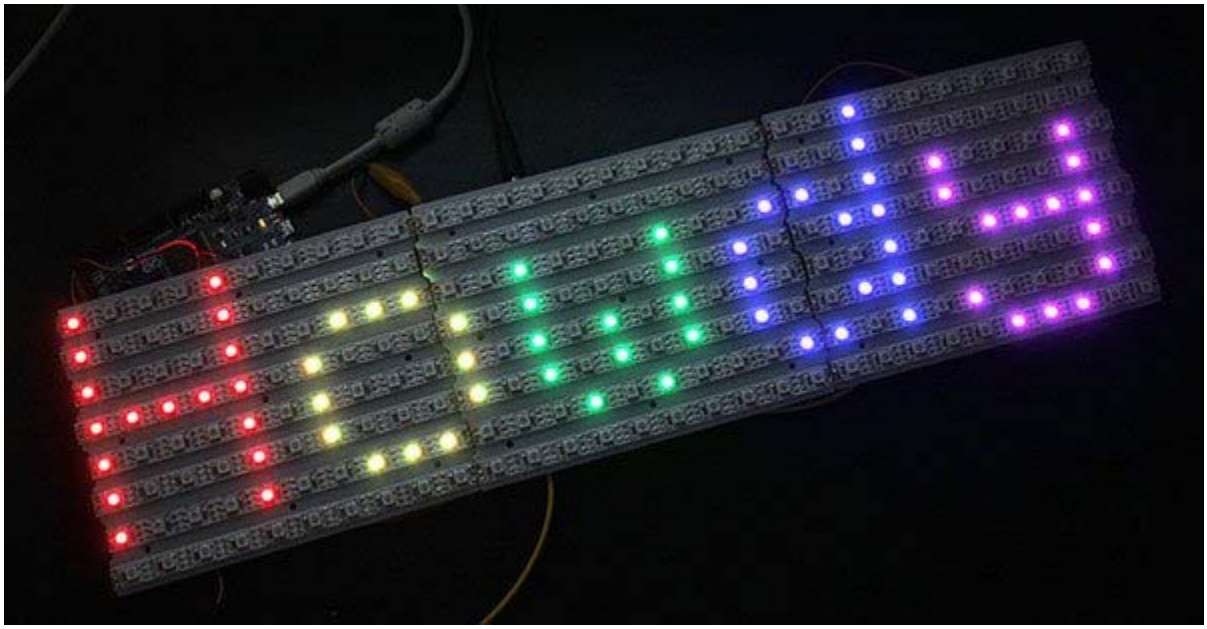
NeoMatrix Library

by Phillip Burgess



The Adafruit_NeoMatrix library builds upon Adafruit_NeoPixel to create two-dimensional graphic displays using NeoPixels. You can then easily draw shapes, text and animation without having to calculate every X/Y pixel position. Small NeoPixel matrices are available in the shop. Larger displays can be formed using sections of NeoPixel strip, as shown in the photo above.

In addition to the Adafruit_NeoPixel library (which was already downloaded and installed in a prior step), NeoMatrix requires two additional libraries:

1. Adafruit_NeoMatrix
2. Adafruit_GFX

If you've previously used any Adafruit LCD or OLED displays, you might already have the latter library installed.

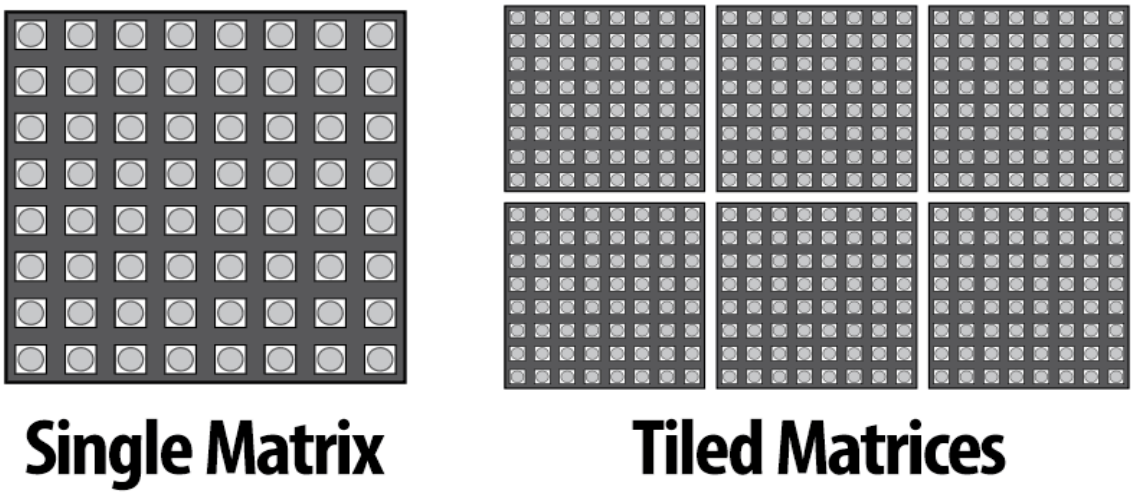
Installation for both is similar to Adafruit_NeoPixel before: unzip, make sure the folder name matches the .cpp and .h files within, then move to your Arduino libraries folder and restart the IDE.
Arduino sketches need to include all three headers just to use this library:

```
#include <Adafruit_GFX.h>
#include <Adafruit_NeoMatrix.h>
#include <Adafruit_NeoPixel.h>
```

