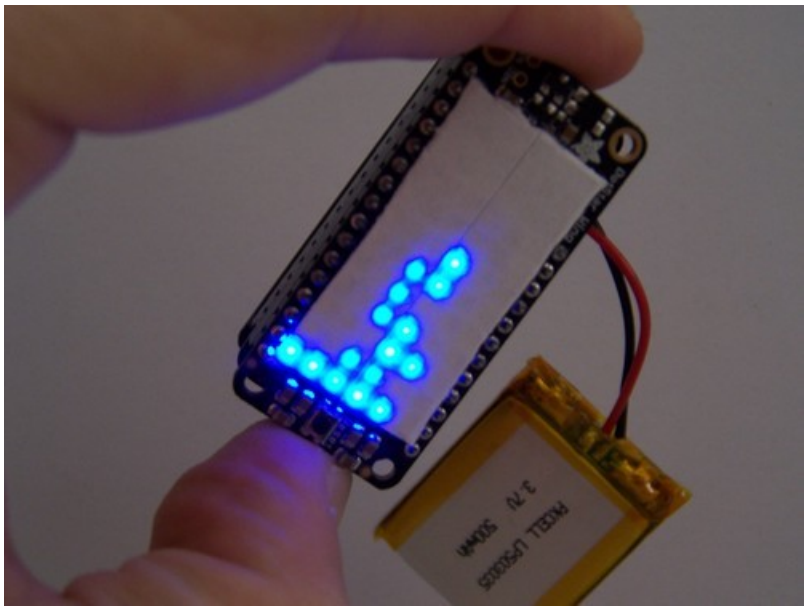
adafruit learning system

Dotstar + CircuitPython Digital Sand

Created by Dave Astels



Last updated on 2018-08-22 04:05:24 PM UTC

Guide Contents

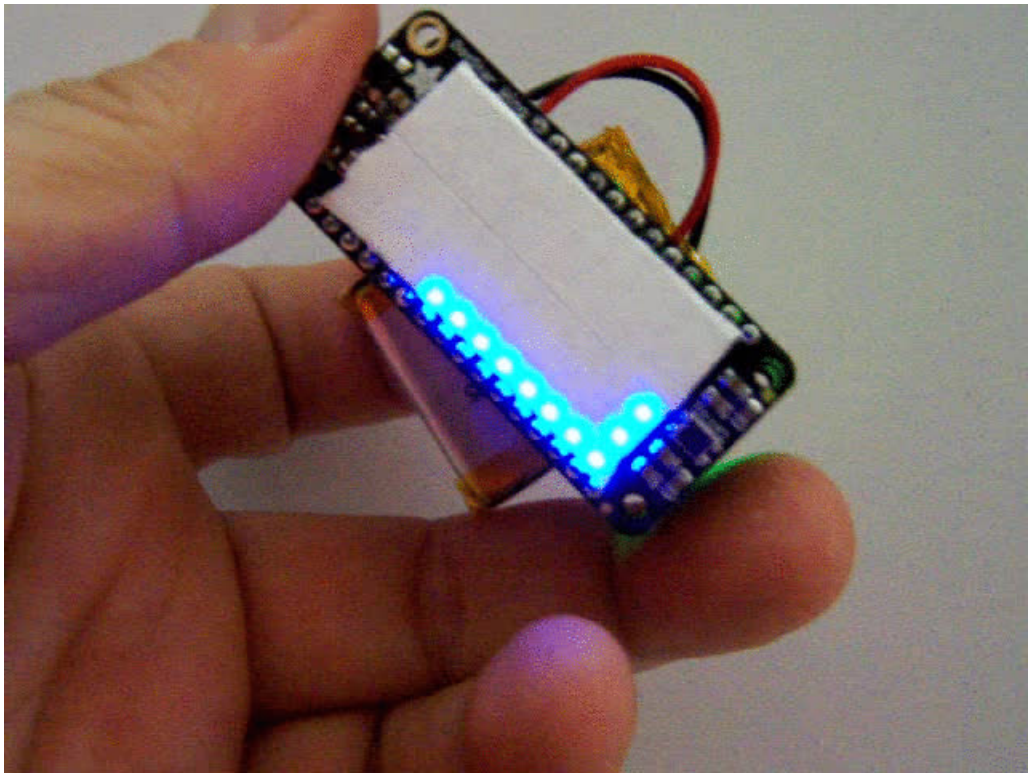
Overview

Something that has gotten popular recently is Phil Burgess' *digital sand* demo (https://adafru.it/Cjb). It's been riffed on a few times now, and since I have been playing around with the Dotstar Featherwing, I thought I'd have a go at it as well.

I really wanted to do this in CircuitPython, partly as a learning exercise since I'm still fairly new to (Circuit)Python, and partly to see how far it could be pushed on a rather constrained environment like the SAMD21G18.

The code is a fairly literal port of Phil's code as listed in the Ruiz Brothers' Animated LED Sand (https://adafru.it/Cjb) learning guide. I did have to make some changes, and do some optimization to get it all to fit and to run at a reasonable speed. I'm pretty happy with how it turned out. I'll go over those optimization on the code page.

