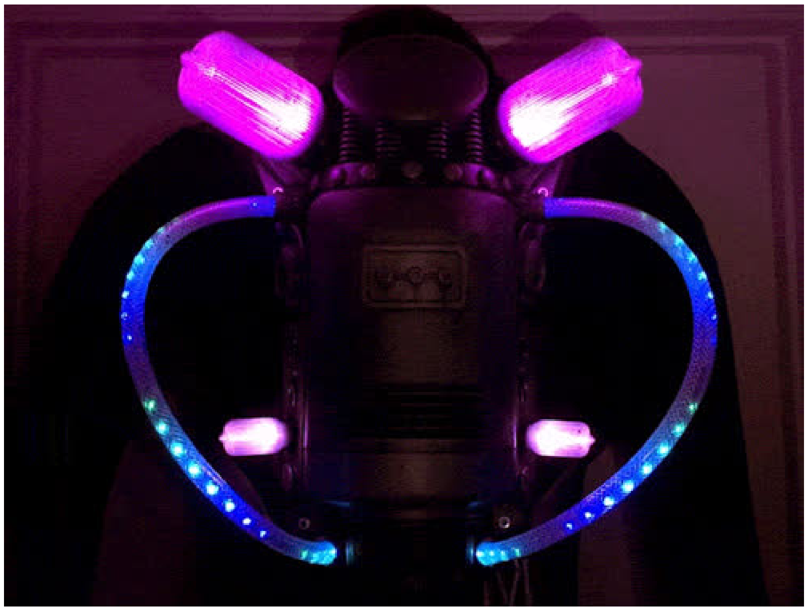
adafruit learning system

Sipping Power With NeoPixels

Created by Phillip Burgess



Last updated on 2018-08-22 04:00:09 PM UTC

Guide Contents

Overview

I'm often asked for help on a portable or wearable project, the designer insisting "it has to be *completely covered* in NeoPixels, and able to have them *all on white at 100% brightness.* But when I do the math it comes out to *1.21 gigawatts.* Help!"

"The math" in this case is a rule-of-thumb — perpetrated in the NeoPixel Überguide (https://adafru.it/iZe) and elsewhere — that each NeoPixel in a circuit may draw up to **60 milliamps** of current. It's a maximum worst case scenario, used in planning wire gauges and battery current and capacity. 100 NeoPixels? 60 × 100 = 6,000 milliamps. Six Amps. That's considerable.

While there are batteries that can deliver this sort of current (and more), they get progressively more heavy, costly and **dangerous**. For safety reasons I downplay the use of large lithium-polymer cells in wearable designs. In cosplay projects especially, something that might see only a few brief outings in a year, I'm still an advocate of single-use **alkaline batteries**. Easy to find, inexpensive and reasonably safe. Three AA cells in series provide 4.5 Volts (adequate for most 5V microcontrollers, and for NeoPixels as well) with a pretty decent 2,500 mAh capacity.

Even if you do decide on some crazy giant battery (at your own peril, hey sometimes I do it too), everything in this guide will be helpful in **improving run time**. Lower current usage also means a project can use narrower gauge wire… cheaper, lighter, and moves better in wearable projects.

The **key** to making this work is to approach the problem from the **other direction**: not how-much-battery, but how-*little*-NeoPixel can you get away with? Thoughtful design in layout, color selection and animation patterns make an *enormous* difference in power requirements.

"Music is the space between the notes." — Claude Debussy

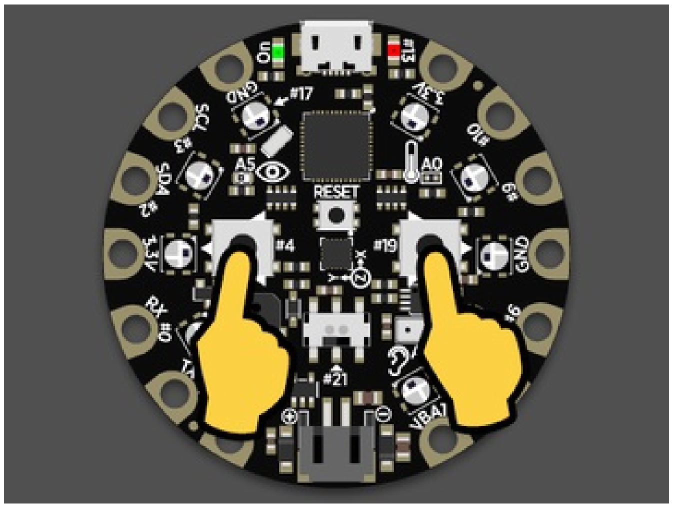
The example sketch later in this guide demonstrates these ideas using an Adafruit Circuit Playground board. If you don't have one, that's okay! The concepts explained here are applicable to *any* NeoPixel circuit.

Insights
Some Finer Points of NeoPixels

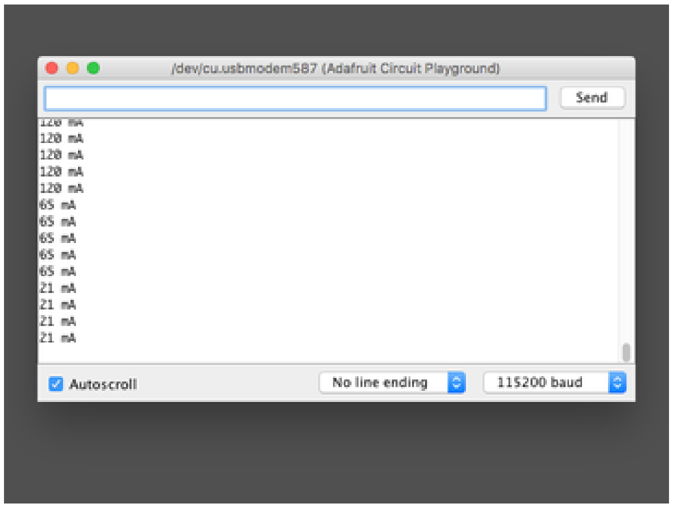
- The “60 milliamps” rule of thumb is just that…a rule of thumb, not hard science. *Actual* peak use is slightly less. But for napkin calculations and mental math, 60 is much easier than fifty-whatever. (The mini NeoPixels on Circuit Playground and “Mini Skinny” NeoPixel strips are more like 35 mA each, max. And RGBW NeoPixels, with a fourth LED element inside, may draw closer to 80 mA each.)
- Also, 60 (or 35, or 80) milliamps is a *worst case.* When animating and mixing colors, current draw will be less. Sometimes *much* less. **That’s what this guide is all about.**
- Even when set to color 0 (no LED elements lit), the driver logic inside each NeoPixel still uses a tiny bit of current. It’s under 1 milliamp per pixel…but with lots of pixels this could add up.

Strategies

Most of these tips are demonstrated in the Arduino sketch on the last page. That code is designed for Circuit Playground. If you don’t have one, that’s okay! The concepts explained here are applicable to *any* NeoPixel circuit.



