CLUE Light Paint Stick with CircuitPython

Created by Phillip Burgess

https://learn.adafruit.com/clue-light-paintstick

Last updated on 2023-10-16 05:51:45 PM EDT
# Table of Contents

## Overview
- Parts

## Build the Light Paint Stick
- The Electronics
- The Support Structure

## Code with CircuitPython

## Using the Software
- Troubleshooting

## Make Pixel Art
- Keep Some Space Free
- Light Painter Hacks

## Take Long Exposure Photos
- Tools
- Action
- Floating Images
- Multiple Stamps

## CircuitPython Magic
- How it Works
- Skateboard Tricks
- Math Shenanigans
- RAM Shenanigans
- File Shenanigans
Overview

Light painting is an “in-camera” photographic effect (that is, not composited after-the-fact) using long exposures and moving light sources. The technique dates back over a century, most of that time relying on small flashlights or incandescent bulbs…but once someone had the idea to use a whole row of computer-controlled LEDs, it quickly became one of the go-to maker projects, one we’ve visited many times. Any time a new technology comes along…Arduino (), Raspberry Pi (), CircuitPython ()…light painting is a fun and creative way to put that tech through its paces.

The challenge of this project was to see what could be squeezed from the CircuitPython language on a mid-range microcontroller board — Adafruit’s CLUE (), which also provides a screen and buttons for a minimal interface.

An earlier project, our DotStar Pi Painter (), had the enormous resources of a Raspberry Pi computer at its disposal, and the results were almost too good. Could we get some of those buttery-smooth visuals on a smaller device? A step (or three) up
from the “bitmappy” LED graphics we’d previously seen from CircuitPython? Watch us work it!

Parts

Adafruit CLUE - nRF52840 Express with Bluetooth LE
Do you feel like you just don't have a CLUE? Well, we can help with that - get a CLUE here at Adafruit by picking up this sensor-packed development board. We wanted to build some...
https://www.adafruit.com/product/4500

Adafruit DotStar Digital LED Strip - Black
144 LED/m - 0.5 Meter
Move over NeoPixels, there's a new LED strip in town! These fancy new DotStar LED strips are a great upgrade for people who have loved and used NeoPixel strips for a few years but...
https://www.adafruit.com/product/2328
Adafruit DotStar Digital LED Strip - White
144 LED/m - 0.5 Meter
Move over NeoPixels, there's a new LED strip in town! These fancy new DotStar LED strips are a great upgrade for people who have loved and used NeoPixel strips for a few years but...
https://www.adafruit.com/product/2329

3 x AA Battery Holder with On/Off Switch, JST, and Belt Clip
This battery holder connects 3 AA batteries together in series for powering all kinds of projects. We spec'd these out because the box is compact, and 3 AA's add up to about...
https://www.adafruit.com/product/3287

Alkaline AA batteries (LR6) - 3 pack
Battery power for your portable project! These batteries are good quality at a good price, and work fantastic with any of the kits or projects in the shop that use AAs. This is a pack...
https://www.adafruit.com/product/3521
Build the Light Paint Stick

The Electronics

Our light painter (or light stick) is a bit of electronics affixed to a rigid support. There can be huge variability depending on what skills, tools and materials you have access to...improvising is a common theme in these projects...but for this one there’s three specific items:

- **Adafruit CLUE nRF52840 Express** — a small CircuitPython board with screen and buttons we can use for a rough user interface.
- A length of DotStar LED strip. Either [one-half meter of high-density](#) (144 LEDs/meter, 72 pixels total) or [one meter of standard-density](#) (60/m, 60 pixels total). Not NeoPixel strip...we require the DotStars’ speed here!
- [3 AA battery pack with JST PH connector and switch](#). Alkaline batteries are old school, but this simplifies the electronics considerably. No level-shifting needed.

I’ll mention some of the other bits that went into my build...but keep in mind there are many opportunities to change things up, depending what you have around...