See the GIF below to see a short demo - its slower than C code, maybe better-called *LED Snow* than sand?



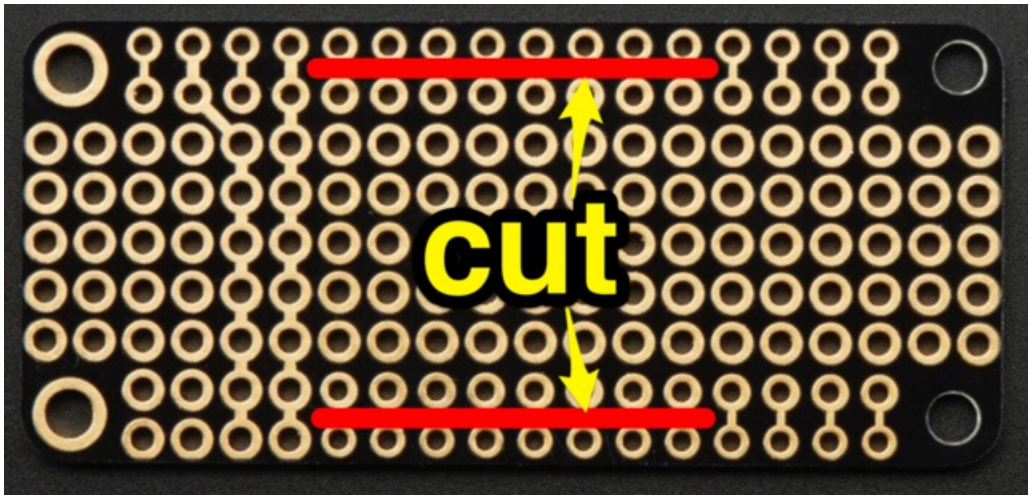
An LSM303 FeatherWing

I wanted to build this project as compactly as possible, meaning I wanted it as a stack of feather/wing boards. And that meant I needed a accelerometer FeatherWing. Alas there isn't such a thing available so I had to make my own. Since I didn't want to undertake a custom wing PCB, I looked for a breakout I could put onto a protowing.

Whatever accelerometer breakout I chose would need to fit between the header strips. In the end I found that the Flora accelerometer breakouts would fit perfectly. I chose the LSM303 as it was a bit smaller and cheaper. Additionally, all I needed for this project was an accelerometer and the LSM303 is an accelerometer/magnetometer combo. For another project I will be using the LSM9DS0 Flora breakout which includes a gyroscope as well.

Since the LSM303 didn't have a CircuitPython library yet, it gave me the opportunity to delve into writing a CircuitPython I2C driver.

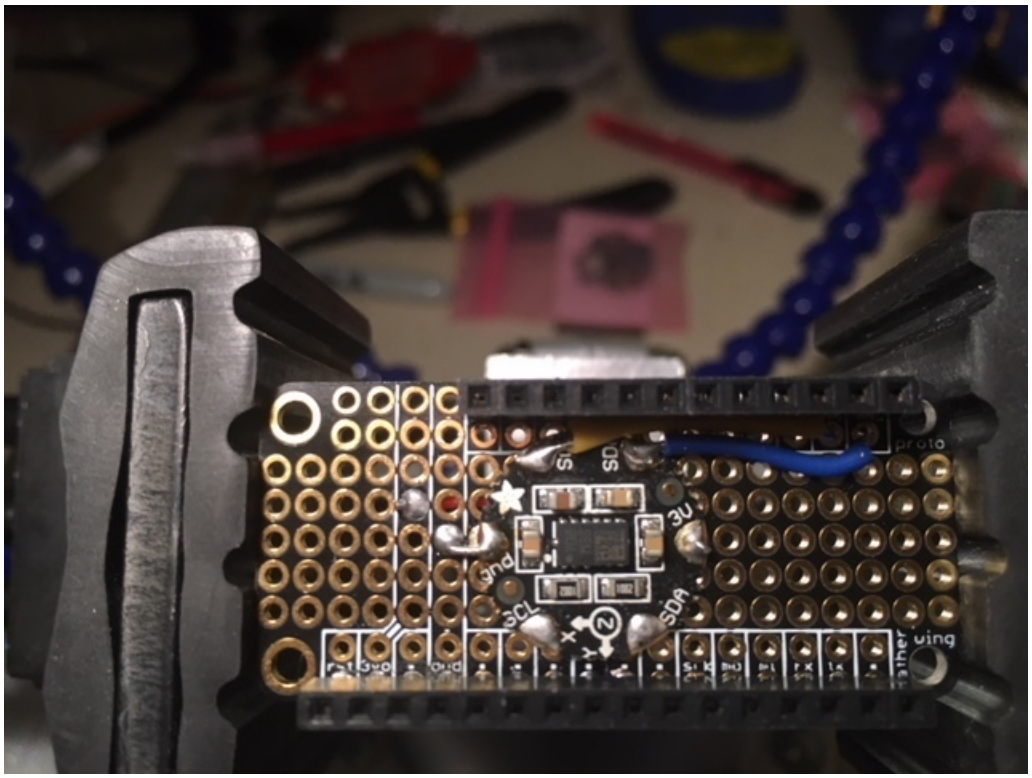
Since I mounted the Flora breakout directly on the protowing I carefully cut the traces between the header holes and their duplicate holes on the bottom of the board; just the few on either side near the center of the board where the breakout would be. This avoided the chance of any undesirable shorts/connections with the signals from the Feather. While I could have mounted it with some non-conductive foam/rubber (I've used bumpers/feet for this in other projects) I wanted a solid, unmoving mounting for this.