If you *do* have a Circuit Playground board…upload the sketch and follow along! Use the **left and right buttons** to switch between different NeoPixel display modes. The **Serial Monitor** window shows a rough estimate of the required current in the active mode (this is just a software-based *estimate,* it’s not actually *measuring* power use).



```
120 mA
120 mA
120 mA
120 mA
120 mA
65 mA
65 mA
65 mA
65 mA
65 mA
21 mA
21 mA
21 mA
21 mA
```

Million-pixels-all-on-white oddly reminds me of the Swensen’s ice cream sundae bar of my youth. Liking each of the toppings individually, I’d heap *everything* on my sundae with no regard to whether the flavors mingled well, ending up with a sickeningly sweet mess. NeoPixels are like that sometimes. A tasteful result stems from showing some restraint.

Strategy: Use. Fewer. Pixels.

I can’t emphasize this one enough. That’s why it’s first.

Don’t use 144-LED-per-meter NeoPixel strip where 60/m will do. Don’t use 60/m where 30/m will do. Don’t encrust an *entire thing* in NeoPixels when some individually-placed pixels or shapes will make a good impression. *A candy red sports car is dazzling, but a well-placed racing stripe can be just as eye-catching.*

Careful NeoPixel placement not only **saves power**, it also requires **less RAM** on the microcontroller, can **improve frame rates**, and with fewer points of failure can make a project **more robust.**

Consider using “Mini Skinny” NeoPixel strips. Not as bright as their full-size brethren, but need only about 60% as much current.

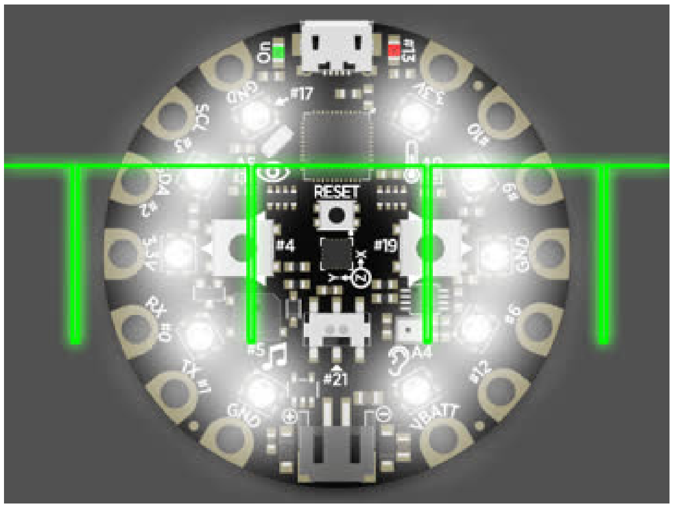
Put some extra effort into everything *around* the pixels; don’t make them a gimmick. Design a prop or a character with strong appeal (https://adafru.it/waA) even if it had *no NeoPixels at all!* Make them part of a bigger thing.

Strategy: Tone Down the Brightness

When you first run the demo sketch, the NeoPixels all start in an “off” state. Tap the right button once to turn them all on, white at maximum brightness.

Immediately, you’ll notice something: **this really hurts to look at.** Please don’t stare directly at it.

It’s a novice impulse to want “MOAR!” of everything. But really, **it’s okay, perfectly legal, to light some or all NeoPixels to less than full brightness.** In fact, by default the Circuit Playground board runs them under 10 percent… anything brighter must be specifically unlocked (the “setBrightness(255)” call in the demo sketch).



NeoPixels use a technique called ***pulse width modulation*** **(PWM)** to control brightness and color. The LED elements inside are switched on and off very quickly (about 400 to 1000 times a second, depending on the generation of NeoPixel). The ***duty cycle*** — a measure of the “on” vs “off” time — determines the perceived brightness. Our brains smooth this out and see a continuous brightness level rather than all that flickering.

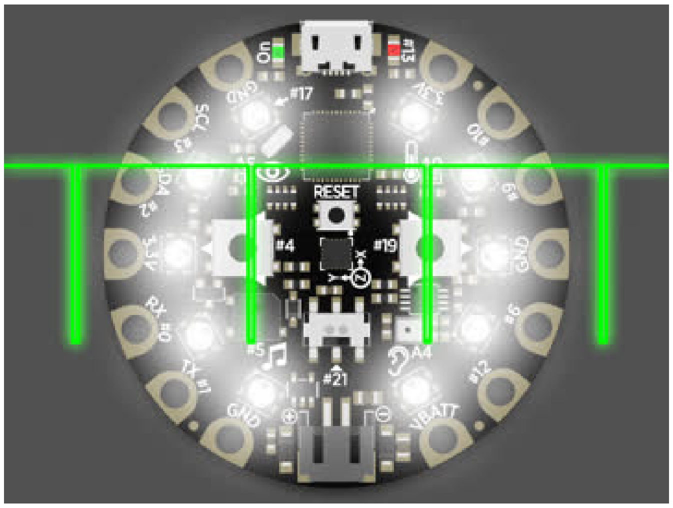
Duty cycle bears a *direct* relationship to power use. Run a NeoPixel at half the duty cycle, and it will use half the current.

Let me repeat that: **Duty cycle bears a direct relationship to power use.**

USELESS TRIVIA: even at their maximum setting (255), NeoPixels never *quite* have a 100% duty cycle…there’s always a tiny off “blip” in the cycle. A minimum setting (0) does correspond to a 0% duty cycle though. Also, you can *never* have *exactly* one half the maximum duty cycle. 255÷2 = 127.5, but NeoPixels expect an 8-bit *integer.* So there’s 127 (49.8% of longest duty cycle) or 128 (50.2%). These are *tiny* differences and perceptually will make no difference, but it's fun to contemplate.

Strategy: Gamma Correction

I really wanted to save this as a grand finale…the power savings can be quite dramatic…but it's a *vital* concept in subsequent examples.



From the "max brightness" mode, tap the right button twice to cycle through 50% and 0% duty cycle, then the left button twice to return through 50% and 100%. Go back and forth a few times.

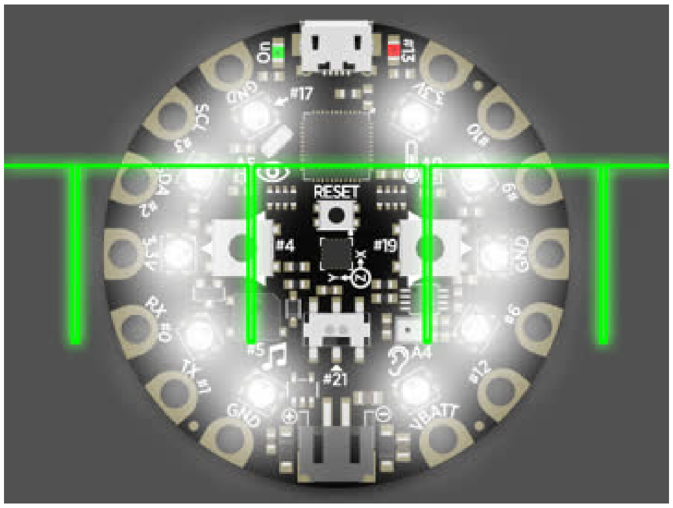
You'll notice something odd here. The 50% duty cycle state seems **disproportionately bright.** You can verify in the code though, could even hook up instruments to measure it. **It really is a 50% duty cycle** (or very close to it, per "useless trivia" above).

