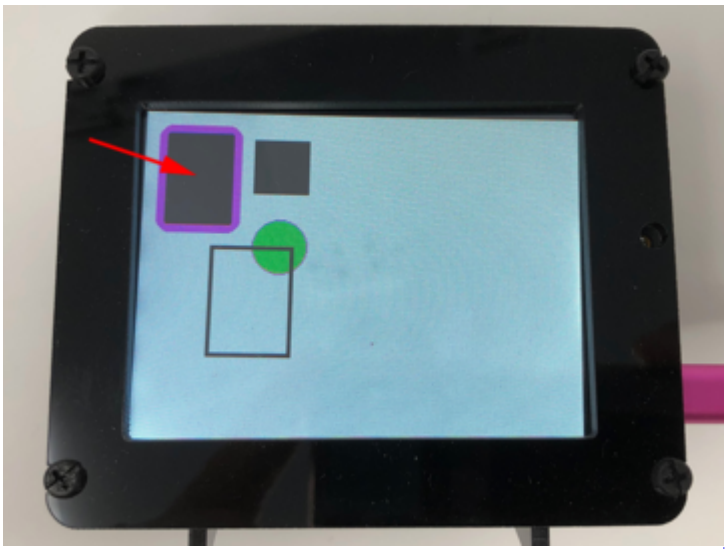
- 

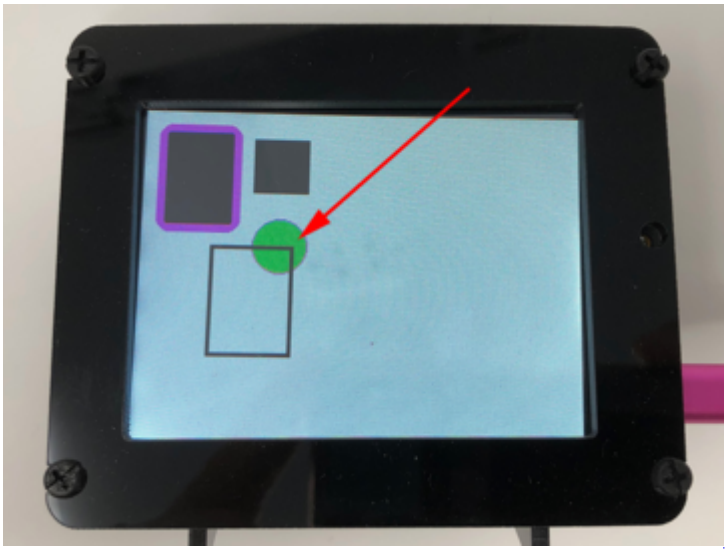
Rectangle

The rectangle is your most basic shape. It can either be filled, outlined, or both.

Rounded Rectangle

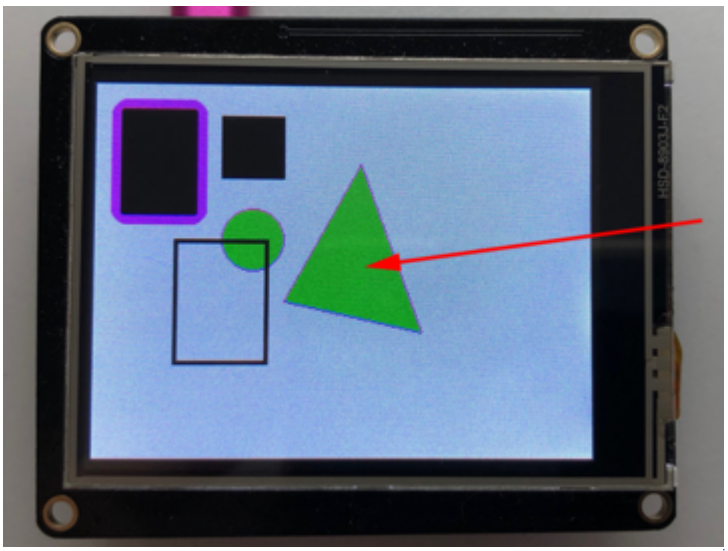
The rounded rectangle is a little more complex and is comprised of 4 lines and quarter circle corners.

- 

- 

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.

```
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://adafru.it/Fiu) with something like:

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

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

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

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

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

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

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

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

