Fake TV Light for Engineers
Created by Phillip Burgess
Guide Contents

Guide Contents ....................................................... 2
Overview .......................................................... 3
Crunching the Numbers ........................................... 5
Arduino Sketch .................................................... 8
Overview

This is a very minor and could-have-been-simplistic project, but makes for an amusing story about obsessiveness. Before getting into computers, I’d planned to follow in my dad’s footsteps in graphic design. The vestiges of this interest sometimes flare up at the oddest times...

A commercial product called FakeTV simulates the changing light cast by a television set, making a home appear occupied at night and thus (in theory) a less appealing target to burglars.

As FakeTV involves microcontrollers and RGB LEDs, it’s inevitable then that fake FakeTV-like projects have appeared on various DIY web sites.

Having a one-time need for such an item, and with scads of microcontroller boards and NeoPixels already on-hand, I gave a few of these DIY projects a try.

All suffered from extreme...well...fakeness. Some cycled predictably through a canned sequence of colors. Others switched among random RGB values. I don’t know if burglars would even notice such a thing, but it immediately struck me as “off.” Way off.

Movies are not random! Whether animated or live-action, they have Art Directors and Colorists and a specific color palette. Color sets the mood and defines locales and characters and is very un-random.

For example, the cinematic gem Thunderpants focuses almost exclusively on shades of green and brown.
The *FakeTV* web site claims their product — unlike DIY clones — began by analyzing actual TV programming, then developing an algorithm that realistically approximates these sorts of color changes mathematically.

Pretty cool idea, but I lacked both the time and inclination to develop a TV simulation algorithm. An idea struck me though...microcontrollers have a fair amount of flash memory...suppose we just stored color palettes of some actual art-directed scenes from actual movies? We don't need to store whole image frames, just the average screen color over time.

Okay then...but how to acquire this kind of data? I was still in a hurry and didn't want to process tons of source video. Turns out there are folks who study just this sort of thing. This Washington Post article (https://adafruit.it/nyE) highlights the work of Dillon Baker in creating color timelines from films, and links to other sites with similar visualizations. “Scraping” these sites might be one option.

The *Disney Animated* (https://adafruit.it/le4) app for iPad includes color timelines for 54 Disney films. Mostly animated, but some with live action segments, others with more “natural” tones.

A couple of screen captures later and I had dozens of hours of source data to work with...

But wait a minute...if we’d simply used random RGB colors...is any would-be thief really gonna stop and say, “Hang on...no director would cut between those two colors...that’s a fake TV!” It no longer mattered at this point. The crust was broken, both the artist and engineer sides of the brain had been tickled. It’s an unstoppable force of nature. We’re doomed.

Some folks obsess over audio. I obsess over images.
Crunching the Numbers

The iPad retina screen capture contains over nine megabytes of data.

There’s 32 kilobytes of flash memory on an ATmega328P-based microcontroller board. Leaving room for playback code and bootloader, it’s more like a maximum of 26K or so for our data. There are other boards with more flash space, or could have used an SD card, but I wanted to keep it simple and use a Metro 328 board...a 40-pixel NeoPixel Shield would neatly fit atop it for a FakeTV-like form-factor...but it can work just as well with NeoPixel strip, if that’s what you’ve got.

To make the data fit, something will have to go. Several orders of magnitude...

Looking closely at the screen capture, you’ll see these aren’t simple frame averages. The app describes their use of “clusters” in the visualization. To change the clusters into averages we’ll do some image editing with Photoshop.

First, a small black band at the bottom of the screen was cropped off.

There are 54 films in the visualization, each with one horizontal band. To produce an average for each one, the image is simply resized to 54 pixels tall (keeping the same width) specifically using bilinear interpolation mode (which will average all the pixels in each band, unlike bicubic which assigns different pixel weights and will “bleed over” from adjacent rows).

Strictly for the sake of illustration here...not done to the source data...the image was re-stretched to the original size in nearest neighbor mode, so you can see the color stripes.

Scaling down the image vertically this way provides a massive 28X reduction. But the iPad screen is huge (2048 pixels across), and this still represents about 330 kilobytes of color data. A few more reductions are needed...
Culling the herd: Some color timelines aren’t as “lively” as others...they might dwell on one color for a very long time, or are just very dark overall...or too bright, or saturated, or desaturated. Three out of four timelines were discarded, keeping the 13 most visually interesting. These were selected manually.