This means you can get *most* of the brightness from NeoPixels using only *half* the power. This is hugely important.

This is a real thing, a biological phenomenon of how our eyes and brains work. And we can exploit this to save a *ton* of power! In fact we've written an entire guide on just this one idea:

LED Tricks: Gamma Correction (https://adafru.it/w2B)

Gamma correction is a process of compensating for this disparity between numerical brightness and perceived brightness. To make something *appear* half as bright, we use *much less* than half the duty cycle…in fact it's about 20%. So it's not just a 2:1 reduction in power use in this case, but a tremendous 5:1!



Following the 100/50/0% modes, tap back and forth through the next three modes. You'll see the middle state now looks more "correct" — it seems closer to half-as-bright as the maximum.

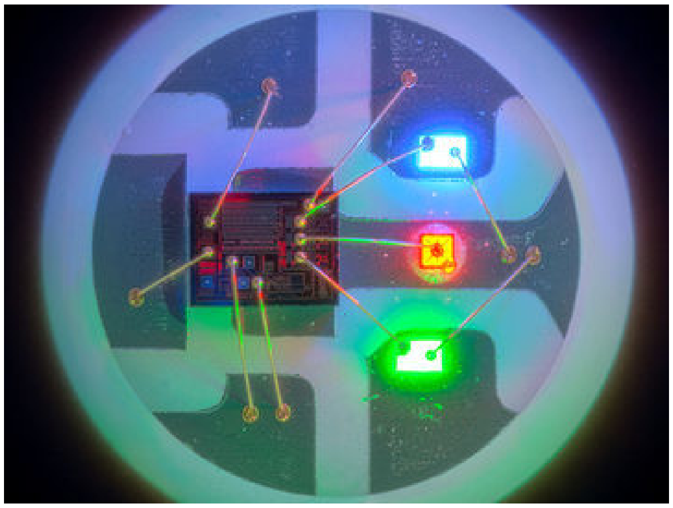
Open the Serial Monitor window and look at the difference!

This adjustment (and some corresponding power savings) works on *any* brightness value in-between the minimum and maximum. There's a PROGMEM table in the example sketch for this…look up an index from 0 to 255, it contains an adjusted value (also 0–255) to assign a pixel's brightness:

```
out = pgm_read_byte(&gammaTable[in]);
```

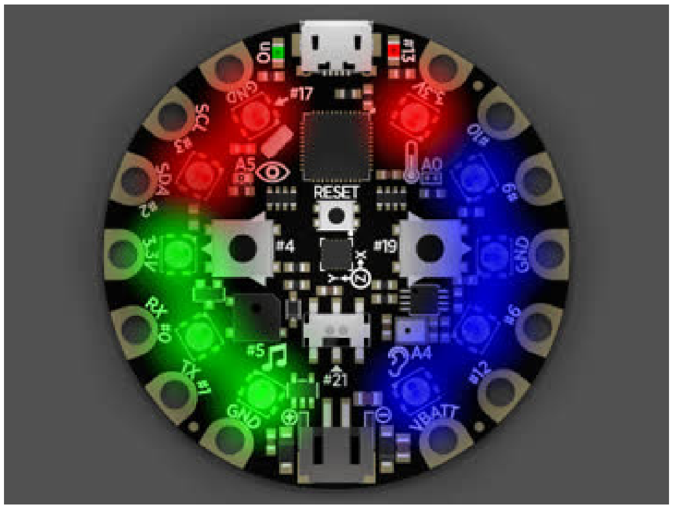
For colors, you'd repeat this for red, green and blue (and white, if using RGBW NeoPixels).

Strategy: Color Selection



Look at a NeoPixel under a microscope and you'll see something fascinating: tiny, separate LED elements for **red**, **green** and **blue.** *Any* color a NeoPixel shows is made by **mixing these three primary colors using PWM.** And if an element isn't being used as part of the color mix, **it's not using any current.**

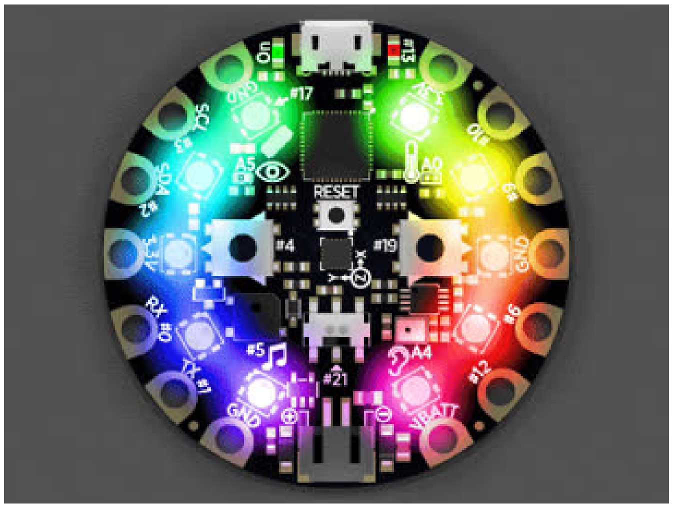
Therefore: primary colors (red, green, blue) use about **1/3 the current** of white. **Secondary colors** (cyan magenta, yellow) use about **2/3** the current. Favor these colors over white and you can reduce your battery needs considerably!



Tap the right button to get to this **RGB pinwheel**.

There's **color**, there's **motion**, and **every single pixel is lit**. But as it's all **primary colors**, this steadily uses about **1/3 the amount of current** of the all-white mode.

Because the colors are at maximum brightness, this mode doesn't benefit from gamma correction, but we'll return to that idea in a moment.

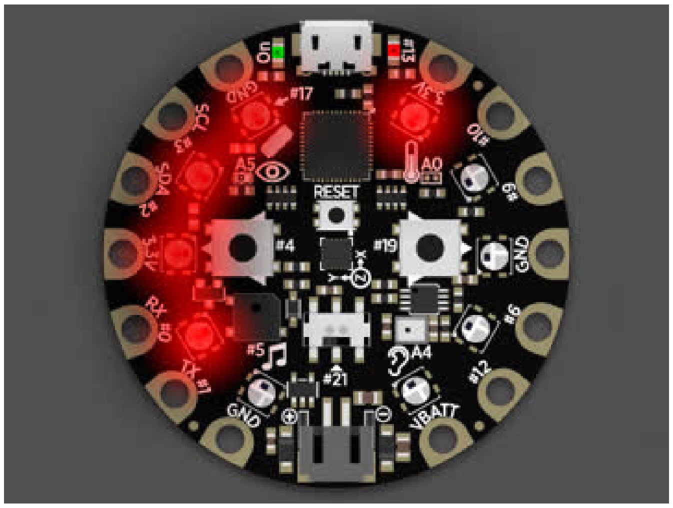


Tap again for this rainbow swirl, a staple of NeoPixel demos! Again **every pixel** is lit, this time a **mix** of **primary** and **secondary** colors and **everything in-between**.

