Circuit Playground or Hallowing Jack-o'-Lantern

Created by Phillip Burgess

https://learn.adafruit.com/circuit-playground-jack-o-lantern

Last updated on 2021-11-15 06:48:11 PM EST

Table of Contents

Halloween!

Claws-down, Halloween is my favorite holiday. Add some electronics to the mix and it's an unstoppable force of nature.

In this simple no-soldering-required project we'll make the Adafruit Circuit Playground Classic or Circuit Playground Express board act like a flickering candle. Pop this inside a jack-o'-lantern (or any other item that might normally use flame) to create a festive spooky atmosphere. (This can also work with an Adafruit Hallowing board with a NeoPixel strip plugged in the NEOPIX port.)

Sure, one can just buy a fake battery-operated candle, but they're usually pretty dim. Also, we can take this as an opportunity to learn a bit about code...and once Halloween's over (how sad!) the Circuit Playground board can be used again in endless other projects.

Items needed:

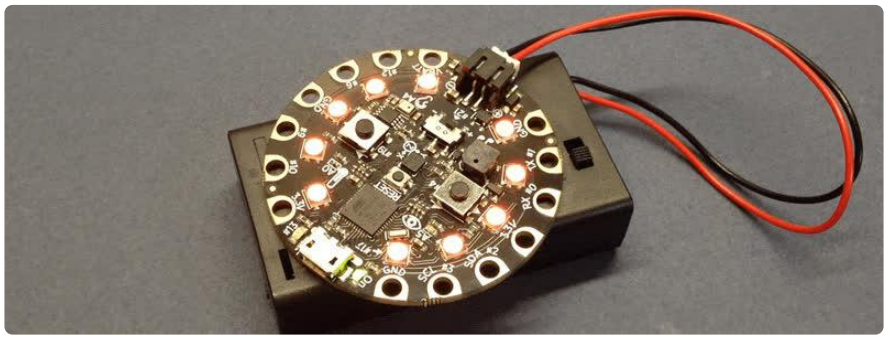
- Adafruit Circuit Playground Classic (https://adafru.it/ncE) or Circuit Playground Express (https://adafru.it/wpF) board.
- Either a 3xAAA battery holder (http://adafru.it/727) -or- a USB phone charger (http://adafru.it/1959) & cable
- Pumpkin and carving tools, or other decorative item to illuminate (very effective with paper lanterns!)
- Zip-Loc bag, if using a real pumpkin

If this is your first time using Circuit Playground, start with our "Introducing Circuit Playground (https://adafru.it/ncG)" or "Circuit Playground Express (https://adafru.it/adafruit-cpx)" guide. There's some software to set up on your computer and some procedures to learn. Try out a couple of the examples, make sure you know how to upload "sketches" (code) to the board.

Circuit Playground Express can be programmed three different ways. You only need to pick ONE of these — whichever you're most comfortable with or looks interesting.

Powering the Project

For stand-alone powering the project, our 3xAAA battery holder has a power switch and a JST connector that fits the Circuit Playground board. A fresh set of batteries should run it for about ten hours, maybe even longer. Alternately, a USB phone charger (and micro-B cable) can be used.



MakeCode

MakeCode for Simulated Candle

Another option for Circuit Playground Express boards (not Classic) is MakeCode, a "block based" visual programming environment.

The basics of using MakeCode are explained in the Circuit Playground Express (https://adafru.it/AQW) guide (https://adafru.it/AQW).

CircuitPython Code

CircuitPython Code for Simulated Candle

Circuit Python Express boards can run CircuitPython — a different approach to programming compared to Arduino sketches. If you want to learn the basics of setting up and using CircuitPython, this is explained in the Circuit Playground Express CircuitPython guide (https://adafru.it/AFI).

This is for Express boards only; Classic boards can't run CircuitPython.

Below is CircuitPython code that works similarly to the Arduino sketch shown elsewhere in this guide. To use this, plug Circuit Playground Express into USB...it should show up on your computer as a small flash drive...then edit the file code.py (or main.py) with your text editor of choice. Select and copy the code below and paste it into that file, entirely replacing its contents (don't mix it in with lingering bits of old code). When you save the file, the code should start running almost immediately (if not, see notes at the bottom of this page).