Another 4X reduction. Now about 80K of data.

Representing each film in its entirety didn’t seem necessary...titles and end credits, for example, were usually very plain...so I cropped just a 1,000-pixel-wide section from the larger image. If we assume each film has about a 100-ish minute average running time, this represents a roughly 45-ish minute chunk from each. That’s about 2.7-ish seconds-ish per pixel-ish.

Slightly over 2X reduction, now at 39 kilobytes. Here’s the resulting image:

This is still too large to fit in the Metro’s flash space, but one more reduction is going to take place...

A Python script will process this image into an Arduino header file. Along the way, this will quantize the 24-bit color data down to 16-bits per pixel (5 bits red, 6 bits green, 5 bits blue)...a 2/3 reduction, or 26 kilobytes total.

Though we lose some color fidelity, these are the least-significant bits...and there are plenty of other factors (LED color balance, diffuse interreflection with a room’s walls and other objects) that will tinge the resulting colors anyway. The playback code will do some shenanigans that should recover some intermediary shades.

Here’s the Python script that converts the image (named crop.png) to a .h file. It requires the Python Imaging Library or Pillow:
from PIL import Image
import sys

# Output one hex byte with C formatting & line wrap ------------------------

numBytes = 0  # Total bytes to be output in table
byteNum = 0  # Current byte number (0 to numBytes-1)
cols = 12 # Current column number in output (force indent on first one)

def writeByte(n):
    global cols, byteNum, numBytes

    cols += 1                      # Increment column #
    if cols >= 12:                 # If max column exceeded...
        print                  # end current line
        sys.stdout.write("  ") # and start new one
        cols = 0               # Reset counter
        byteNum += 1
        if byteNum < numBytes:
            sys.stdout.write(",")
        if cols < 11:
            sys.stdout.write(" ")

# Mainline code -----------------------------------------------------------

# Output 8-bit gamma-correction table:
sys.stdout.write("const uint8_t PROGMEM gamma8[] = {\n")
numBytes = 256
for i in range(256):
    base = 1 + (i / 3)  # LCD, CRT contrast is never pure black
    overhead = 255.0 - base
    writeByte(base + int(pow(i / 255.0, 2.7) * overhead + 0.5))
print "};",

# Output color data (2 bytes per pixel):
sys.stdout.write("colors[] = {\n")
image = Image.open("crop.png")
image.pixels = image.load()
umBytes = image.size[0] * image.size[1] * 2
byteNum = 0
cols = 12
for y in range(image.size[1]):
    for x in range(image.size[0]):
        r = image.pixels[x, y][0]
        g = image.pixels[x, y][1]
        b = image.pixels[x, y][2]
        # Convert 8/8/8 (24-bit) RGB to 5/6/5 (16-bit):
        writeByte((r & 0xF8) | (g >> 5))
        writeByte(((g & 0x1C) << 3) | (b >> 3))
print "};"

The output of this program is redirected to a file (e.g. data.h) and can then be #included by the Arduino code on the next page...
Arduino Sketch