An interesting artifact of the math behind this pattern is that the average duty cycle is close to 50%.

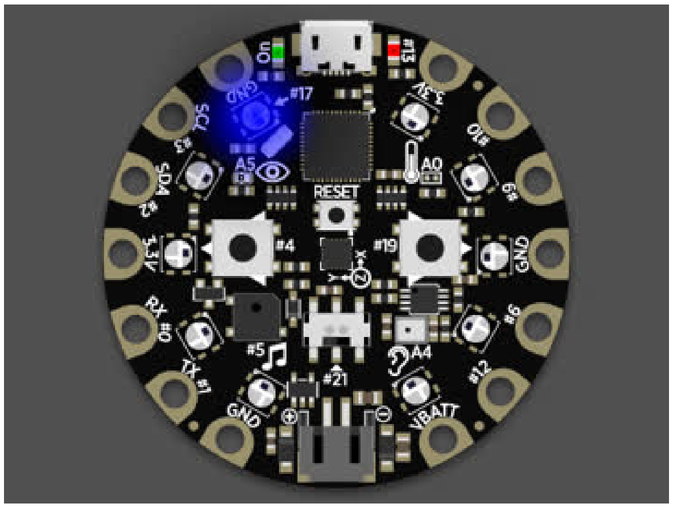
Tapping once more shows an almost, but not quite identical rainbow. *Now it's using gamma correction.* But since only a fraction of the colors are "in-between" (most are 0 or 100%), this effect doesn't benefit much. In the Serial Monitor you'll see there's some modest power savings, and the colors are just a tiny bit dimmer and more "true."

Strategy: Light Fewer Pixels



Simple idea. Light the *least* amount of NeoPixels while still doing something interesting. The fewer the better.

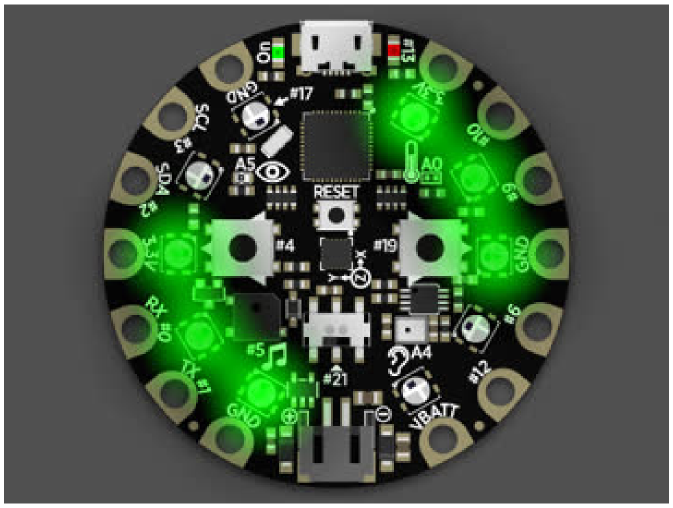
Tap right again. This red spinner has **half the NeoPixels off** at any moment. And because it's a primary color, that's another 3-fold reduction vs. white. 1/2 × 1/3 = **1/6** (about 17%) of our initial "rule of thumb" current.



Tap again. This sparkle effect takes the least-pixels idea to its extreme, with only **one** NeoPixel lit at a time. This can be quite dazzling done in white, but any bright color will do…we'll use blue here just to emphasize the primary color trick again, and because folks are still obsessed with blue LEDs. Take a look in the Serial Monitor and see how little this needs.

Combo Effects

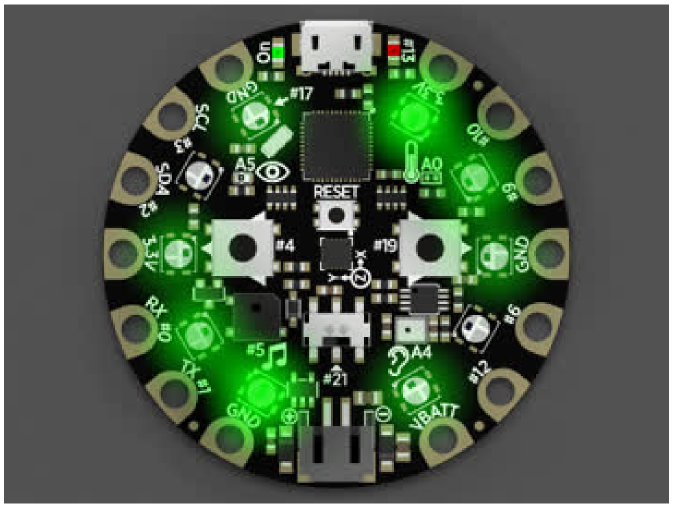
Any or all of the above strategies can be used **in combination**…the energy-saving effects are **cumulative**.



Let's begin with this green spinner (tap right, you know the drill)…

On average, **half** the NeoPixels are off (it varies from 4 to 6 per frame…but on average, half).

Green was chosen because it's a **primary color**. So already, we're **combining two strategies**. The current draw is about **1/6** the worst-case value. And we're **just getting started**.



Tapping again switches to this sine wave spinner. Rather than on-or-off, the brightness of each pixel is determined by a moving sine wave. ***I really like effects like this****, they give electronic projects a more "living" feel than the usual blippy animations. But I digress...*

The sin() function is avoided because it's slow. Instead, a precomputed PROGMEM array — sineTable[] — holds 256 values, varying from 0 to 255 in proportion through one cycle of the wave.

Interestingly, because the *average* value in the sine table is about *one half* of the *maximum* value, this effect uses roughly the **same amount** of current as the simple on/off green spinner above, despite the fact that **more pixels are lit** (though varying in brightness) at any given time. Averaging out all the duty cycles comes close to **50%**. So we get a **way cooler effect** and are still at the same **1/6 current draw** just as before.

The sine table lookup in the demo sketch works a bit like this:

```
uint8_t y = pgm_read_byte(&sineTable[x]);  // x = 0 to 255, y = 0 to 255
```

Tapping right again, a slight variation on the sine wave spinner. Now we've added gamma correction. Something like this:

```
uint8_t y = pgm_read_byte(&sineTable[x]);  // x = 0 to 255, y = 0 to 255
uint8_t z = pgm_read_byte(&gammaTable[y]); // z = 0 to 255 (gamma-corrected sine)
```

The sine wave is a bit dimmer but also more "defined" — the "ons" and "offs" appear more balanced. It **looks better** and also the power consumption has **dropped by about a third.**

One more tap...the last mode refers back to one of the first strategies. Maybe we don't need the sine wave peaks at full brightness. Maybe half the perceived brightness would suffice.

```
uint8_t y = pgm_read_byte(&sineTable[x]) / 2; // x = 0 to 255, y = 0 to 127
uint8_t z = pgm_read_byte(&gammaTable[y]);    // z = 0 to 39
```