- **STEMMA QT / Qwiic cable (100mm or 200mm)** — uses the CLUE STEMMA port to transfer data to the DotStar strip. But you have options here. With a small change to the code, CLUE pins 1 and 2 could be used with alligator clips. Not rugged, but might be adequate for a one-evening project before rebuilding into something else. Or there’s STEMMA QT cables with [jumper header ends](#), or even [alligator clip ends](#), whatever suits your skills and plans.
- A [4-pin JST SM connector](#) lets you easily disconnect the DotStar strip and use it for other things.
• **Alligator clip test lead** — for this project, whenever files or code on the board need editing, pin 0 must be connected to GND. Any temporary connection could work, even a bent paper clip, but the 'gator clip is convenient.

• Soldering iron and related paraphernalia, some wire and such.

• Heat-shrink tubing in various sizes.

• Something to make a rigid support/handle. Could be a square dowel, some PVC pipe, a sacrificial yardstick. Improvise!

• Crafty bits and bobs like tape, glue, paint, Velcro...read on for ideas and suggestions.

Here's a schematic diagram of how the electronic parts go together:

![Schematic diagram](image)

Keep in mind this is a schematic...it shows all the connections clearly but doesn't represent the components real positions or how the wiring will actually look or be spliced. Plan out the physical placement of everything on your light stick, then make connections using the schematic for reference.

Also, DotStar strips have been known to change from time to time...the pin order at the ends of the strips might be different. It's always a good idea to load up a minimal DotStar test program and verify which pins are clock and data.
As mentioned earlier, it’s also possible to build this using alligator leads and not need the STEMMA cable.

If using DotStar-compatible (APA102) LEDs from another source, the wires be a different color, or you might be soldering to a bare strip.

If you have access to a 3D printer, the Ruiz Bros’ CLUE Slim Case (🔗) lets the board sit flat against the battery pack or square dowel with some double-stick foam tape. Entirely optional, work with what you’ve got...it’s just a nice-to-have addition.
Reiterating a prior point: the STEMMA red wire must NOT connect to the battery! This will wreck your CLUE board!

I went so far as to pluck the red wire out of the STEMMA connector to avoid any possibility of making that mistake. Maybe that’s a bit obsessive.

The Support Structure

Let’s keep this short as it’s been covered many, many times before in other projects. Here’s pages you can skim for ideas:

- A yardstick can be re-purposed, as in this HalloWing light painter project ().
- The NeoPixel Painter () and DotStar Pi Painter () projects used 3/4" square dowel and 1" aluminum square tubing, respectively.
- An early Pi light painter () used PVC pipe and a hula hoop! That’s right, there’s no law that you have to light-paint with a straight line.
- What else do you have around? Even a flap from a large cardboard box, cut and folded into a triangular tube, can provide adequate support.

Other things learned from prior projects:

- Unless something is really shiny and metallic, you really don’t need to paint everything black. It also makes no difference if using black or white DotStar strips. As long as it’s in motion, it won’t register in a long-exposure photo. Mostly, black looks cool.
- A diffuser over the LEDs helps blend them into a cohesive 2D image instead of light streaks. Haven’t found a more convenient material for this than 1" white elastic.
- The waterproof sleeve on LED strips is difficult to attach to anything. Hot tip here is two-sided carpet tape. Alternatives include using cable zip-ties at the ends, or carefully cut off the waterproof sleeve and glue the strip down with hot glue.
A closeup of the elastic diffuser, and using some zip-ties to hold the ends.

This is an old DotStar strip I had around, with the waterproof covering removed and held with double-stick tape. If you’d prefer to keep the covering on, that’s probably for the best...more options to re-use these parts in other projects later!

If you don’t have a mating JST connector, or forgot to order one with the other parts...one trick is to cut the connector off the “out” end of the strip, and use it to connect to the “in” connector, though this does mean you can no longer chain this strip to others. Also, be super extra sure you’re cutting off the OUT end! Look for the arrows on the strip showing the data direction.

Fresh DotStar strips also have extra power wires at both ends. If you’re not using some of these wires, clip off the exposed tips and/or insulate them with some heat-shrink to prevent electric shorts.