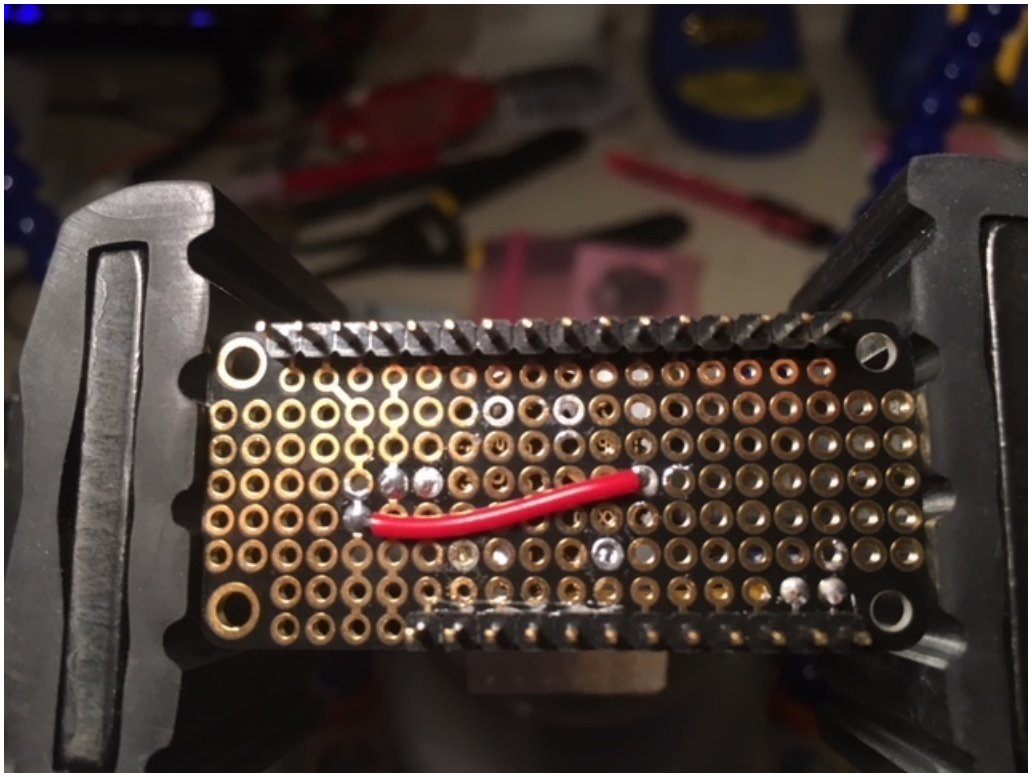


The Flora breakout is especially nice as there are just 3.3v, ground, SDA, and SCL to connect. That made wiring quick and simple.

Be very careful to get the LSM303 breakout centered on the wing, and with the X and Y axies aligned with the sides. I oriented the X axis along the length of the wing, with the Y axis aligned along the width.

Wiring is simply a matter of connecting 3v & ground to the breakout along with SDA and SCL. The Flora breakout boards are beautifully minimal.





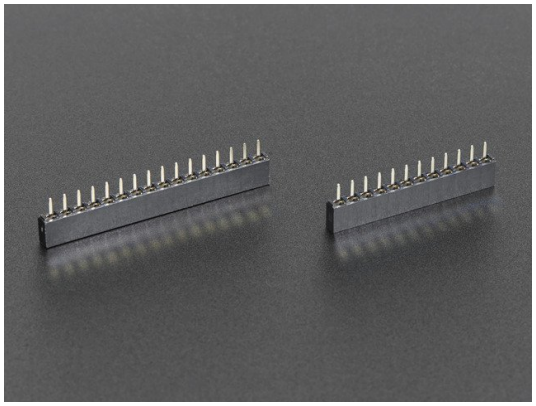
API

The API follows the unified sensor approach used in Adafruit's sensor APIs, both in C as well as CircuitPython. This means using standardized naming, as well as implementing as properties in CircuitPython. Here's an example that uses the core of the API: reading raw and processed accelerometer and magnetometer values.

```
""" Display both accelerometer and magnetometer data once per second """

import time
import board
import busio

import adafruit_lsm303

i2c = busio.I2C(board.SCL, board.SDA)
sensor = adafruit_lsm303.LSM303(i2c)

while True:
    raw_accel_x, raw_accel_y, raw_accel_z = sensor.raw_acceleration
    accel_x, accel_y, accel_z = sensor.acceleration
    raw_mag_x, raw_mag_y, raw_mag_z = sensor.raw_magnetic
    mag_x, mag_y, mag_z = sensor.magnetic

    print('Acceleration raw: ({0:6d}, {1:6d}, {2:6d}), (m/s^2): ({3:10.3f}, {4:10.3f}, {5:10.3f})'
          .format(raw_accel_x, raw_accel_y, raw_accel_z, accel_x, accel_y, accel_z))
    print('Magnetometer raw: ({0:6d}, {1:6d}, {2:6d}), (gauss): ({3:10.3f}, {4:10.3f}, {5:10.3f})'
          .format(raw_mag_x, raw_mag_y, raw_mag_z, mag_x, mag_y, mag_z))
    print('')
    time.sleep(1.0)
```