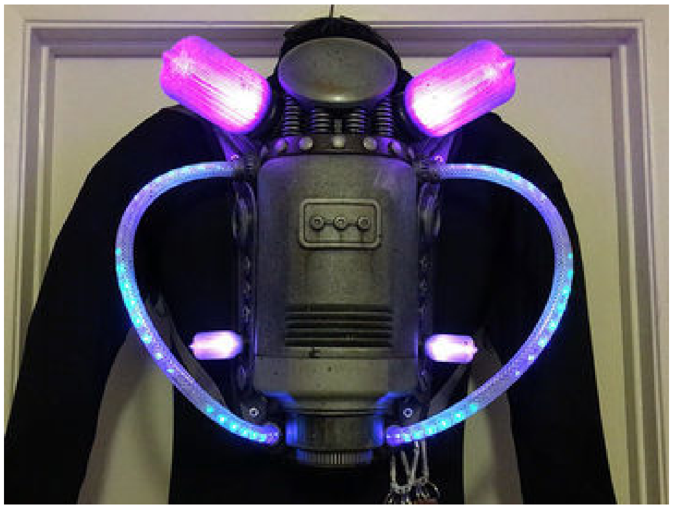
We've still got that lovely organic sine wave effect, just less blinding now. The power consumption though...

Combining several strategies — using primary colors, limiting brightness, and applying gamma correction — this animation needs only about **7 milliamps** over the base current (about 10 mA) used by the microcontroller, NeoPixels and other components on the board. 17 mA total.

17 milliamps. Down from an initial 340 mA with everything running full-blast. That's a **20-fold reduction** from the rule-of-thumb estimate.

Putting It All Together

A friend had approached me about helping to complete a **cosplay** project in short order. It was to be a backpack for a cyborg character, with some lighting effects. Normally I'd decline, but there were NeoPixels involved. I'm sure you understand.



The backpack itself was an assemblage of **sporting goods**, **hardware** and **thrift store parts**, along with some **3D printing** and a **paint job** to give it a weathered look. There are whole blogs and forums and web sites devoted to these topics in depth, much better than I could cover here, so we won't dwell on this aspect in this guide, which is mostly software.

The **electronic** parts — most from Adafruit — included:

- 1 meter of Mini Skinny NeoPixel strip, 60 LEDs/meter (https://adafru.it/lFs) (cut into two half-meter segments)
- 2 NeoPixel Jewels (https://adafru.it/e8a)
- 2 Flora RGB Smart NeoPixels (https://adafru.it/w3e)
- 5V Trinket microcontroller (https://adafru.it/dyV)
- 3 AA alkaline batteries (https://adafru.it/w3A)
- Battery holder with switch (https://adafru.it/w3B)

Battery Selection

Folks are often surprised at my recommendation for alkaline cells in cosplay projects instead of something more modern and rechargeable. *All* battery types have pros and cons. Cosplay is an unusual situation…traveling, away from home and one's workbench…the choice isn't necessarily made on *technical* merits, but on **Murphy's Law** and the "**KISS principle**."

- **Nickel-metal hydride (NiMH) cells:** forget to pack your charger and you're out of commission. Someone "helps" by putting alkaline cells in your 4-cell holder and fries your 5V electronics (four NiMH=4.8V, alkaline=6V).

- **USB power bank:** most will automatically shut off when connected to a low-current circuit. Also, USB plugs do not latch strongly and they stick way out.
- **Lithium-polymer (LiPo) cells:** Somewhat fragile, potentially flammable. Because cosplay can limit vision and mobility, and because costume materials are sometimes themselves flammable, adding an incendiary device to the mix *really* scares me. Also the pack-the-charger problem again. A USB charge circuit *can* be built in, but this adds cost and complexity to something you only use a few times a year.

Alkalines may not be "sexy" but they're ubiquitous and cheap (unless purchased in a hotel gift shop, don't do that). You can travel light and buy them at your destination. And no probing from the TSA.

(I'm *not* saying to *never* use other varieties…all have a place in portable projects…just that this route is beginner-friendly and is affordable. The alkaline battery holder is four bucks!)

Doing *The Math™*

So this circuit has **16 regular-size** (two 7-pixel Jewels, two Flora pixels) and **60 mini NeoPixels**. Doing The Math to estimate a worst-case current use for all these pixels:

16 × 60 + 60 × 35 = 3060 mA

Three amps! Consider, your typical **alkaline AA** cell has a capacity around **2,500 mA**…so this would run for **less than an hour**, assuming the battery could even deliver that kind of current (no…they fare best **under one Amp**, the lower the better).

Switching out batteries all day is no fun. Cosplay is powered by caffeine, adrenaline and foolishness and ideally should go on for *hours.* **Time to put all those strategies to use…**

Designing The Code

I did *not* actually plan everything out and budget for power, instead just running with power-saving approaches I've internalized from prior projects. Basically, the more "off" the pixels — the more "spaces between the notes," the better. I had a hunch that four or possibly five hours' run time could be achieved…

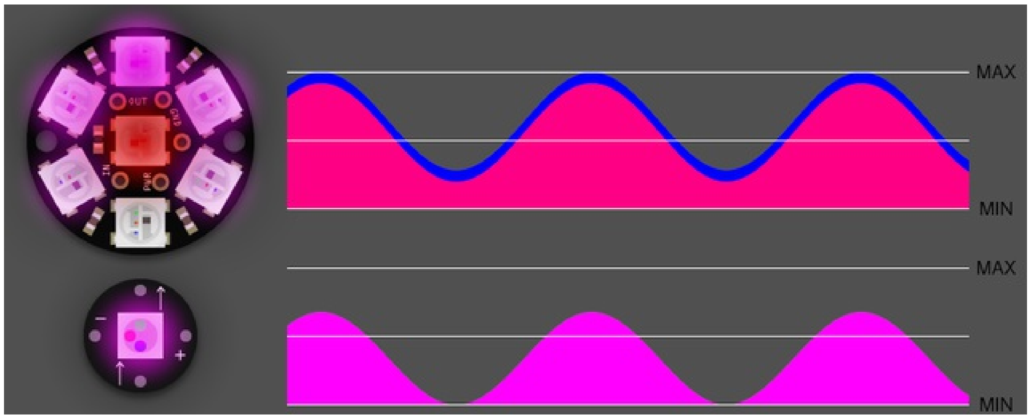
Some artistic license could be taken, proportions were different and time was short, but I didn't have *complete* freedom in color selection…for example, the big "vacuum tube" things *had* to be **purple** (a secondary color)…partly because it's **based on an existing toy**, partly because the cosplayer **loves all things purple**.

Original figure on left, cosplay on right. Photos used with permission.

Graphing the brightness of the "vacuum tubes" — there's two big ones up top, and two smaller ones on the side —

these could have simply been lit solid purple and call it done, but I added sine waves to both (except for the one red pixel at the center)…partly to make it more **visually interesting**, partly to **save power.** For the NeoPixel jewel, each pixel is 60 degrees out of phase along the wave.

The smaller tubes never go to full brightness, it just wasn't necessary and looked more "balanced" with the large ones.

