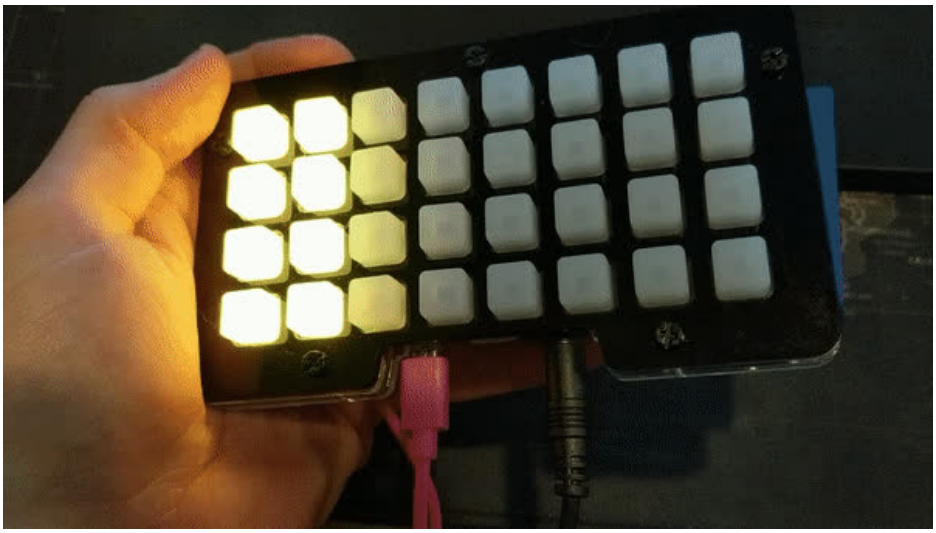
adafruit learning system

NeoTrellis M4 Noisy Grains of Sand

Created by John Thurmond

Last updated on 2020-12-17 10:52:44 AM EST

Overview



The 'grains of sand' code is a great way to demo any LED matrix with an accelerometer, and the NeoTrellis M4 can make this even more entrancing by adding tactile, tap, and sound interaction as well.

This guide will show you how to get this code up and running on the NeoTrellis M4. A walk through the code will show you how you can get started making your own styles of interactions.

WARNING: This demo can be addictive - do not loan your NeoTrellis M4 running this demo out to others without being prepared to point them to this article and/or buy/setup one for them yourself!

Parts

Your browser does not support the video tag.

Adafruit NeoTrellis M4 with Enclosure and Buttons Kit Pack

So you've got a cool/witty name for your band, a Soundcloud account, a 3D-printed Daft Punk...

Out of Stock

Out of Stock

USB cable - USB A to Micro-B

This here is your standard A to micro-B USB cable, for USB 1.1 or 2.0. Perfect for connecting a PC to your Metro, Feather, Raspberry Pi or other dev-board or...

$2.95

In Stock

Add to Cart

To listen to the sounds, you can choose to use headphones, speakers, or a connection to your own amplifier:

Analog Potentiometer Volume Adjustable TRRS Headset

Most modern headphone sets are purely digital - with three volume control buttons in-line with the cable. These headphones are interesting in that they have an analog volume...

$7.50

In Stock

Add to Cart

USB Powered Speakers

Add some extra boom to your audio project with these powered loudspeakers. We sampled half a dozen different models to find ones with a good frequency response, so you'll get...

Out of Stock

Out of Stock

3.5mm Male/Male Stereo Cable

Seamlessly transmit high-quality stereo audio with this 3.5mm Male/Male Stereo Cable. Ideal for "passing the AUX cord,"...

$2.50

In Stock

Add to Cart

Install CircuitPython

Getting Started with CircuitPython

Are you new to using CircuitPython? No worries, there is a full getting started guide here (https://adafru.it/cpy-welcome).

Additionally, Adafruit has made life even easier for you with this getting started guide for the TrellisM4 (https://adafru.it/C-O). If you need to install CircuitPython, go to that page and follow the steps.

The great news is that this isn't really any more complicated than copying files to a USB drive! Just make sure you put things in the right place!

Install Libraries

Check out the CircuitPython Libraries page of this guide (https://adafru.it/CYo) for a detailed explanation of how to load the library bundle on your board.

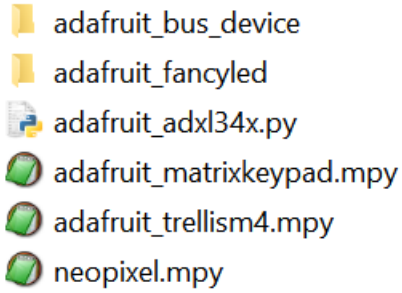
For this project, you will also need to install the following libraries according to the instructions above:

- **adafruit_bus_device** (folder)
- **adafruit_fancyled** (folder)
- **adafruit_adxl34x.mpy** (file)
- **adafruit_matrixkeypad.mpy** (file)
- **adafruit_trellism4.mpy** (file)
- **neopixel.mpy** (file)

This will let us take advantage of the features of the TrellisM4, they keypad, the accelerometer, and the blinky lights - all of which we'll need.

Place the libraries in the **/lib** folder of your **CIRCUITPY** drive that shows up when the NeoTrellis M4 is connected to your computer via USB. If the **lib** folder is not on your **CIRCUITPY** drive, create the folder and place the needed libraries inside it.

The contents of the **/lib** folder should look like this:



Code

Install Noisy Grains of Sand code

The code for this project is available in the Adafruit Learning System Github here (https://adafru.it/DYs). In the code listing below, you can click **code.py** and save it to your computer.