In addition to the code below, you'll need the output from the Python program on the prior page (or we also include it later on this page).

```cpp
#include <Adafruit_NeoPixel.h>
#include "data.h" // Output of Python script

#define NUM_LEDS 40
#define PIN 6
Adafruit_NeoPixel strip = Adafruit_NeoPixel(NUM_LEDS, PIN, NEO_GRB);

#define numPixels (sizeof(colors) / sizeof(colors[0]))

uint32_t pixelNum;

#define pr = 0, pg = 0, pb = 0; // Prev R, G, B

void setup() {
  strip.begin();
  randomSeed(analogRead(A0));
  pixelNum = random(numPixels); // Begin at random point
}

void loop() {
  uint32_t totalTime, fadeTime, holdTime, startTime, elapsed;
  uint16_t nr, ng, nb, r, g, b, i;
  uint8_t hi, lo, r8, g8, b8, frac;

  // Read next 16-bit (5/6/5) color
  hi = pgm_read_byte(&colors[pixelNum * 2    ]);
  lo = pgm_read_byte(&colors[pixelNum * 2 + 1]);
  if(++pixelNum >= numPixels) pixelNum = 0;

  // Expand to 24-bit (8/8/8)
  r8 = (hi & 0xF8) | (hi >> 5);
  g8 = (hi << 5) | ((lo & 0xE0) >> 3) | ((hi & 0x06) >> 1);
  b8 = (lo << 3) | ((lo & 0x1F) >> 2);

  // Apply gamma correction, further expand to 16/16/16
  nr = (uint8_t)pgm_read_byte(&gamma8[r8]) * 257; // New R/G/B
  ng = (uint8_t)pgm_read_byte(&gamma8[g8]) * 257;
  nb = (uint8_t)pgm_read_byte(&gamma8[b8]) * 257;

  totalTime = random(250, 2500); // Semi-random pixel-to-pixel time
  fadeTime = random(0, totalTime); // Pixel-to-pixel transition time
  if(random(10) < 3) fadeTime = 0; // Force scene cut 30% of time
  holdTime = totalTime - fadeTime; // Non-transition time

  startTime = millis();
  for(;;) {
    elapsed = millis() - startTime;
    if(elapsed >= fadeTime) elapsed = fadeTime;
    if(fadeTime) {
      r = map(elapsed, 0, fadeTime, pr, nr); // 16-bit interp
      g = map(elapsed, 0, fadeTime, pg, ng);
      b = map(elapsed, 0, fadeTime, pb, nb);
    } else { // Avoid divide-by-zero in map()
      r = nr;
      g = ng;
      b = nb;
    }
  }
} // void loop()
```
if Python is unavailable on your system, or if it’s just easier this way, here are the entire gamma and color tables.

Create a new tab in the Arduino sketch, name it “data.h”, then cut and paste this whole thing:

```cpp
const uint8_t PROGMEM gamma8[] = {
  0X01, 0X01, 0X01, 0X02, 0X02, 0X03, 0X03, 0X04, 0X04, 0X04, 0X04, 0X05, 0X05, 0X05, 0X06, 0X06, 0X07, 0X07, 0X07, 0X08, 0X08, 0X08, 0X09, 0X09, 0X0A, 0X0B, 0X0B, 0X0C, 0X0C, 0X0D, 0X0D, 0X0E, 0X0E, 0X0F, 0X10, 0X10, 0X11, 0X11, 0X12, 0X12, 0X13, 0X14, 0X14, 0X15, 0X15, 0X16, 0X17, 0X17, 0X18, 0X18, 0X19, 0X1A, 0X1A, 0X1A, 0X1B, 0X1B, 0X1C, 0X1C, 0X1D, 0X1D, 0X1E, 0X1E, 0X1F, 0X1F, 0X20, 0X21, 0X21, 0X22, 0X23, 0X24, 0X25, 0X25, 0X26, 0X27, 0X28, 0X29, 0X2A, 0X2A, 0X2B, 0X2C, 0X2C, 0X2D, 0X2D, 0X2F, 0X2F, 0X30, 0X31, 0X31, 0X32, 0X33, 0X34, 0X34, 0X36, 0X36, 0X37, 0X38, 0X38, 0X39, 0X3A, 0X3B, 0X3C, 0X3D, 0X3E, 0X3E, 0X40, 0X40, 0X41, 0X42, 0X43, 0X43, 0X44, 0X45, 0X46, 0X46, 0X48, 0X49, 0X4B, 0X4B, 0X4C, 0X4E, 0X4E, 0X4F, 0X50, 0X51, 0X52, 0X54, 0X54, 0X55, 0X57, 0X57, 0X58, 0X5A, 0X5B, 0X5B, 0X5D, 0X5E, 0X5F, 0X60, 0X61, 0X62, 0X64, 0X65, 0X65, 0X67, 0X68, 0X69,
};
```

Points of interest in the Arduino sketch:

- The colors are from actual films, but the timing is semi-random…the goal isn’t to match specific scene tempos, just needed believable color sequences. Some transitions are abrupt cuts, others fade (implying a camera pan or something moving in or out of frame).
- Color interpolation takes place in 16-bit space…LEDs are only 8-bit, but some fraction of the LEDs is used to get in-between shades.
- The NeoPixel library periodically disables interrupts and is known to mess with timekeeping functions like millis(). That’s okay for this application…we’re not precisely beat-matching any source material, just measuring relative time.