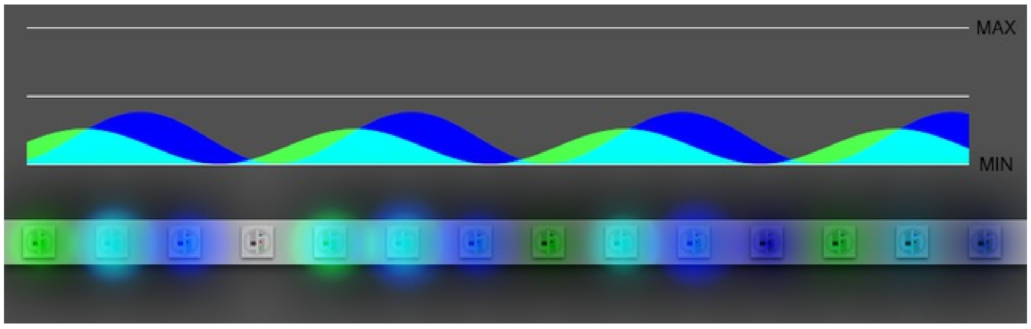


These graphs aren't very precise, they're just to give an impression of what the software is calculating. Also, these are the perceived brightness levels *before* gamma correction. The actual duty cycle assigned to the LEDs is a fraction of this!

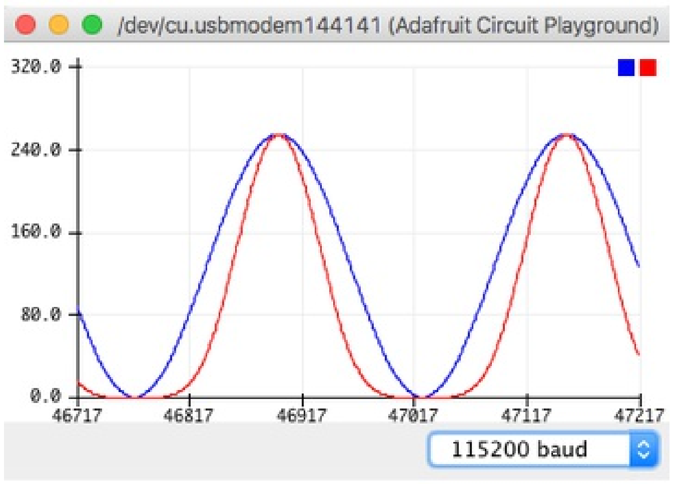
The hoses are even more subdued. Though the Mini Skinny NeoPixels are already somewhat dimmer than the full-sized variety, the brightness was further turned *way* down on these. Power saving, of course, but also because I didn't want the LEDs to be overly distracting and **upstage** the rest of the costume.

The compulsion to run a rainbow cycle on NeoPixel strips is a force of nature. But again, so as not to overpower the rest of the suit (which is mostly muted colors), the palette was kept toward the cyan side of things. Think of early computer graphics, with all its garish, overdone colors…the medium gained respectability as artists dialed down the palette toward nature.

Rather than just sine-waving pure cyan, there are two waves slightly out of phase, so you see pulses of green-cyan-blue travel down the line. It really *looks* as though there's some purple mixed in as well, but this is just a reflection of the vacuum tubes.



Again the graph is before gamma correction is applied. The actual "on" time of any of these LEDs is *tiny.*

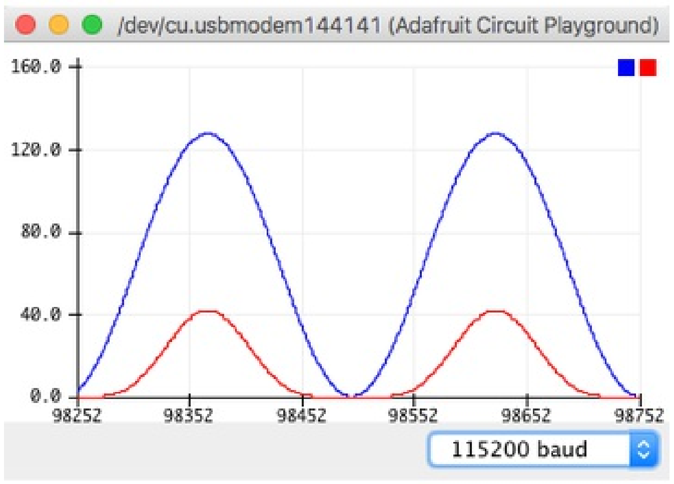


Oh, of course! I'll use Arduino's *Serial Plotter* window to actually graph some gamma-corrected sine waves so you can see the real difference…

Blue is the *perceived* brightness, red is the *gamma-corrected* level issued to the LEDs.

First image depicts a sine wave with the peaks at full brightness (255). Average perceived value is 50%. Average corrected duty cycle is 33%.

Second image is a half-brightness wave (127 max — the vertical scale is stretched here, note the values on left). Average perceived value is 25%. Average duty cycle is 5.5%. Big change!



How'd It Go?

There are many vagaries in trying to do math and estimate the total current draw and run time. Instruments and meters are one option. Sometimes it’s just easier to pop in a fresh set of batteries, switch it on, take note of the time and check on it periodically. As the battery runs down, at some point the voltage will drop to a level where the microcontroller **locks up**. The LEDs will stay lit in their last configuration for a while, but no longer animate. Take note of that time, subtract the starting time, and **there’s your run time.** If it’s insufficient, get into the code and adjust your levels and effects for better efficiency.

I’d hoped for a five hour run time. Nothing too scientific, just gut feeling with the LED count and the numbers I was using in the code. The actual run time for this project was **sixteen hours** from three AA batteries! So *all those NeoPixels* are sipping a mere 160 milliamps (~2,500 ÷ 16), not the 500 mA I’d guesstimated. Even with the secondary colors, it’s getting a *19-fold* improvement over the crude rule-of-thumb estimate. **You *can* make cool stuff work with modest batteries!** If space and weight were at a premium, I probably could even have run this from AAA cells (about half the capacity of AA’s).

Think about that for a moment: in terms of average, overall duty cycle, the **LEDs are off something like 95% of the time**. We just notice all the motion and colors.

This “Guggenhat” project (https://adafru.it/tfU) with *hundreds* of NeoPixels also does its job just fine with lowly AA batteries.

Scale the run time down by at least **20 percent** to allow yourself some emergency reserve…conventions are full of surprises. Also, it’s futile to keep track of the cumulative run time when using this off-and-on over a weekend. So if you’re doing something **important**…a masquerade event, a video shoot, surprising the artist with your homage…**put in fresh batteries!** Eking out those last few joules is *not* worth ruining a performance before an audience! Save any partially-run cells and use them in your TV remote or something.

Demo Code

This Arduino sketch is designed for the Adafruit Circuit Playground board. But the concepts explained on the prior pages are applicable to *any* NeoPixel circuit.

Use the **left and right buttons** to switch between different NeoPixel display modes.