```
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://adafru.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.

```
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/Fiy) 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://adafru.it/FiB) makes use of the **adafruit_button** (https://adafru.it/FiC) library and builds on top of the **adafruit_display_shapes**, **adafruit_label**, and **adafruit_touchscreen** (https://adafru.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.

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

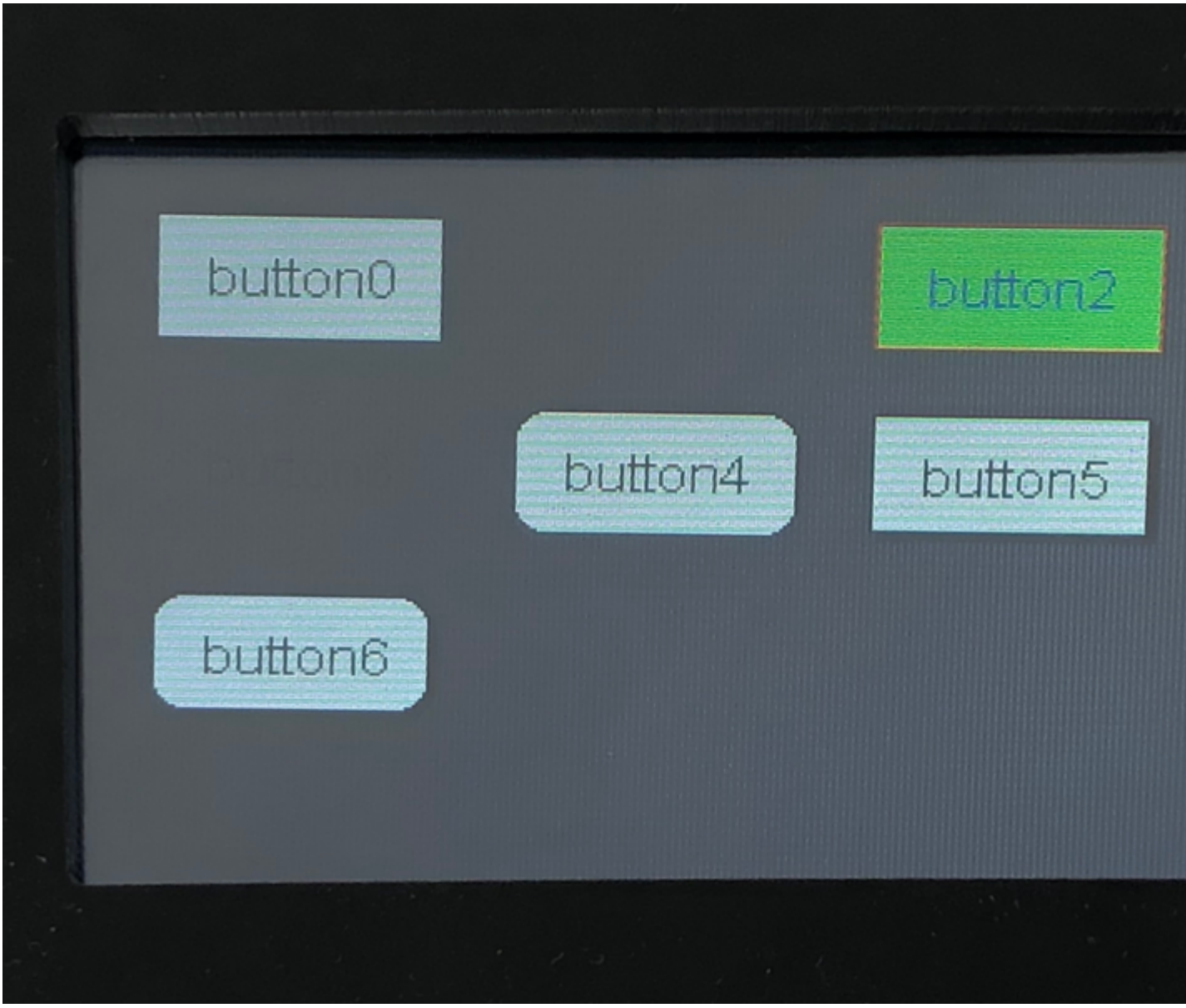
Next create your button. There are lots of options and you can take a look at some of the examples provided in the button library to get an idea of the various things you can do.

```
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.

```
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** - This library loads images from storage directly into memory. While faster, enough memory to hold the image data must be available.
- **OnDiskBitmap** - This uses the image data directly from storage (disk). While slower, this approach requires less memory.

ImageLoad Library

Imageload (https://adafru.it/FiE) is the main class in the **adafruit_imageload** (https://adafru.it/EFL) library provides an easy way to decode and display bitmaps. To use it, first include the library at the top of your file.

```
import adafruit_imageload
```

Second, Generate the Bitmap and Palette from the image:

```
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:

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

Finally add the TileGrid to your display group.

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

OnDiskBitmap

OnDiskBitmap (https://adafru.it/EFt) 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.

```
my_bitmap = displayio.OnDiskBitmap("/my_bitmap.bmp")
```

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

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

Finally add the TileGrid to your display group.

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

Calculator UI Elements

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

Helper Libraries

There are numerous helper libraries that have been created to use along with **displayio**. Some of these have already been mentioned, but here we provide a summary list.

The List

In no particular order.

- Adafruit_CircuitPython_Display_Text (https://adafru.it/FiA) - text labels

- Adafruit_CircuitPython_Bitmap_Font (https://adafru.it/Fiv) - custom font support
- Adafruit_CircuitPython_ImageLoad (https://adafru.it/EFL) - load image files
- Adafruit_CircuitPython_Display_Shapes (https://adafru.it/Jaq) - lines and circles and triangles, oh my!
- Adafruit_CircuitPython_Display_Button (https://adafru.it/FiC) - clickable buttons
- Adafruit_CircuitPython_ProgressBar (https://adafru.it/Kpb) - progress bars
- Adafruit_CircuitPython_Display_Notification (https://adafru.it/QGD) - notifications
- Adafruit_CircuitPython_DisplayIO_Layout (https://adafru.it/QGE) - graphical element layout assistant
- Adafruit_CircuitPython_Slideshow (https://adafru.it/QGF) - multi-image slideshow driver

FAQs

Is there an easy way to draw bar graphs?

Yes! We have the Adafruit CircuitPython ProgressBar (https://adafru.it/Kpb) library available. Examples can be found here (https://adafru.it/Ody).

How did displayio naming change in CircuitPython 9?

The names for various components of `displayio` were changed in CircuitPython 9. The different kinds of displays were split into separate top-level modules. Using the old names in CircuitPython 9 will generate warnings, but will still work. In CircuitPython 10 the old names will be removed completely.

- `displayio.Display` is now `busdisplay.BusDisplay`.
- `displayio.FourWire` is now `fourwire.FourWire`.
- `displayio.EPaperDisplay` is now `epaperdisplay.EPaperDisplay`.
- `displayio.I2CDisplay` is now `i2cdisplaybus.I2CDisplayBus`.

Why doesn't `display.show()` work anymore?

In CircuitPython 8, `display.show(some_group)` was replaced by `display.root_group = some_group`. You can also read the current root

group with `display.root_group`. `show()` no longer works in CircuitPython 9.