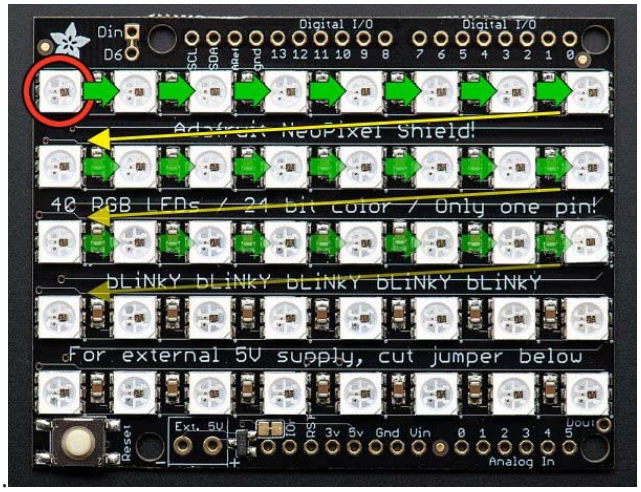
Layouts

Adafruit_NeoMatrix uses exactly the same coordinate system, color functions and graphics commands as the Adafruit_GFX library. If you're new to the latter, a separate tutorial explains its use. There are also example sketches included with the Adafruit_NeoMatrix library.

We'll just focus on the *constructor* here — how to declare a two-dimensional display made from NeoPixels. Powering the beast is another matter, covered on the prior page.
The library handles both *single* matrices — all NeoPixels in a single uniform grid — and *tiled* matrices — multiple grids combined into a larger display:

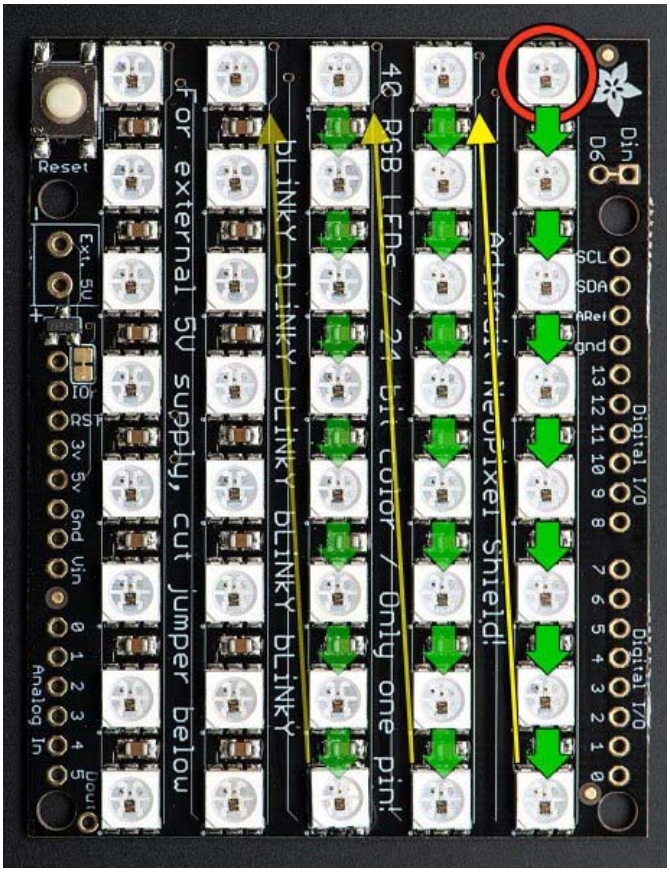


Let's begin with the declaration for a single matrix, because it's simpler to explain.
We'll be demonstrating the NeoPixel Shield for Arduino in this case — an 8x5 matrix of NeoPixels. When looking at this shield with the text in a readable orientation, the first pixel, #0, is at the top left. Each successive pixel is right one position — pixel 1 is directly to the right of pixel 0, and so forth. At the end of each row, the next pixel is at the left side of the next row. This isn't something we decide in code…it's how the NeoPixels are hard-wired in the circuit board comprising the shield



We refer to this layout as *row major* and *progressive. Row major* means the pixels are arranged in horizontal lines (the opposite, in vertical lines, is *column major*). *Progressive* means each row proceeds in the same direction. Some matrices will reverse direction on each row, as it can be easier to wire that way. We call that a *zigzag* layout.