Skimming through the prior light painter projects, most of those tended to have a stick and diffuser the same length as the LED strip. Tried something different here since it’s a small half-meter painter: there are hand-hold spots at either end, providing more options for moving it around in photos. Not a requirement, just an idea.
Code with CircuitPython

If your CLUE board isn’t already running CircuitPython…

READ THIS PAGE CAREFULLY BEFORE INSTALLING CODE, or the board will seem unresponsive!

If your CLUE board isn’t already running CircuitPython…

- Head to the CircuitPython download page for CLUE nRF52840 Express () and fetch the latest build.
- Then go to the CircuitPython libraries download page () and fetch the corresponding “bundle” for that version.
- If using Windows 7, you’ll also need a USB driver. This and other setup basics are covered in the Welcome to CircuitPython () and Introducing Adafruit CLUE () guides…if things behave oddly, start there.
- Connect a USB cable from CLUE to your computer and double-tap CLUE’s reset button on the back. When the CLUEBOOT drive appears, drag-and-drop the
CircuitPython .UF2 file to that drive. Any program currently on the board (Arduino, earlier CircuitPython or anything else) will be overwritten.

Next steps after CircuitPython is installed...

- Unzip the CircuitPython bundle and copy two folders — adafruit_display_shapes and adafruit_display_text — to the CIRCUITPY drive lib folder.
- Download the code and images for this project from GitHub:

  Download CLUE Light Painter Project

- Unzip the project and copy the folder bmps-72px to the CIRCUITPY drive.
- Do not copy any of the Python files to the drive until the following is understood:

  When “boot.py” runs at startup, it lets the code write to CLUE’s flash filesystem, but CODE AND FILES ON THE BOARD CAN'T BE EDITED OR DELETED OVER USB.
  boot.py checks for a connection between PIN 0 and GND. If present at boot or reset, code and files CAN be changed, but the light painter won’t run.

To edit or remove files, make this connection between PIN 0 and GND and boot or reset the board. Now you can edit, but the light painter code won’t run.

Remove the connection and boot/reset and the light painter code runs, but you can’t edit. This is normal, it’s just part of how CircuitPython interacts with the flash filesystem and a USB-connected computer.

If you misplace the alligator cable, it is possible to get back into the board, but this involves accessing the CircuitPython REPL (e.g. from a terminal program) and entering some commands to remove or rename the boot.py file, as documented in this guide().
Copy the four .py files to the CIRCUITPY drive:

- bmp2led.py — this processes BMP image files into a format that our light-painting code can read quickly. It’s in a separate file so we have the option to use it in other projects later.
- boot.py — this runs when the board initially boots or is reset, and sets the flash filesystem to write-enabled mode if the alligator jumper is not present.
- code.py — the main light-painting application, using the CLUE’s display and buttons to provide a barebones user interface.
- richbutton.py — a CircuitPython library that processes button inputs to distinguish taps, double-taps and long holds.

Once these files are installed, you may want to edit the code.py file (using your text editor of preference) to configure some things for your particular setup. Remember, the PIN 0 to GND jumper must be in place when you boot or reset the board to edit files. Then remove the jumper and reset to test the changes. You might be going back and forth a few times like this.

Near the top of code.py, look for this section:

```
NUM_PIXELS = 72                   # LED strip length
PIXEL_PINS = board.SDA, board.SCL # Data, clock pins for DotStars
PIXEL_ORDER = 'bgr'               # Pixel color order
PATH = '/bmps-72px'           # Folder with BMP images (or '' for root path)
TEMPFILE = '/led.dat'         # Working file for LED data (will be clobbered!)
FLIP_SCREEN = False           # If True, turn CLUE screen & buttons upside-down
GAMMA = 2.4                   # Correction for perceptually linear brightness
BRIGHTNESS_RANGE = 0.15, 0.75 # Min, max brightness (0.0-1.0)
```

NUM_PIXELS is the length of the DotStar strip — **72** or **60** pixels are both well suited to the performance of this board, but you’re welcome to try building something a different size.