An aside: shorty stackers

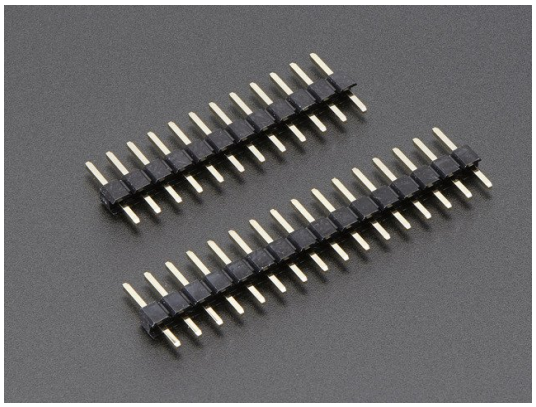
I really like the shorty headers from Adafruit for working with Feather/wings. They save quite a bit of space between boards that would otherwise be wasted and just make the assembly bigger than needed. After trying them on a few projects, I now keep a drawer full in stock for use in most, if not all, my Feather based work.



Short Headers Kit for Feather - 12-pin + 16-pin Female Headers

$1.50
IN STOCK

ADD TO CART



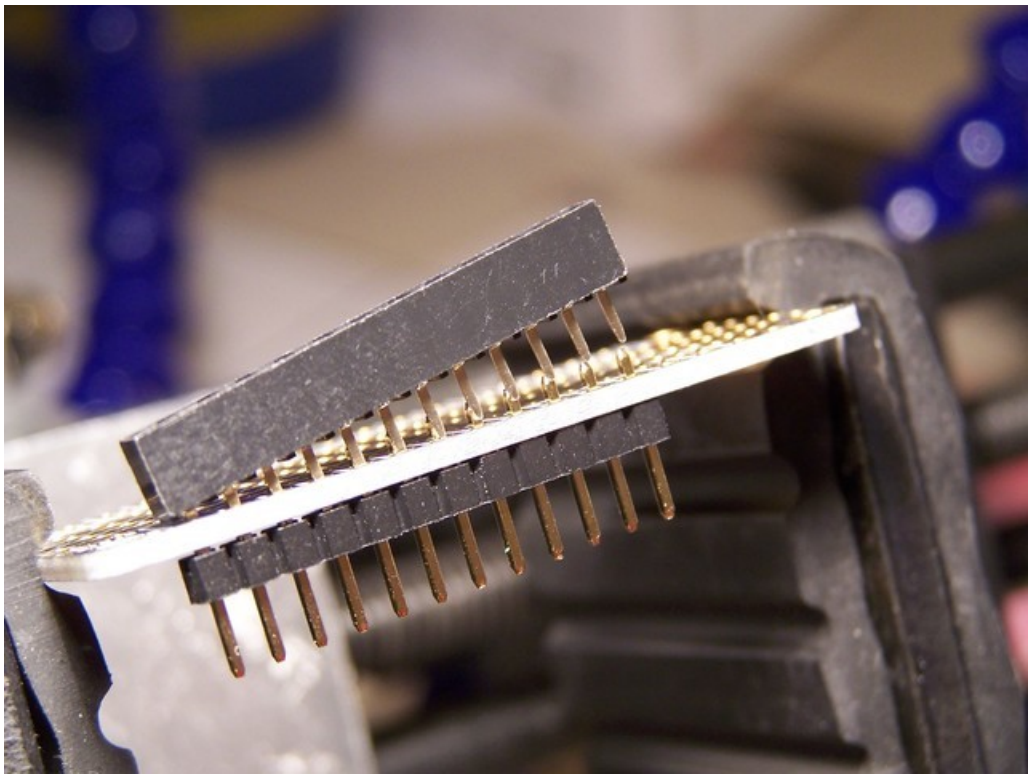
Short Feather Male Headers - 12-pin and 16-pin Male Header Set