However…for this example, we want to use the shield in the "tall" direction, so the Arduino is standing up on the desk with the USB cable at the top. When we turn the board this way, the matrix layout changes…



Now the first pixel is at the **top right**. Pixels increment top-to-bottom — it's now **column major**. The order of the columns is still **progressive** though.

We declare the matrix thusly:

```
Adafruit_NeoMatrix matrix = Adafruit_NeoMatrix(5, 8, 6,
  NEO_MATRIX_TOP     + NEO_MATRIX_RIGHT +
  NEO_MATRIX_COLUMNS + NEO_MATRIX_PROGRESSIVE,
  NEO_GRB            + NEO_KHZ800);
```

The first two arguments — 5 and 8 — are the width and height of the matrix, in pixels. The third argument — 6 — is the pin number to which the NeoPixels are connected. On the shield this is hard-wired to digital pin 6, but standalone matrices are free to use other pins.

The next argument is the interesting one. This indicates where the first pixel in the matrix is positioned and the arrangement of rows or columns. The first pixel must be at one of the four corners; *which* corner is indicated by adding either NEO_MATRIX_TOP or NEO_MATRIX_BOTTOM to
either NEO_MATRIX_LEFT or NEO_MATRIX_RIGHT. The row/column arrangement is indicated by further adding either NEO_MATRIX_COLUMNS or NEO_MATRIX_ROWS to
eitherNEO_MATRIX_PROGRESSIVE or NEO_MATRIX_ZIGZAG. These values are all added to form a single value as in the above code.

NEO_MATRIX_TOP + NEO_MATRIX_RIGHT + NEO_MATRIX_COLUMNS + NEO_MATRIX_PROGRESSIVE

The last argument is exactly the same as with the NeoPixel library, indicating the type of LED pixels being used. In the majority of cases with the latest NeoPixel products, you can simply leave this argument off…the example code is just being extra descriptive.

The point of this setup is that the rest of the sketch never needs to think about the layout of the matrix. Coordinate (0,0) for drawing graphics will always be at the top-left, regardless of the actual position of the first NeoPixel.

Why not just use the rotation feature in Adafruit_GFX?

Adafruit_GFX only handles rotation. Though it would handle our example above, it doesn't cover every permutation of rotation *and mirroring* that may occur with certain matrix layouts, not to mention the zig-zag capability, or this next bit…

Tiled Matrices

A *tiled* matrix is comprised of multiple smaller NeoPixel matrices. This is sometimes easier for assembly or for distributing power. All of the sub-matrices need to be the same size, and must be ordered in a predictable manner. The Adafruit_NeoMatrix() constructor then receives some additional arguments:

```
Adafruit_NeoMatrix matrix = Adafruit_NeoMatrix(
  matrixWidth, matrixHeight, tilesX, tilesY, pin, matrixType, ledType);
```

The first two arguments are the width and height, in pixels, of each tiled sub-matrix, not the entire display.

The next two arguments are the number of tiles, in the horizontal and vertical direction. The dimensions of the overall display then will always be a multiple of the sub-matrix dimensions.

The fifth argument is the pin number, same as before and as with the NeoPixel library. The last argument also follows prior behaviors, and in most cases can be left off.

The second-to-last argument though…this gets complicated…

With a single matrix, there was a starting corner, a major axis (rows or columns) and a line sequence (progressive or zigzag). This is now doubled — similar information is needed both for the pixel order within the individual