If Circuit Playground Express doesn't show up as a drive, follow the guide link above to prepare the board for CircuitPython.

```
"""Jack-o'-Lantern flame example Adafruit Circuit Playground Express"""

import math
import board
import neopixel
try:
    import urandom as random  # for v1.0 API support
except ImportError:
    import random

NUMPIX = 10         # Number of NeoPixels
PIXPIN = board.D8   # Pin where NeoPixels are connected
STRIP = neopixel.NeoPixel(PIXPIN, NUMPIX, brightness=1.0)
PREV = 128

def split(first, second, offset):
    """
    Subdivide a brightness range, introducing a random offset in middle,
    then call recursively with smaller offsets along the way.
    @param1 first:  Initial brightness value.
    @param1 second: Ending brightness value.
    @param1 offset: Midpoint offset range is +/- this amount max.
```

```
    """
    if offset != 0:
        mid = ((first + second + 1) / 2 + random.randint(-offset, offset))
        offset = int(offset / 2)
        split(first, mid, offset)
        split(mid, second, offset)
    else:
        level = math.pow(first / 255.0, 2.7) * 255.0 + 0.5
        STRIP.fill((int(level), int(level / 8), int(level / 48)))
        STRIP.write()

while True:  # Loop forever...
    LVL = random.randint(64, 191)
    split(PREV, LVL, 32)
    PREV = LVL
```

This code can also work on an Adafruit HalloWing board with just a small change to lines 10 and 11:

```
NUMPIX = 30                       # Number of NeoPixels
PIXPIN = board.EXTERNAL_NEOPIXEL  # Pin where NeoPixels are connected
```

Arduino Code

Arduino Code for Simulated Candle

This is the code you'll use for Circuit Playground Classic. If you have a Circuit Playground Express, you can use this code as well, but also have the option of using MakeCode and CircuitPython as shown on prior pages.

Copy and paste the code below into a new Arduino sketch:

```
// Jack-o-Lantern sketch for Adafruit Circuit Playground (Classic or Express)

#include "Adafruit_CircuitPlayground.h"

void setup() {
  CircuitPlayground.begin();
  CircuitPlayground.setBrightness(255); // LEDs full blast!
}

uint8_t prev = 128;               // Start brightness in middle

void loop() {
  uint8_t lvl = random(64, 192); // End brightness at 128±64
  split(prev, lvl, 32);          // Start subdividing, ±32 at midpoint
  prev = lvl;                    // Assign end brightness to next start
}

void split(uint8_t y1, uint8_t y2, uint8_t offset) {
  if(offset) { // Split further into sub-segments w/midpoint at ±offset
    uint8_t mid = (y1 + y2 + 1) / 2 + random(-offset, offset);
    split(y1 , mid, offset / 2); // First segment (offset is halved)
    split(mid, y2 , offset / 2); // Second segment (ditto)
  } else { // No further subdivision - y1 determines LED brightness
```

```
    uint32_t c = (((int)(pow((float)y1 / 255.0, 2.7) * 255.0 + 0.5) // Gamma
                 * 0x1004004) >> 8) & 0xFF3F03; // Expand to 32-bit RGB color
    for(uint8_t i=0; i<10; i++) CircuitPlayground.strip.setPixelColor(i, c);
    CircuitPlayground.strip.show();
    delay(4);
  }
}
```

From the “Tools” menu, select “Board→Adafruit Circuit Playground” (or Circuit Playground Express). Also make sure the correct serial port is selected, then upload the sketch to the board.

If all goes well, the NeoPixels on the Circuit Playground board should start flickering (mostly shades of yellow). There’s our electronic “candle”!

If you have an Adafruit Hallowing board with a NeoPixel strip plugged into the NEOPIX port, you can use this version of the code instead:

```
// Jack-o-Lantern sketch for Adafruit Hallowing

#include "Adafruit_NeoPixel.h"

#define NUM_PIXELS 30

Adafruit_NeoPixel strip(NUM_PIXELS, 4, NEO_GRB + NEO_KHZ800);

void setup() {
  strip.begin();
}

uint8_t prev = 128;              // Start brightness in middle

void loop() {
  uint8_t lvl = random(64, 192); // End brightness at 128±64
  split(prev, lvl, 32);          // Start subdividing, ±32 at midpoint
  prev = lvl;                    // Assign end brightness to next start
}

void split(uint8_t y1, uint8_t y2, uint8_t offset) {
  if(offset) { // Split further into sub-segments w/midpoint at ±offset
    uint8_t mid = (y1 + y2 + 1) / 2 + random(-offset, offset);
    split(y1 , mid, offset / 2); // First segment (offset is halved)
    split(mid, y2 , offset / 2); // Second segment (ditto)
  } else { // No further subdivision - y1 determines LED brightness
    uint32_t c = (((int)(pow((float)y1 / 255.0, 2.7) * 255.0 + 0.5) // Gamma
                 * 0x1004004) >> 8) & 0xFF3F03; // Expand to 32-bit RGB color
    for(uint8_t i=0; i<NUM_PIXELS; i++) strip.setPixelColor(i, c);
    strip.show();
    delay(4);
  }
}
```

Pumpkins!