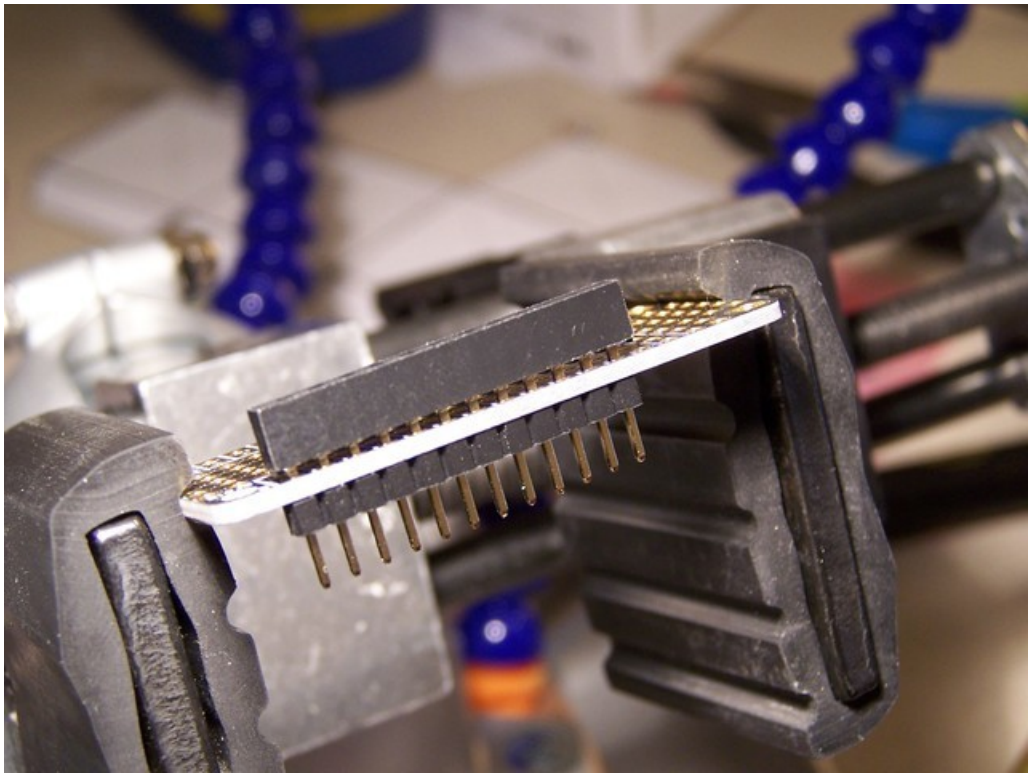
$0.50
IN STOCK

ADD TO CART

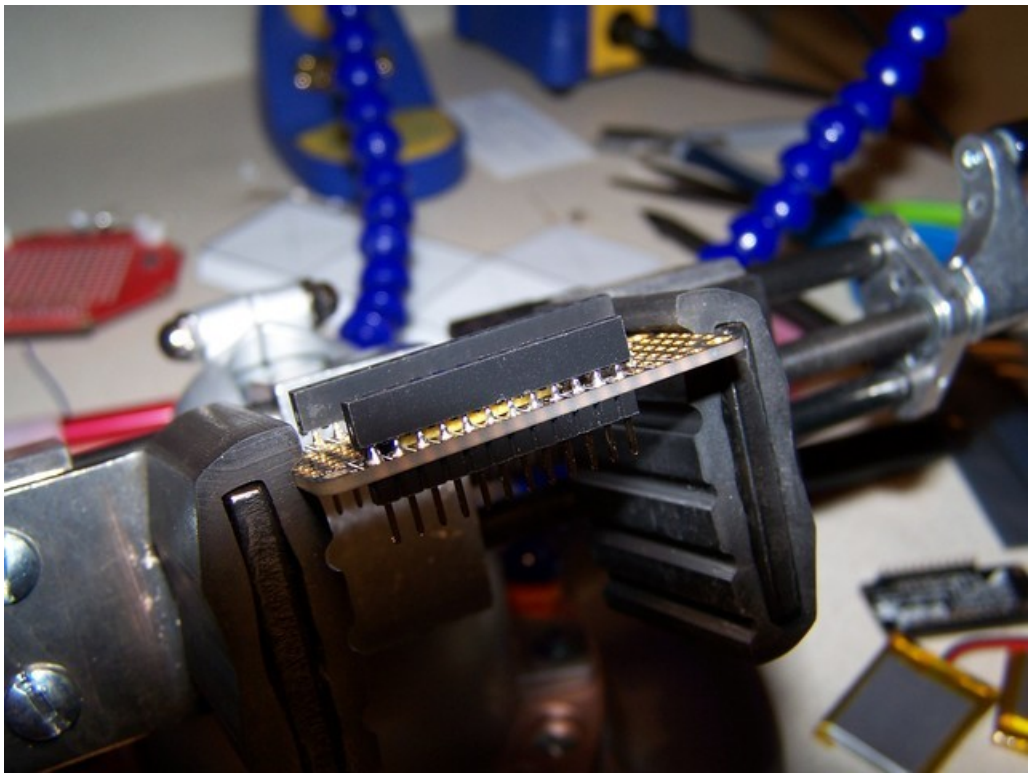
There are great if you just need to stack a wing on top of a Feather, but to go beyond one wing you need stacking headers. However, it seems that short stacking headers aren't a thing. So I did as makers do when they can't find what they want... I made my own. This project turned into a series of these "make it yourself" sub-projects.

First, do like you would when using the male headers. Before you start soldering, slip the pieces of female header in place from the top. Be sure that each header is on the same side of the matching pins that go through the board so that the spacing will be correct. I.e. **don't** put both on the outside or on the inside; you'll want one on the outside and the other on the inside. It doesn't matter which is which.