PIXEL_PINS are the DotStar data and clock pins, respectively. Comma-separated. By default these are the board.SDA and board.SCL pins of the STEMMA QT socket (blue and yellow wires, respectively). If you’ve gone with a more temporary build, using ’gator clips to the board edge connector, you can use board.P1 and board.P2 here (avoid P0, already in use in boot.py).

PIXEL_ORDER is the sequence in which DotStar LEDs process the red, green and blue values of each pixel. Currently-shipping DotStar strips use ‘bgr’ order (blue,
green, red)…but older DotStars, and compatible APA-102 LEDs from other sources, may use a different sequence (‘gbr’ was previously common).

**PATH** is the folder where the code looks for BMP images. See the “Pixel Art” page for requirements.

**FLIP_SCREEN** can be set **True** or **False**. If **True**, the display and buttons are flipped 180°…sometimes the physical assembly of the light stick just works out easier with the board turned around, and this compensates for it.

**BRIGHTNESS_RANGE**, two comma-separated values from **0.0** (off) to **1.0** (brightest), are the minimum and maximum LED brightness to use when light painting (a setting in the UI). **0.0** isn’t very useful, since that’s just off, so the default minimum is **0.15**. The maximum is set below **1.0** (**0.85** by default) because the LEDs coming on full brightness can cause a voltage sag…the CLUE board may lock up and, in severe situations, may even require reinstalling CircuitPython. If you see this happen, dial back the maximum value until the DotStars, battery and CLUE all play nice.

**TEMPFILE** and **GAMMA** can be ignored, probably won’t need to change these.

If you make changes but can’t save the file: the alligator jumper wasn’t in place when booting. Connect P0 and GND, reset before editing files. Remove the jumper and reset to run the code.

---

**Using the Software**

The “user interface” is simplistic but gets the job done. With only two buttons on the CLUE board, it requires a little patience to navigate around. The main things to know are:

- Remove the jumper between pin 0 and ground before using the light painter.
• The software distinguishes between button “taps” and “holds” — short and long presses perform different operations. A “hold” does not repeat taps like a keyboard does…it’s a distinct interaction here.

• The display refers to the buttons as L and R (left and right) rather than A and B. There’s a software option to rotate everything 180° if it’s easier to wire up your light stick that way, so “A” isn’t always on the left.

On startup, the code scans the folder defined by the PATH variable (e.g. 'bmps-72px' by default) looking for compatible BMP image files, and builds a list. (“Compatible BMPs” are explained on the “Make Pixel Art” page.)

The first screen displayed then lets you choose one of these images, by tapping the left and right buttons to cycle among them.

This is really the first of several configurable settings. When navigating these settings, let’s call it “config mode.”

By holding the left button, this takes you “up” a level in the configuration menu. Now tapping left/right cycles among other settings: IMAGE, TIME, LOOP and BRIGHTNESS. Find a setting you want to change, then hold left to go “down” a level and change that setting, hold left again to go “up” and pick another setting.

• IMAGE is where you started. This lets you pick an image to load and light-paint with.

• TIME sets how long the LEDs are “on” for light painting. Wider/taller images will usually need longer times, and correspondingly longer camera exposures.

• LOOP sets whether an image “paints” just one time (loop off), or repeats in a continuous cycle (loop on).
• BRIGHTNESS sets the overall image brightness. By default this is at the maximum, but you can dial it down if it helps with your camera exposure, if you want background objects in the scene more visible.

Holding the right button from any config screen will switch over to “paint mode.” You’ll see a LOADING message on the screen, and the image-loading progress is displayed along the DotStar strip. Once loaded, the DotStar strip and the display will turn off (so the backlight doesn’t mess up your photo).

In paint mode, tap either the left or right button to trigger one “paint” of the image while your camera’s shutter is open. The “on” time of the LEDs is determined by the TIME setting, 1 second by default.

It’s a little different if LOOP mode is ON, where tapping left or right starts and stops an ongoing paint cycle, rather than a single pass.

Photos look best if the stick is already in motion when you tap a button...you’ll have a bit of momentum going already, whereas tap-and-then-move will make a “smushed” image. It’s awfully helpful to have a second person working the camera. One person adjusts camera settings and gives directorial feedback on the other’s speed and movement. It can be done solo, it just requires more patience.