Upload sketch to a Metro board (or Arduino Uno or compatible) with NeoPixel shield installed. Power from a quality USB supply. This can then be plugged into a basic lamp timer. The sketch starts at a random point in the color data, so it’s not repeating the same sequence every time it powers up.

Aim it at the ceiling to wash a room in color, similar to the glow from a TV, or directly toward curtains if you need more brightness.

If Python is unavailable on your system, or if it’s just easier this way, here are the entire gamma and color tables.

Create a new tab in the Arduino sketch, name it “data.h”, then cut and paste this whole thing:
colors = {
0X6B, 0x6C, 0x6E, 0x6F, 0x70, 0x72, 0x73, 0x74, 0x75, 0x76, 0x77, 0x79, 0x7A, 0x7B, 0x7D, 0x7E, 0x7F, 0x81, 0x82, 0x83, 0x85, 0x86, 0x87, 0x89, 0x8A, 0x8B, 0x8D, 0x8E, 0x8F, 0x91, 0x92, 0x93, 0x95, 0x96, 0x98, 0x99, 0x9B, 0x9C, 0x9E, 0x9F, 0xA0, 0xA2, 0xA3, 0xA5, 0xA6, 0xA8, 0xA9, 0x9B, 0xA2, 0xA4, 0xA5, 0xB1, 0xB2, 0xB4, 0xB5, 0xB7, 0xB9, 0xBA, 0xBB, 0xBD, 0xBF, 0xC0, 0xC2, 0xC3, 0xC5, 0xCF, 0xC0, 0xC2, 0xCD, 0xCE, 0xEF, 0xF1, 0xF4, 0xF6, 0xF8, 0xFA, 0xFB, 0xFD, 0xFF,
}
© Adafruit Industries

https://learn.adafruit.com/fake-tv-light-for-engineers

Page 11 of 48
© Adafruit Industries

https://learn.adafruit.com/fake-tv-light-for-engineers
data.h

```c
#define

0x20, 0x21, 0x29, 0x31, 0x32, 0x39, 0x3a, 0x3b, 0x3c, 0x3d, 0x3e, 0x3f, 0x40, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48, 0x49, 0x4a, 0x4b, 0x4c, 0x4d, 0x4e, 0x4f, 0x50, 0x51, 0x52, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58, 0x59, 0x5a, 0x5b, 0x5c, 0x5d, 0x5e, 0x5f, 0x60, 0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x6a, 0x6b, 0x6c, 0x6d, 0x6e, 0x6f, 0x70, 0x71, 0x72, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78, 0x79, 0x7a, 0x7b, 0x7c, 0x7d, 0x7e, 0x7f, 0x80, 0x81, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87, 0x88, 0x89, 0x8a, 0x8b, 0x8c, 0x8d, 0x8e, 0x8f, 0x90, 0x91, 0x92, 0x93, 0x94, 0x95, 0x96, 0x97, 0x98, 0x99, 0x9a, 0x9b, 0x9c, 0x9d, 0x9e, 0x9f, 0xa0, 0xa1, 0xa2, 0xa3, 0xa4, 0xa5, 0xa6, 0xa7, 0xa8, 0xa9, 0xaa, 0xab, 0xac, 0xad, 0xae, 0xaf, 0xb0, 0xb1, 0xb2, 0xb3, 0xb4, 0xb5, 0xb6, 0xb7, 0xb8, 0xb9, 0xba, 0xbb, 0xbc, 0xbd, 0xbe, 0xbf, 0xc0, 0xc1, 0xc2, 0xc3, 0xc4, 0xc5, 0xc6, 0xc7, 0xc8, 0xc9, 0xca, 0xcb, 0xcc, 0xcd, 0xce, 0xcf, 0xd0, 0xd1, 0xd2, 0xd3, 0xd4, 0xd5, 0xd6, 0xd7, 0xd8, 0xd9, 0xda, 0xdb, 0xdc, 0xdd, 0xde, 0xdf, 0xe0, 0xe1, 0xe2, 0xe3, 0xe4, 0xe5, 0xe6, 0xe7, 0xe8, 0xe9, 0xea, 0xeb, 0xec, 0xed, 0xee, 0xef, 0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7, 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff;
```

© Adafruit Industries

Last Updated: 2018-08-22 03:53:50 PM UTC