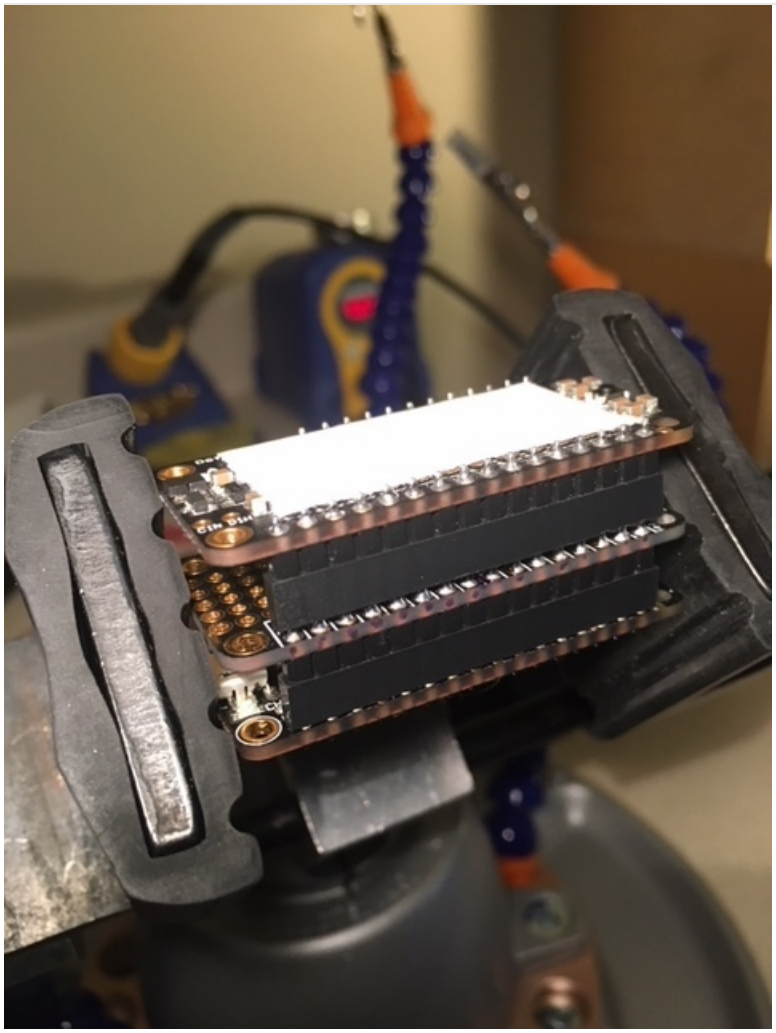




Now the tricky part. Using a iron with a thin tip (an SMT tip works well) carefully poke in between the female header strip and the board to heat the header pins. Then touch solder to them and let it wick into the hole and join the two pieces of header together. It takes a bit of practice but it's pretty easy to get a good connection. Be careful not to melt the header too much; you're bound to melt a bit here & there. It might look a bit nasty, but as long as it works, right?



Assembly



Now the boards snap together: the Dotstar wing on top of the LSM303 wing, which is on top of the Feather M0.

That arrangement puts the accelerometer chip pretty much smack-dab in the middle of the entire thing. Which is exactly where it should be.

The white? I put a couple pieces of white label stickers (the ones I use to label the drawers in my parts cabinets) over the Dotstars as a slight diffusion layer to make them show better on camera.

Code

This is a fairly literal port of Phil's C code. Of course, management of the display is quite different since it's different hardware. The major optimizations are using an array of Boolean to track where grains are, and replacing as much multiplication and division as possible with bit shifting. Other than that it is a pretty faithful port of Phil's code. So much so that I've reused his comments verbatim.

The code is complex and not immediately obvious, but the comments do a good job of walking you through it.

It doesn't always act exactly as you might expect, but keep in mind that this isn't a physics simulation, it's more of an approximation. The rules are quite simple and, as Phil said on Show & Tell, it's an example of fairly complex emergent behavior.

I'm quite pleased and impressed with how much I could get CircuitPython to do and how well it performs given the speed and memory constraints it's running under.

Division & multiplication by powers of 2

The optimizations around division and multiplication are based on the mapping between base 10 numbers and binary numbers. Especially where 2 and powers of 2 are concerned.

If we take the number 10 and represent it in binary, we have 00001010 (we'll stick to 8 bits for these examples). If we divide 10 by 2 we get 5. In binary that's 00000101. So if we keep dividing by 2 (integer division)

```
10     00001010
5      00000101
2      00000010
1      00000001
```

In decimal it's a free for all without any noticeable pattern. But look at the binary representations. Each time we divide by 2, the digits move one place to the right (with the rightmost one being discarded and a 0 begin shifted into the left.

Another thing to notice is, that since each shift right is a dividing by 2, shifting multiple times divides by a power of 2. Shifting twice is dividing by 4 (2 squared), 3 times is dividing by 8 (2 cubed), etc. In general, shifting right n digits is dividing by 2 to the power of n.

Multiplying by powers of 2 works the same way, but by shifting left rather than right.