From paint mode, hold either the left or right button to return to config mode and make changes, load a different image, etc.

Troubleshooting

If no response from the LEDs, try a known-working CircuitPython DotStar test program, changing the pin numbers to match your wiring. Common issues include:

• Clock and data wires swapped. On the CLUE these can easily be switched in code...it’s not picky about which pin has which job.
• Verify there’s a GND connection from CLUE to the DotStar strip.
• If testing with CLUE plugged into USB and DotStars powered from the battery pack, verify the power switch is ON.
Make Pixel Art

Images require some resizing and conversion in preparation for the light painter. The code can only read one specific format — 24-bit BMP — because we currently lack the ability to decode complex formats like JPEG or PNG.

The BMP image format is sometimes called “Microsoft Bitmap” or “Windows Bitmap,” but there’s really nothing Windows-specific about it; plenty of software on Mac and Linux handles the format just as well, typically an option in a “Save As...” or “Export...” dialog box. Software like The GIMP, Pixelmator or Photoshop can export this format.

Be sure to select 24-bit BMP. Not 1- or 8- or 32-bit. The CircuitPython code only handles the 24-bit variety.

If painting an image vertically (that is, holding the strip horizontally and lifting it upward like a barbell), the image should be resized so its width matches the DotStar strip length (e.g. 72 pixels, unless you’ve made a custom variant). Any wider and it’ll be cropped, any narrower and the strip isn’t fully utilized. No scaling is performed.

Vertical images are painted bottom to top. This might seem odd, but is on purpose: the ground or tabletop provides a consistent point of reference for starting. If you try painting from the top down, you may bump into the ground before the image is finished.
If painting an image horizontally (holding the strip vertically, moving left to right across the camera's field of view), the image should be resized so its height matches the strip length (e.g. 72 pixels) and then rotated 90 degrees counter-clockwise (so the top of the image is now at the left side) before saving.

This extra step is necessary to reduce the amount of processing done on the CLUE board; it would otherwise take several minutes (instead of seconds) to decode an image when loading.

If LOOP mode is ON while painting, the image will be repeated. This can be used to create patterns...
Keep Some Space Free

Image files are large. BMP images doubly so, because they’re not compressed.

The light painter code processes BMPs to a temporary working file so they can be read back and issued to the DotStars very quickly. This requires some free space on the CIRCUITPY drive.

Therefore…any BMP images you’re not using, or other files like unrelated CircuitPython libraries you may have previously installed…move these off the CIRCUITPY drive to your computer for safe keeping. There’s no fixed limit (the software adapts to what’s available), but if possible aim for at least 600 kilobytes free. This is only true when using the light painter code, not a general rule for all situations.

Keep at least 600 KB free space on the CIRCUITPY drive.

Light Painter Hacks

Pixels don’t need to be square! While the DotStar strip length is fixed (e.g. 72px), the perpendicular axis is fluid…it’s all a matter of how fast you move the light stick while painting. We can exploit this to make slightly sharper images…

If your image editor allows freeform resizing of images (that is, the resulting aspect ratio doesn’t have to match the original), set only the minor axis of the image to the DotStar strip length, and keep the other axis at its original value. When light painting, the image can be squeezed back to its original aspect, but using the additional pixels for more detail on one axis:
Exception would be if the original image is truly enormous. Bigger images need more drive space, and we want to keep some available. A good upper limit is around 500 pixels, give or take a bit.

If the original image is already less than 500 pixels on the major axis, there’s no benefit to scaling it up! The light painter interpolates along this axis and will provide the best image it can with the data it’s given.

---

**Take Long Exposure Photos**

©Adafruit Industries
Now we come to the fun part of making art that can hover in space! You'll freeze temporal, sequential art into what appears to be a single instant! How is this done? The trick is long exposure photography.