Now we need something to light up. Tradtionally this has been a carved pumpkin, but it doesn’t have to be. If you’ve got a 3D printer, how about this amazing Voronoi Skull

(https://adafru.it/rVc)? With our “candle” inside it should cast some interesting shadows!

If going the carved-pumpkin route, you’ll need a design...either your own, or an internet search for “pumpkin stencil” turns up thousands of options, some free, some asking a small charge. Here’s a couple free ones I made that might get you started...

Dragon_Stencil.pdf

https://adafru.it/rUb

Werewolf_Stencil.pdf

https://adafru.it/rUc

Both of these are designed with the “shallow carve” technique in mind, where one scoops away at the surface but doesn’t cut all the way through the pumpkin...they may work well enough for the latter, though you’d lose some smaller details like the dragon’s nostril or the werewolf’s slobber.

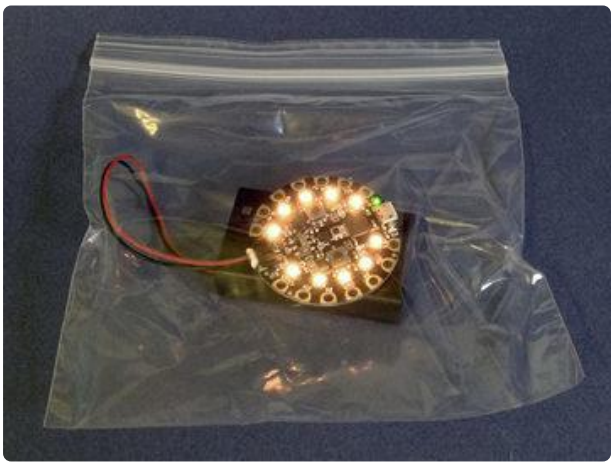
I tried out these foam craft pumpkins WHICH TURNED OUT TO BE A HORRIBLE IDEA. Fine for painting or for cut-through stencil designs, but nearly impossible to shallow carve! Real gourds are best for that.

PRO TIP: print out your design first and take it with you when selecting a pumpkin. This ensures the size and shape will work with the design...you might even find a gourd with one slightly flatter side for the art.

Various techniques can be used to transfer the flat art to the curved pumpkin...one could just “eyeball” it and redraw, or use a projector if available. Here I’ve scribbled on the back of the pattern with a soft pencil (6B), taped it down, then traced on the front with a ball-point pen to transfer some of the graphite to the pumpkin surface...this requires a bit of cleanup and re-interpreting bits of the art lost to creases.

I also learned that white pumpkins are a poor choice for shallow-carving. They look great at night with a light inside...but in daylight, the design just vanishes!

Adding some black paint salvaged the “day look” of the white pumpkin, making the design “pop.” I liked it so much, the orange pumpkin (which didn’t really need it) got the same treatment!



Using a natural pumpkin? Place the Circuit Playground and battery pack inside a Zip-Loc sandwich bag to keep them clean.

How Does it Work?

The simulated flame code isn't very sophisticated or even technically accurate, yet does a good enough job of fooling our eyes. How can it do that? Fractals!

Fractals are mathematical objects exhibiting self-similarity at the macro and micro scales...any small part resembles the whole...

Fractals are great for simulating natural phenomena!

Notice how each floret of this (real) romanesco broccoli resembles the full head...and each floret, in turn, has smaller florettes.

This notion of repetition across scales is used in computer simulations...for example, each peak and valley in this fractal terrain has several smaller sub-peaks, with progressively smaller mounds and nooks all the way down...but the same mathematical formula defines them at every level, only the scale changes.