So why is this relevant? Shifting is a lot faster than division. A good compiler (like GCC used by the Arduino IDE) or a hardware integer math unit will make this optimization (i.e. shifting for factors that are powers of 2) without the programmer having to worry about it. However, CircuitPython and MicroPython don't.

This comes up quite a bit in this code. Grain positions are kept in a coordinate system that has 256 times the resolution of the pixel coordinate system. To get the pixel location of a grain, both x and y need to be divided by 256. Fortunately that's just shifting 8 times.

Remove unnecessary code

One optimization that had a significant impact was getting rid of my Dotstar Featherwing library in favor of using the underlying Dotstar library directly. When it struck me that the display was just being updated all at once, in one place, getting rid of the library was an obvious win. Not only did it free memory, but it also removed a non-trivial amount of work that was being done each time through the loop.

Consider the Dotstar update code. In early versions I started by clearing the display out of habit. That's completely unnecessary when it's updating each pixel. Look for easy wins like this. The code should only do what it absolutely needs to. Anything more is wasting memory and/or time.

I expect I could get another jump in performance if I stripped down the Dotstar library and made a version customized for this application. One that did just what is required.

Optimize carefully

This was a good exercise in optimization. The first version of the python code could barely move 5 grains around in anything approaching a smooth speed. The current version does a reasonable job with 10 grains.

I do want to warn you about trying to optimize in an all or nothing way. Be very deliberate and cautious. Pick one thing to optimize and do it. Measure the results. If it didn't help significantly, remove it and try something else. The thing about optimizing is that it generally makes the code harder to understand so only do optimizations that make a difference.