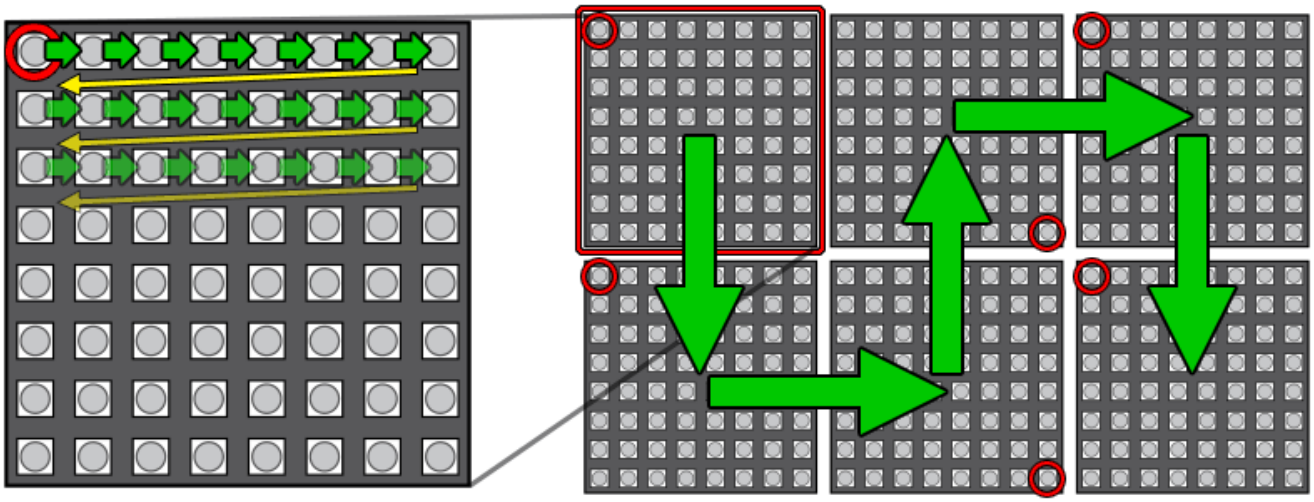
tiles, and the overall arrangement of tiles in the display. As before, we add up a list of symbols to produce a single argument describing the display format.

The NEO_MATRIX_* symbols work the same as in the prior single-matrix case, and now refer to the individual sub-matrices within the overall display. All tiles must follow the same format. An additional set of symbols work similarly to then describe the tile order.

The first tile must be located at one of the four corners. Add either NEO_TILE_TOP orNEO_TILE_BOTTOM and NEO_TILE_LEFT or NEO_TILE_RIGHT to indicate the position of the first tile. This is independent of the position of the first pixel within the tiles; they can be different corners.

Tiles can be arranged in horizontal rows or vertical columns. Again this is independent of the pixel order within the tiles. Add either NEO_TILE_ROWS or NEO_TILE_COLUMNS.

Finally, rows or columns of tiles may be arranged in progressive or zigzag order; that is, every row or column proceeds in the same order, or alternating rows/columns switch direction. Add eitherNEO_TILE_PROGRESSIVE or NEO_TILE_ZIGZAG to indicate the order. **BUT**…if NEO_TILE_ZIGZAGorder is selected, alternate lines of tiles must be rotated 180 degrees. This is intentional and by design; it keeps the tile-to-tile wiring more consistent and simple. This rotation is not required forNEO_TILE_PROGRESSIVE.



Tiles don’t need to be square! The above is just one possible layout. The display shown at the top of this page is three 10x8 tiles assembled from NeoPixel strip.

Once the matrix is defined, the remainder of the project is similar to Adafruit_NeoPixel. Remember to use matrix.begin() in the setup() function and matrix.show() to update the display after drawing. ThesetBrightness() function is also available. The library includes a couple of example sketches for reference.

Other Layouts

For any other cases that are not uniformly tiled, you can provide your own function to remap X/Y coordinates to NeoPixel strip indices. This function should accept two unsigned 16-bit arguments (pixel X, Y coordinates) and return an unsigned 16-bit value (corresponding strip index). The simplest row-major progressive function might resemble this:

```
uint16_t myRemapFn(uint16_t x, uint16_t y) {
  return WIDTH * y + x;
}
```

That’s a crude example. Yours might be designed for pixels arranged in a spiral (easy wiring), or a Hilbert curve.

The function is then enabled using setRemapFunction():

```
matrix.setRemapFunction(myRemapFn);
```

RAM Again

On a per-pixel basis, Adafruit_NeoMatrix is no more memory-hungry than Adafruit_NeoPixel, requiring 3 bytes of RAM per pixel. But the number of pixels in a two-dimensional display takes off exponentially…a 16x16 display requires *four times* the memory of an 8x8 display, or about 768 bytes of RAM (nearly half the available space on an Arduino Uno). It can be anywhere from *tricky* to*impossible* to combine large displays with memory-hungry libraries such as SD or ffft.

Gamma Correction

Because the Adafruit_GFX library was originally designed for LCDs (having limited color fidelity), it handles colors as 16-bit values (rather than the full 24 bits that NeoPixels are capable of). This is not the big loss it might seem. A quirk of human vision makes bright colors less discernible than dim ones. The Adafruit_NeoMatrix library uses *gamma correction* to select brightness levels that are visually (though not numerically) equidistant. There are 32 levels for red and blue, 64 levels for green.

The Color() function performs the necessary conversion; you don’t need to do any math. It accepts 8-bit red, green and blue values, and returns a gamma-corrected 16-bit color that can then be passed to other drawing functions.