Typically, we take photographs that only expose the sensor (or film) for a very tiny fraction of a second. This is a simplification that ignores many factors, but you can think of it as: the shorter the exposure (also called shutter speed) the sharper the image. This is because any subjects that are moving will create a blur if the exposure is long, since they will occupy more than one point in space during the time that light is exposing onto the sensor.

Additionally, if we use typical aperture sizes, or f-stops, (think of it as the size of the hole letting light pass onto the shutter) we can let in a lot of light during that very quick exposure. If the shutter is open too long, then too much light will hit the sensor and the shot will be over exposed.

Long exposure photography flips these conventions on their head! We'll use very long exposures -- anywhere from 4 to 30 seconds for our Light Paintstick images -- so that our subject (the NeoPixels) will occupy many different points in space during the time that the shutter is open. But, to avoid over exposing the sensor and creating a blindingly bright image with no details, we'll use a very small aperture. This means that only very bright objects (such as our NeoPixel LEDs) will send enough light to the sensor to be exposed on the final image.
The size of the aperture is expressed in f-stop values. It can be confusing at first because the lower the f-stop number the larger the opening. This is because the f-number is a ratio of focal length to aperture diameter.

Typical settings for "normal" daylight photography are fast shutters -- 1/250 second for example, and wide open apertures -- f/5.6 for example. Long exposure photographs taken in dark settings will use slow shutter speeds such as 4" (seconds) to 30" and small apertures such as f/22.

Tools

Ideally, you will want to use a good camera with manual control over the settings, mounted on a tripod. Any mirrorless system, DSLR, or higher-end point-and-shoot should give you the control you need. The camera will need to allow you to shoot either long exposures or in "bulb" mode where the shutter stays open indefinitely until you release it.

Alternately, you can use a smart phone and dedicated apps. Search for the terms "long exposure" and "light trails" to find some options.

Don't forget that the HalloWing has an ON/OFF switch!

Action

Now, you get to start experimenting! Start off simply, with a rainbow pattern. In a dark environment, set up your camera, trigger the shutter, get in front of the lens, turn on your Light Paintstick, and sweep an arc shape over your head.

Release the shutter and check out your photo! You can now start to tune the settings to dial things in.

Next, try some longer exposures and run around with your Light Paintstick. Get creative! It's also fun to have some context in you photos, so try tuning the exposure settings on your camera so that some of your environment is visible, not just LED streaks against black.
Floating Images

Now, you can try stamping an image into midair! Switch the CircuitPython code `LOOP = True` to `LOOP = False` and re-save the code.py file onto the board so that you're displaying one of the individual bitmaps, such as the pumpkin.

You will want to tune the speed of the play back so that the image draws in about 3-4 seconds. Set your camera for a 5-6 second exposure. Trigger the shutter and then move the Light Paintstick in a straight line parallel to the camera.
You can draw logos in midair, too!
Bats!

Multiple Stamps

You can trigger your image multiple times during a single exposure, just try to not overlap! Also notice that you can create a "backwards" image by moving the want from left-to-right instead of right to left as with the bottom pumpkin shown here.
You can also switch the code back to looping `LOOP = True` so that as long as you move the wand the image will repeat.
Here's an example of this on a playground merry-go-round.

If your image is squashed, try turning the potentiometer to the right a bit so that it draws the raster a bit slower. If the image is too wide, turn the pot to the left to speed things up. You'll want to still move the wand at the same speed so that you only adjust a single variable at a time.
Here's an example of the same image being played back at different speeds by tuning the potentiometer between takes.

Rotate your arm in a big circle!

Have fun with your light painting! You can even start to get fancy and include yourself in the photos -- just draw your images as usual in the air, and then at the end, hold very still and point your Light Paintbrush at your own face for a few seconds to add it to the exposed portion of the frame!
So, have fun experimenting with different artwork and techniques as you explore the fascinating art of long exposure light painting with your HalloWing Light Paintstick!

Note, you can create some creepy outtakes while you’re at it.