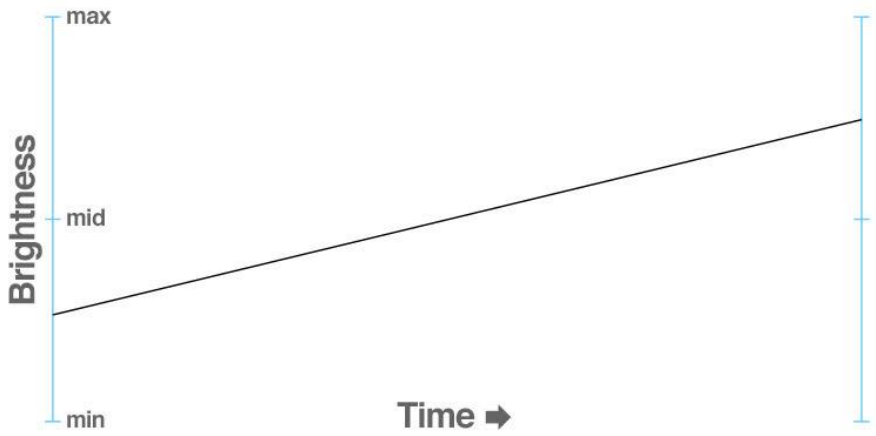
(Image credits: Wikimedia Commons public domain.)

Fire is another natural phenomenon where this math could help. The flame code works a little like the fractal terrain, but much simpler. Instead of three dimensions, it only has to work in two (time and brightness), and one of those...time...always moves linearly forward. So really the code just has to create those “hills” and “valleys” along the brightness axis.

In the sketch, brightness is represented with a uint8_t variable — an 8-bit value in the range 0 to 255 (darkest to lightest). Consider two values randomly offset from the middle (128) up or down by 64 (or less). Those will be the starting and ending brightness for some time interval. Now consider a straight line between them...

Divide the line at its midpoint into two equal segments, then move that midpoint up or down randomly by up to half the previous range; somewhere between -32 and +32.

Do the same on each of the two resulting segments: split in two, randomly move the midpoint of each up or down by 16. Repeat with the four resulting segments, ±8, then eight segments ±4 and so forth...



And that's all there is to a simple fractal! Assign those numbers over time to the brightness of an LED (or all the NeoPixels around the Circuit Playground board) and you have a not-too-shabby candle flicker effect.

To achieve that line segment split, and the split-split, and split-split-split-split and so forth, the code uses a techniqe called recursion.

Normally when programming you'll call some function (e.g. Serial.println()), it performs some operation and then returns, and the code resumes at the next statement.

Recursion is what happens when a function calls itself...

```
void foo() {
  foo();
}
```

Normally this would create a sort of infinite loop, but even worse.

Any time any C function is called, a tiny bit of RAM is used: a pointer back to the next statement upon return (called the stack pointer), plus space for any local variables used in that function. This RAM is freed when the function returns. If just keeps "nesting" calls and never returns though...the example above, despite being just a couple of bytes per function call...with only 2K RAM on most Arduino-type boards, it'll run out of space in no time and the program will soon crash.

To avoid this, some limit can be placed on the maximum depth of recursion...the number of times a function can call itself from within itself from within itself. This can be passed along as an argument...

```
void foo(int depth) {
  if(depth &lt; 10) {
    foo(depth + 1);
  }
}
```

foo() now requires four bytes per invocation (two for the stack pointer, two for the "depth" argument...a local variable). The first call to foo() would pass a value of 0. By checking the value of depth and limiting it to 10, this won't use more than 40 bytes worst case.

In the candle flame sketch, the split() function calls itself recursively, twice: one call handles the left or first half of the line segment, other handles the right or second half. Each of those, in turn, may call split() again, and so on...but only up to a point...

```
void split(uint8_t y1, uint8_t y2, uint8_t offset) {
  if(offset) { // Split further into sub-segments w/midpoint at ±offset
    uint8_t mid = (y1 + y2 + 1) / 2 + random(-offset, offset);
    split(y1 , mid, offset / 2); // First segment (offset is halved)
    split(mid, y2 , offset / 2); // Second segment (ditto)
  } else ...
  ...
}
```

"offset" is the maximum amount to move the center point up or down. Each time we call split() recursively, we divide that value by 2...because that's the nature of fractals ...big broccoli head = big offset, little broccoli florets = little offset. When offset reaches zero (which will happen due to integer rounding: 1÷2 = 0), no further subdivision/recursion takes place...a RAM disaster is averted and the LEDs are set to a brightness level corresponding to the "y1" argument before returning.

The code recurses and returns down and up and down and up like this until the final brightness level is reached, at which point this becomes the next starting value and a new random end value is chosen and the whole process is repeated.

The MakeCode version works a little differently, because this sort of recursion is not supported in "BLOCKS" mode. Instead the brightness level bounces back and forth semi-randomly...not quite the same, but still pretty effective.