The **Serial Monitor** window shows a rough estimate of the required current in a given mode.

```
// Sketch to accompany "Sipping Power With NeoPixels" guide.  Designed for
// Adafruit Circuit Playground but could be adapted to other projects.

#include <Adafruit_CircuitPlayground.h>

// GLOBAL VARIABLES -------------------------------------------------------

// This bizarre construct isn't Arduino code in the conventional sense.
// It exploits features of GCC's preprocessor to generate a PROGMEM
// table (in flash memory) holding an 8-bit unsigned sine wave (0-255).
const int _SBASE_ = __COUNTER__ + 1; // Index of 1st __COUNTER__ ref below
#define _S1_ (sin((__COUNTER__ - _SBASE_) / 128.0 * M_PI) + 1.0) * 127.5 + 0.5,
#define _S2_ _S1_ _S1_ _S1_ _S1_ _S1_ _S1_ _S1_ _S1_ // Expands to 8 items
#define _S3_ _S2_ _S2_ _S2_ _S2_ _S2_ _S2_ _S2_ _S2_ // Expands to 64 items
const uint8_t PROGMEM sineTable[] = { _S3_ _S3_ _S3_ _S3_ }; // 256 items

// Similar to above, but for an 8-bit gamma-correction table.
#define _GAMMA_ 2.6
const int _GBASE_ = __COUNTER__ + 1; // Index of 1st __COUNTER__ ref below
#define _G1_ pow((__COUNTER__ - _GBASE_) / 255.0, _GAMMA_) * 255.0 + 0.5,
#define _G2_ _G1_ _G1_ _G1_ _G1_ _G1_ _G1_ _G1_ _G1_ // Expands to 8 items
#define _G3_ _G2_ _G2_ _G2_ _G2_ _G2_ _G2_ _G2_ _G2_ // Expands to 64 items
const uint8_t PROGMEM gammaTable[] = { _G3_ _G3_ _G3_ _G3_ }; // 256 items

// These are used for 'debouncing' the left & right button inputs,
// for switching between modes.
boolean  prevStateLeft, prevStateRight;
uint32_t lastChangeTimeLeft = 0, lastChangeTimeRight = 0;
#define  DEBOUNCE_MS 15 // Button debounce time, in milliseconds

// These are used in estimating (very approximately) the current draw of
// the board and NeoPixels.  BASECURRENT is the MINIMUM current (in mA)
// used by the entire system (microcontroller board plus NeoPixels) --
// keep in mind that even when "off," NeoPixels use a tiny amount of
// current (a bit under 1 milliamp each).  LEDCURRENT is the maximum
// additional current PER PRIMARY COLOR of ONE NeoPixel -- total current
// for an RGB NeoPixel could be up to 3X this.  The '3535' NeoPixels on
// Circuit Playground are smaller and use less current than the more
// common '5050' type used in NeoPixel strips and shapes.
#define BASECURRENT 10
#define LEDCURRENT  11 // Try using 18 for '5050' NeoPixels
uint8_t  frame = 0;    // Frame count, results displayed every 256 frames
uint32_t sum   = 0;    // Cumulative current, for calculating average
uint8_t *pixelPtr;     // -> NeoPixel color data

// This array lists each of the display/animation drawing functions
// (which appear later in this code) in the order they're selected with
// the right button.  Some functions appear repeatedly...for example,
// we return to "mode_off" at several points in the sequence.
```

```
void (*renderFunc[])(void) {
  mode_off, // Starts here, with LEDs off
  mode_white_max, mode_white_half_duty      , mode_off,
  mode_white_max, mode_white_half_perceptual, mode_off,
  mode_primaries, mode_colorwheel, mode_colorwheel_gamma,
  mode_half, mode_sparkle,
  mode_marquee, mode_sine, mode_sine_gamma, mode_sine_half_gamma
};
#define N_MODES (sizeof(renderFunc) / sizeof(renderFunc[0]))
uint8_t mode = 0; // Index of current mode in table

// SETUP FUNCTION -- RUNS ONCE AT PROGRAM START ----------------------------

void setup() {
  CircuitPlayground.begin();
  CircuitPlayground.setBrightness(255); // NeoPixels at full brightness
  pixelPtr = CircuitPlayground.strip.getPixels();

  Serial.begin(19200);

  prevStateLeft  = CircuitPlayground.leftButton(); // Initial button states
  prevStateRight = CircuitPlayground.rightButton();
}

// LOOP FUNCTION -- RUNS OVER AND OVER FOREVER -----------------------------

void loop() {

  // Read and debounce left/right buttons
  uint32_t t = millis();
  if((t - lastChangeTimeLeft) >= DEBOUNCE_MS) {
    boolean b = CircuitPlayground.leftButton();
    if(b != prevStateLeft) {             // Left button state changed?
      prevStateLeft      = b;
      lastChangeTimeLeft = t;
      if(b) {                            // Left button pressed?
        if(mode) mode--;                 // Go to prior mode
        else     mode = N_MODES - 1;     // or "wrap around" to last mode
        frame = sum = 0;                 // Reset power calculation
      }
    }
  }
  if((t - lastChangeTimeRight) >= DEBOUNCE_MS) {
    boolean b = CircuitPlayground.rightButton();
    if(b != prevStateRight) {            // Right button state changed?
      prevStateRight      = b;
      lastChangeTimeRight = t;
      if(b) {                            // Right button pressed?
        if(mode < (N_MODES-1)) mode++;   // Advance to next mode
        else                   mode = 0; // or "wrap around" to start
        frame = sum = 0;                 // Reset power calc
      }
    }
  }

  (*renderFunc[mode])();          // Render one frame in current mode
  CircuitPlayground.strip.show(); // and update the NeoPixels to show it

  // Accumulate total brightness value for all NeoPixels (assumes RGB).
  for(uint8_t i=0; i<CircuitPlayground.strip.numPixels() * 3; i++) {
    sum += pixelPtr[i];
  }
  if(!++frame) { // Every 256th frame, estimate & print current
    Serial.print(BASECURRENT + (sum * LEDCURRENT + 32640) / 65280);
    Serial.println(" mA");
    sum = 0; // Reset pixel accumulator
  }
}

// RENDERING FUNCTIONS FOR EACH DISPLAY/ANIMATION MODE ---------------------

// All NeoPixels off
void mode_off() {
  CircuitPlayground.strip.clear();
}

// All NeoPixels on at max: white (R+G+B) at 100% duty cycle
void mode_white_max() {
  for(uint8_t i=0; i<10; i++) {
    CircuitPlayground.strip.setPixelColor(i, 0xFFFFFF);
  }
}

// All NeoPixels on at 50% duty cycle white.  Numerically speaking,
// this is half power, but perceptually it appears brighter than 50%.
void mode_white_half_duty() {
  for(uint8_t i=0; i<10; i++) {
    CircuitPlayground.strip.setPixelColor(i, 0x7F7F7F);
  }
}

// All NeoPixels on at 50% perceptial brightness, using gamma table lookup.