The code

Here is the code in its entirety.

```
# The MIT License (MIT)
#
# Copyright (c) 2018 Dave Astels
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
# THE SOFTWARE.

"""
Ported from the C code writen by Phillip Burgess
as used in https://learn.adafruit.com/animated-led-sand
Explainatory comments are used verbatim from that code.
"""

import math
import random

import adafruit_dotstar
import adafruit_lsm303
import board
import busio
```

```
N_GRAINS = 10  # Number of grains of sand
WIDTH = 12  # Display width in pixels
HEIGHT = 6  # Display height in pixels
NUMBER_PIXELS = WIDTH * HEIGHT
MAX_FPS = 45  # Maximum redraw rate, frames/second
GRAIN_COLOR = (0, 0, 16)
MAX_X = WIDTH * 256 - 1
MAX_Y = HEIGHT * 256 - 1


class Grain:
    """A simple struct to hold position and velocity information
    for a single grain."""

    def __init__(self):
        """Initialize grain position and velocity."""
        self.x = 0
        self.y = 0
        self.vx = 0
        self.vy = 0


grains = [Grain() for _ in range(N_GRAINS)]
i2c = busio.I2C(board.SCL, board.SDA)
sensor = adafruit_lsm303.LSM303(i2c)
wing = adafruit_dotstar.DotStar(
    board.D13, board.D11, WIDTH * HEIGHT, 0.25, False)

oldidx = 0
newidx = 0
delta = 0
newx = 0
newy = 0

occupied_bits = [False for _ in range(WIDTH * HEIGHT)]


def index_of_xy(x, y):
    """Convert an x/column and y/row into an index into
    a linear pixel array.

    :param int x: column value
    :param int y: row value
    """
    return (y >> 8) * WIDTH + (x >> 8)


def already_present(limit, x, y):
    """Check if a pixel is already used.

    :param int limit: the index into the grain array of
    the grain being assigned a pixel Only grains already
    allocated need to be checks against.
    :param int x: proposed clumn value for the new grain
    :param int y: proposed row valuse for the new grain
    """
    for j in range(limit):
        if x == grains[j].x or y == grains[j].y:
            return True
```

```
            return True
    return False


for g in grains:
    placed = False
    while not placed:
        g.x = random.randint(0, WIDTH * 256 - 1)
        g.y = random.randint(0, HEIGHT * 256 - 1)
        placed = not occupied_bits[index_of_xy(g.x, g.y)]
    occupied_bits[index_of_xy(g.x, g.y)] = True
    g.vx = 0
    g.vy = 0

while True:
    # Display frame rendered on prior pass.  It's done immediately after the
    # FPS sync (rather than after rendering) for consistent animation timing.

    for i in range(NUMBER_PIXELS):
        wing[i] = GRAIN_COLOR if occupied_bits[i] else (0, 0, 0)
    wing.show()

    # Read accelerometer...
    f_x, f_y, f_z = sensor.raw_acceleration
    ax = f_x >> 8  # Transform accelerometer axes
    ay = f_y >> 8  # to grain coordinate space
    az = abs(f_z) >> 11  # Random motion factor
    az = 1 if (az >= 3) else (4 - az)  # Clip & invert
    ax -= az  # Subtract motion factor from X, Y
    ay -= az
    az2 = (az << 1) + 1  # Range of random motion to add back in

    # ...and apply 2D accel vector to grain velocities...
    v2 = 0  # Velocity squared
    v = 0.0  # Absolute velociy
    for g in grains:
        g.vx += ax + random.randint(0, az2)  # A little randomness makes
        g.vy += ay + random.randint(0, az2)  # tall stacks topple better!

        # Terminal velocity (in any direction) is 256 units -- equal to
        # 1 pixel -- which keeps moving grains from passing through each other
        # and other such mayhem.  Though it takes some extra math, velocity is
        # clipped as a 2D vector (not separately-limited X & Y) so that
        # diagonal movement isn't faster

        v2 = g.vx * g.vx + g.vy * g.vy
        if v2 > 65536:  # If v^2 > 65536, then v > 256
            v = math.floor(math.sqrt(v2))  # Velocity vector magnitude
            g.vx = (g.vx // v) << 8  # Maintain heading
            g.vy = (g.vy // v) << 8  # Limit magnitude

    # ...then update position of each grain, one at a time, checking for
    # collisions and having them react.  This really seems like it shouldn't
    # work, as only one grain is considered at a time while the rest are
    # regarded as stationary.  Yet this naive algorithm, taking many not-
    # technically-quite-correct steps, and repeated quickly enough,
    # visually integrates into something that somewhat resembles physics.
    # (I'd initially tried implementing this as a bunch of concurrent and
    # "realistic" elastic collisions among circular grains, but the
    # calculations and volument of code quickly got out of hand for both
```

```
    # the tiny 8-bit AVR microcontroller and my tiny dinosaur brain.)

    for g in grains:
        newx = g.x + g.vx  # New position in grain space
        newy = g.y + g.vy
        if newx > MAX_X:  # If grain would go out of bounds
            newx = MAX_X  # keep it inside, and
            g.vx //= -2  # give a slight bounce off the wall
        elif newx < 0:
            newx = 0
            g.vx //= -2
        if newy > MAX_Y:
            newy = MAX_Y
            g.vy //= -2
        elif newy < 0:
            newy = 0
            g.vy //= -2

        oldidx = index_of_xy(g.x, g.y)  # prior pixel
        newidx = index_of_xy(newx, newy)  # new pixel
        # If grain is moving to a new pixel...
        if oldidx != newidx and occupied_bits[newidx]:
            # but if that pixel is already occupied...
            # What direction when blocked?
            delta = abs(newidx - oldidx)
            if delta == 1:  # 1 pixel left or right
                newx = g.x  # cancel x motion
                # and bounce X velocity (Y is ok)
                g.vx //= -2
                newidx = oldidx  # no pixel change
            elif delta == WIDTH:  # 1 pixel up or down
                newy = g.y  # cancel Y motion
                # and bounce Y velocity (X is ok)
                g.vy //= -2
                newidx = oldidx  # no pixel change
            else:  # Diagonal intersection is more tricky...
                # Try skidding along just one axis of motion if
                # possible (start w/ faster axis). Because we've
                # already established that diagonal (both-axis)
                # motion is occurring, moving on either axis alone
                # WILL change the pixel index, no need to check
                # that again.
                if abs(g.vx) > abs(g.vy):  # x axis is faster
                    newidx = index_of_xy(newx, g.y)
                    # that pixel is free, take it! But...
                    if not occupied_bits[newidx]:
                        newy = g.y  # cancel Y motion
                        g.vy //= -2  # and bounce Y velocity
                    else:  # X pixel is taken, so try Y...
                        newidx = index_of_xy(g.x, newy)
                        # Pixel is free, take it, but first...
                        if not occupied_bits[newidx]:
                            newx = g.x  # Cancel X motion
                            g.vx //= -2  # Bounce X velocity
                        else:  # both spots are occupied
                            newx = g.x  # Cancel X & Y motion
                            newy = g.y
                            g.vx //= -2  # Bounce X & Y velocity
                            g.vy //= -2
                            newidx = oldidx  # Not moving
```

```
                    newidx = oldidx  # Not moving
            else:  # y axis is faster. start there
                newidx = index_of_xy(g.x, newy)
                # Pixel's free! Take it! But...
                if not occupied_bits[newidx]:
                    newx = g.x  # Cancel X motion
                    g.vx //= -2  # Bounce X velocity
                else:  # Y pixel is taken, so try X...
                    newidx = index_of_xy(newx, g.y)
                    # Pixel is free, take it, but first...
                    if not occupied_bits[newidx]:
                        newy = g.y  # cancel Y motion
                        g.vy //= -2  # and bounce Y velocity
                    else:  # both spots are occupied
                        newx = g.x  # Cancel X & Y motion
                        newy = g.y
                        g.vx //= -2  # Bounce X & Y velocity
                        g.vy //= -2
                        newidx = oldidx  # Not moving
        occupied_bits[oldidx] = False
        occupied_bits[newidx] = True
        g.x = newx
        g.y = newy
```

Downloads

https://adafru.it/Axe

https://adafru.it/Axe

https://adafru.it/Ax6

https://adafru.it/Ax6

You will also need the CircuitPython library bundle that matches the version of CircuitPython you are using (I used 2.2.0 when working on this).

After downloading and unzipping, you can just drag the `adafruit_lsm303.py` file into `CIRCUITPY/lib`.

For the demo code, just rename it `main.py` and drag onto `CIRCUITPY`.