Here's the ghost of photographer Joel looking highly weird. This is what happens when you test the long exposure + flash theory. NOTE: He is not wearing a mask.
CircuitPython Magic
The goal of this project was to make a small-scale light painter using CircuitPython on a mid-range microcontroller board, with the level of image quality we achieved with the DotStar Pi Painter (not blocky NES-like pixel graphics, but subtle smooth colors. This was, putting it lightly, “a challenge.” The CLUE board has just a small fraction of the Raspberry Pi’s speed or RAM...and then to do this all in CircuitPython, no C code as in the Pi project, it just seemed impossible.

I’d always viewed Python as “simple” and “maintainable,” but rarely “fast”... and early versions of the painter were indeed awful. Thankfully, lady ada (), Scott Shawcroft () and Jeff Epler () — without judgment at my repeated failed attempts — all shared some of their “skateboard tricks” for improving parts of the Python code. In the end, it actually needed a slight delay added for best effect!

How it Works

The brightness range from addressable LED strips like DotStars and NeoPixels isn’t perceptually linear. Getting images to “look right” requires a reduction in the mid-range values. But...with only 256 possible values...this results in colors being grouped (“quantized”) into fewer “bins.” It’s all explained in this guide ()

Dithering is a common workaround for this, using alternating just-slightly brighter and darker values in rapid succession to simulate intermediate tones. This works okay to a point, but the way it’s typically handled, with pixels being square (as on most screens), it still leaves a blocky pixel residue, a la early Macintosh graphics.

Interesting thing about light painting is that there isn’t the traditional two-dimensional fixed “X” and “Y” axes. The pixels along the strip are one axis, sure...but the perpendicular axis is time, and the resolution of that axis is a function of how quickly the light bar moves in physical space, as seen through the camera. Pixels need not be square.

The CLUE light painter, and the DotStar Pi Painter before it, rely on DotStar LEDs being really fast to update...the half-meter strip can refresh about 1,000 times a second. What these programs do then is stretch an image along the time axis. Each second equals about 1,000 rows, and the quantization and dithering is performed in that much higher-resolution space. But, moved slowly across the camera sensor, those 1,000 rows are squeezed back to the original image’s size (or close to it, is the goal), and much of that normally-lost detail is recovered as the dithered bits blend together. The software also interpolates between rows to further reduce pixelation effects. The resulting images don’t look like your typical digital light paintings at all... they’re super buttery!
As you can imagine, that involves a ton of math, and with only a few grams of microcontroller to work with, things went very poorly at first.

Skateboard Tricks

First insight came from Ladyada, who was adamant in using DotStar LEDs for this, not NeoPixels. A couple early attempts used NeoPixels and the images were really blocky. I figured, it’s a short strip, no big deal? But it is a big deal. Partly this has to do with the transfer rate of the LED strips: the speed at which they can receive data. With NeoPixel strips, this is always fixed at 800 kilobits/second. And the CPU is completely tied up during that transfer… everything else stops. DotStars can receive at ten times this rate, 8 MHz, and one perk of the CLUE’s nRF52840 processor is that we get SPI DMA “free” — that transfer doesn’t take time away from other Python duties. So…while NeoPixels are cheaper and easier to wire up, this is one task where DotStars really shine.

Math Shenanigans

Ladyada was also leaning on me to use the ulab library (pronounced “micro lab”), to showcase its inclusion in recent CircuitPython builds. ulab is a numerical processing tool, performing operations on whole lists and tables of numbers much more quickly than iterating through Python loops. Jeff, who was pivotal in getting ulab into CircuitPython, [has written a guide on its use](https://www.adafruit.comdocumento) and assisted in this part of the light painter code.
ulab was the secret ingredient to that buttery interpolation and dithering! It seemed like the wrong tool for this at first, especially for the dithering...but really just required a new understanding of the problem, thinking about rows of pixels as a whole thing, not discrete elements. You can see this in the `process()` function in `bmp2led.py`. Even with all the comments it's still kind of "weird code."

**RAM Shenanigans**

Automatic garbage collection is one of the blessings of a language like Python, allowing for quicker development. There’s no need to track every little object allocation. Unused objects are periodically cleaned up by the system when space is needed, behind the scenes.