// Though it visually appears to be about half brightness, numerically the
// duty cycle is much less, a bit under 20% -- meaning "half brightness"
// can actually be using 1/5 the power!
void mode_white_half_perceptual() {
  uint32_t c = pgm_read_byte(&gammaTable[127]) * 0x010101;
  for(uint8_t i=0; i<10; i++) {
    CircuitPlayground.strip.setPixelColor(i, c);
  }
}

// Cycle through primary colors (red, green, blue), full brightness.
// Because only a single primary color within each NeoPixel is turned on
// at any given time, this uses 1/3 the power of the "white max" mode.
void mode_primaries() {
  uint32_t c;
  for(uint8_t i=0; i<10; i++) {
    // This animation (and many of the rest) pretend spatially that there's
    // 12 equally-spaced NeoPixels, though in reality there's only 10 with
    // gaps at the USB and battery connectors.
    uint8_t j = i + (i > 4);     // Mind the gap
    j = ((millis() / 100) + j) % 12;
    if(j < 4)      c = 0xFF0000; // Bed
    else if(j < 8) c = 0x00FF00; // Green
    else           c = 0x0000FF; // Blue
    CircuitPlayground.strip.setPixelColor(i, c);
  }
}
```

```
// HSV (hue-saturation-value) to RGB function used for the next two modes.
uint32_t hsv2rgb(int32_t h, uint8_t s, uint8_t v, boolean gc=false) {
  uint8_t n, r, g, b;

  // Hue circle = 0 to 1530 (NOT 1536!)
  h %= 1530;           // -1529 to +1529
  if(h < 0) h += 1530; //     0 to +1529
  n  = h % 255;        // Angle within sextant; 0 to 254 (NOT 255!)
  switch(h / 255) {    // Sextant number; 0 to 5
   case 0 : r = 255    ; g =   n    ; b =   0    ; break; // R to Y
   case 1 : r = 254 - n; g = 255    ; b =   0    ; break; // Y to G
   case 2 : r =   0    ; g = 255    ; b =   n    ; break; // G to C
   case 3 : r =   0    ; g = 254 - n; b = 255    ; break; // C to B
   case 4 : r =   n    ; g =   0    ; b = 255    ; break; // B to M
   default: r = 255    ; g =   0    ; b = 254 - n; break; // M to R
  }

  uint32_t v1 =   1 + v; // 1 to 256; allows >>8 instead of /255
  uint16_t s1 =   1 + s; // 1 to 256; same reason
  uint8_t  s2 = 255 - s; // 255 to 0

  r = ((((r * s1) >> 8) + s2) * v1) >> 8;
  g = ((((g * s1) >> 8) + s2) * v1) >> 8;
  b = ((((b * s1) >> 8) + s2) * v1) >> 8;
  if(gc) { // Gamma correct?
    r = pgm_read_byte(&gammaTable[r]);
    g = pgm_read_byte(&gammaTable[g]);
    b = pgm_read_byte(&gammaTable[b]);
  }
  return ((uint32_t)r << 16) | ((uint16_t)g << 8) | b;
}

// Rotating color wheel, using 'raw' RGB values (no gamma correction).
// Average current use is about 1/2 of the max-all-white case.
void mode_colorwheel() {
  uint32_t t = millis();
  for(uint8_t i=0; i<10; i++) {
    uint8_t j = i + (i > 4);
    CircuitPlayground.strip.setPixelColor(i,
      hsv2rgb(t + j * 1530 / 12, 255, 255, false));
  }
}

// Color wheel using gamma-corrected values.  Current use is slightly less
// than the 'raw' case, but not tremendously so, as only 1/3 of pixels at
// any time are in transition cases (else 100% on or off).
void mode_colorwheel_gamma() {
  uint32_t t = millis();
  for(uint8_t i=0; i<10; i++) {
    uint8_t j = i + (i > 4);
    CircuitPlayground.strip.setPixelColor(i,
      hsv2rgb(t + j * 1530 / 12, 255, 255, true));
  }
}

// Cycle with half the pixels on, half off at any given time.
// Simple idea.  Half the pixels means half the power use.
void mode_half() {
  uint32_t t = millis() / 4;
  for(uint8_t i=0; i<10; i++) {
    uint8_t j = t + i * 256 / 10;
    j = (j >> 7) * 255;
    CircuitPlayground.strip.setPixelColor(i, j * 0x010000);
  }
}

// Blue sparkles.  Randomly turns on ONE pixel at a time.  This demonstrates
// minimal power use while still doing something "catchy."  And because it's
// a primary color, power use is even minimal-er (see 'primaries' above).
void mode_sparkle() {
  static uint8_t randomPixel = 0;
  if(!(frame & 0x7F)) {              // Every 128 frames...
    CircuitPlayground.strip.clear(); // Clear pixels
    uint8_t r;
    do {
      r = random(10);                // Pick a new random pixel
    } while(r == randomPixel);       // but not the same as last time
    randomPixel = r;                 // Save new random pixel index
    CircuitPlayground.strip.setPixelColor(randomPixel, 0x0000FF);
  }
}

// Simple on-or-off "marquee" animation w/ about 50% of pixels lit at once.
// Not much different than the 'half' animation, but provides a conceptual
// transition into the examples that follow.
void mode_marquee() {
  uint32_t t = millis() / 4;
  for(uint8_t i=0; i<10; i++) {
    uint8_t j = i + (i > 4);
    uint8_t k = (t + j * 256 / 12) & 0xFF;
    k = ((k >> 6) & 1) * 255;
    CircuitPlayground.strip.setPixelColor(i, k * 0x000100L);
  }
}

// Sine wave marquee, no gamma correction.  Avg. overall duty cycle is 50%,
// and combined with being a primary color, uses about 1/6 the max current.
void mode_sine() {
  uint32_t t = millis() / 4;
  for(uint8_t i=0; i<10; i++) {
    uint8_t j = i + (i > 4);
    uint8_t k = pgm_read_byte(&sineTable[(t + j * 512 / 12) & 0xFF]);
    CircuitPlayground.strip.setPixelColor(i, k * 0x000100L);
  }
}

// Sine wave with gamma correction.  Because nearly all the pixels have
// "in-between" values (not 100% on or off), there's considerable power
// savings to gamma correction, in addition to looking more "correct."
void mode_sine_gamma() {
  uint32_t t = millis() / 4;
  for(uint8_t i=0; i<10; i++) {
    uint8_t j = i + (i > 4);
    uint8_t k = pgm_read_byte(&sineTable[(t + j * 512 / 12) & 0xFF]);
    k = pgm_read_byte(&gammaTable[k]);
    CircuitPlayground.strip.setPixelColor(i, k * 0x000100L);
  }
}
```

```
// Perceptually half-brightness gamma-corrected sine wave.  Sometimes you
// don't need things going to peak brightness all the time.  Combined with
// gamma and primary color use, it's super effective!
void mode_sine_half_gamma() {
  uint32_t t = millis() / 4;
  for(uint8_t i=0; i<10; i++) {
    uint8_t j = i + (i > 4);
    uint8_t k = pgm_read_byte(&sineTable[(t + j * 512 / 12) & 0xFF]) / 2;
    k = pgm_read_byte(&gammaTable[k]);
    CircuitPlayground.strip.setPixelColor(i, k * 0x000100L);
  }
}
```