In 99% of applications, this is automagic and unseen...for instance, a weather application might poll a website and update a screen a few times a day, and a momentary pause to clean up has no perceptible impact on this.

Light painting is in that peculiar 1% though, because it’s reliant on physical things. A camera’s shutter, the movement of a stick. Pausing for garbage collection in the middle of a photo will just wreck that photo, there’s no recourse. And the way I’d initially written this, it was garbage-collecting a lot. Scott and Jeff had insights here, that with a bit of forethought, sections of code can be written to never need temporary allocations or garbage collection. Light painting is esoteric, but the idea might have implications for other time-critical code, like games.

Here’s the seemingly innocuous, but actually disastrous, inner-most loop as originally written:

```python
while True:
    action_set = {self.button_left.action(),
                  self.button_right.action()}
    if RichButton.TAP in action_set:
        painting = not painting # Toggle paint mode on/off
    elif RichButton.HOLD in action_set:
        return # Exit painting, enter config mode
# Code continues here...
```

Here it’s testing for “taps” and “holds” on both the left and right buttons, by putting both button action values into a set. This is cool in that it checks both buttons (or potentially any number) with a single Python “in” statement...no if-or-or-or construct is needed.

The fatal flaw is the `action_set =` line. This allocates a new set object, and any old value is set aside for later garbage collection. It’s a very tiny allocation, just a few
bytes...but this loop runs about 1,000 times a second, and very quickly even the CLUE’s capacious memory was filled. Every single photograph was getting glitched as the garbage collector ran; there weren’t even any lucky shots.

The fix was small and simple: avoid the allocation inside the loop. Rather than a set, create a small known-size list before getting into the loop. Once in the loop, set existing elements of the list. Requires one extra line of code, but all that allocation and cleanup goes away, and the timing became super uniform and glitch-free:

```python
action_list = [None, None]
while True:
    action_list[0] = self.button_left.action()
    action_list[1] = self.button_right.action()
    if RichButton.TAP in action_list:
        painting = not painting # Toggle paint mode on/off
    elif RichButton.HOLD in action_list:
        break # End paint loop
    # Code continues here...
```

Little things like that. And we still get to use the cool “in” syntax. The two distinct assignments would probably get pooh-poohed as “not Pythonic,” but oh well, here we are.

File Shenanigans

A related problem existed in the code that was reading the BMP images, and then later when reading the temporary (processed) file to feed that data to the DotStar LEDs. The normal `file.read()` function allocates and returns a buffer each time it’s called:

```python
while (condition):
    led_data = file.read(length_in_bytes)
    spi.write(led_data)
```

It’s quicker and easier to write this way (the read and write could even be expressed on a single line), but results in a fresh `led_data` being allocated on every pass of the loop, leading to frequent garbage collection.

Fix is similar to the button situation: move the allocation outside the loop, then use `file.readinto()` to keep re-using the same memory.

```python
led_data = bytearray(length_in_bytes)
while (condition):
    file.readinto(led_data)
    spi.write(led_data)
```
Boom! Massive speedup. This isn’t a solution to every problem, but it works in this case because the size of the `led_data` buffer is consistent from call to call…we have that luxury then of only allocating it once.

A different and unrelated problem involved the LED temporary file, where an image is “stretched” and converted to a DotStar-ready format. It turns out that appending to a file can get really, really slow if you do it a lot.

```python
with open(output_filename, 'wb') as led_file:
    while (condition):
        led_file.write(output_buffer)
```

It was doing this on every single row…on an image that might get stretched to 1,000 rows or more!

Like the `readinto()` fix, the workaround here exploits the fact that the resulting file will be a known fixed size, something we can calculate ahead of time. We can then use `file.seek()` and write a single byte at the end to very quickly create a huge file full of nothing…then go back to the start and fill in the data, much faster now because it’s not appending:

```python
with open(output_filename, 'wb') as led_file:
    led_file.seek((output_buffer_size * rows) - 1)
    led_file.write(b'\0')
    led_file.seek(0)
    while (condition):
        led_file.write(output_buffer)
```

This change alone doubled the speed of the image conversion!