Now that you have CircuitPython installed, you simply need to place the **code.py** file onto the NeoTrellis M4 by copying it into the mounted USB folder named **CIRCUITPY.**

Install Sound File

One of the great things that makes the NeoTrellis M4 different from other LED matrices is that it can play sounds! We should take advantage of that for this grains of sand demo.

I recommend this water click sound (https://adafru.it/DYt). This is derived from a Creative Commons (https://adafru.it/DSc) licensed sound available on FreeSound (https://adafru.it/DSd), and I have processed it to work well on the NeoTrellisM4 (PS - the author of this sound really likes this demo!).

Copy **water-click.wav** onto your **CIRCUITPY** drive in the main directory.

There are many other sound files available which could more humorous or suitable for your application. Please feel free to experiment!

```
# Digital sand demo uses the accelerometer to move sand particles in a
# realistic way.  Tilt the board to see the sand grains tumble around and light
# up LEDs.  Based on the code created by Phil Burgess and Dave Astels, see:
#   https://learn.adafruit.com/digital-sand-dotstar-circuitpython-edition/code
#   https://learn.adafruit.com/animated-led-sand
# Ported to NeoTrellis M4 by John Thurmond.
#
# The MIT License (MIT)
#
# 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.

import math
import random
import board
import audioio
import busio
import adafruit_trellism4
import adafruit_adxl34x

N_GRAINS = 8  # Number of grains of sand
```

```
N_GRAINS = 8  # Number of grains of sand
WIDTH = 8  # Display width in pixels
HEIGHT = 4  # Display height in pixels
NUMBER_PIXELS = WIDTH * HEIGHT
MAX_FPS = 20  # Maximum redraw rate, frames/second
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)]

color = random.randint(1, 254) # Set a random color to start
current_press = set() # Get ready for button presses

# Set up Trellis and accelerometer
trellis = adafruit_trellism4.TrellisM4Express(rotation=0)
i2c = busio.I2C(board.ACCELEROMETER_SCL, board.ACCELEROMETER_SDA)
sensor = adafruit_adxl34x.ADXL345(i2c)

# Add tap detection - with a pretty hard tap
sensor.enable_tap_detection(threshold=50)

color_mode = 0

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

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

# Add Audio file...
f = open("water-click.wav", "rb")
wav = audioio.WaveFile(f)
print("%d channels, %d bits per sample, %d Hz sample rate " %
      (wav.channel_count, wav.bits_per_sample, wav.sample_rate))
audio = audioio.AudioOut(board.A1)
#audio.play(wav)

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
```

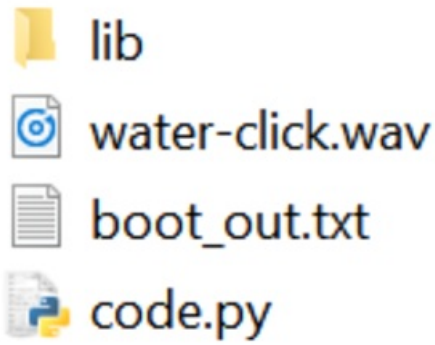
```
    the grain being assigned a pixel Only grains already
    allocated need to be checks against.
    :param int x: proposed clumn value for the new grain
    :param int y: proposed row valuse for the new grain
    """
    for j in range(limit):
        if x == grains[j].x or y == grains[j].y:
 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

def wheel(pos):
    # Input a value 0 to 255 to get a color value.
    # The colours are a transition r - g - b - back to r.
    if pos < 0 or pos > 255:
        return 0, 0, 0
    if pos < 85:
        return int(255 - pos*3), int(pos*3), 0
    if pos < 170:
        pos -= 85
        return 0, int(255 - pos*3), int(pos*3)
    pos -= 170
    return int(pos * 3), 0, int(255 - (pos*3))

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:
    # Check for tap and adjust color mode
    if sensor.events['tap']:
        color_mode += 1
    if color_mode > 2:
        color_mode = 0

    # 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):

        # Some color options:

        # Random color every refresh
        if color_mode == 0:
            if occupied_bits[i]:
                trellis.pixels[(i%8, i//8)] = wheel(random.randint(1, 254))
            else:
                trellis.pixels[(i%8, i//8)] = (0, 0, 0)
        # Color by pixel
        if color_mode == 1:
            trellis.pixels[(i%8, i//8)] = wheel(i*2) if occupied_bits[i] else (0, 0, 0)

        # Change color to random on button press, or cycle when you hold one down
        if color_mode == 2:
            trellis.pixels[(i%8, i//8)] = wheel(color) if occupied_bits[i] else (0, 0, 0)

    # Change color to a new random color on button press
    pressed = set(trellis.pressed_keys)
```

```
    for press in pressed - current_press:
        if press:
            print("Pressed:", press)
            color = random.randint(1, 254)
            print("Color:", color)

    # Read accelerometer...
    f_x, f_y, f_z = sensor.acceleration

    # I had to manually scale these to get them in the -128 to 128 range-ish - should be done better
    f_x = int(f_x * 9.80665 * 16704/1000)
    f_y = int(f_y * 9.80665 * 16704/1000)
    f_z = int(f_z * 9.80665 * 16704/1000)

    ax = f_x >> 3  # Transform accelerometer axes
    ay = f_y >> 3  # to grain coordinate space
    az = abs(f_z) >> 6  # Random motion factor

    print("%6d %6d %6d"%(ax,ay,az))
    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

    # Adjust axes for the NeoTrellis M4 (reuses code above rather than fixing it - inefficient)
    ax2 = ax
    ax = -ay
    ay = ax2

    # ...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
                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
        if oldidx != newidx:
            audio.play(wav) # If there's an update, play the sound
        g.x = newx
        g.y = newy
```

Sanity Check

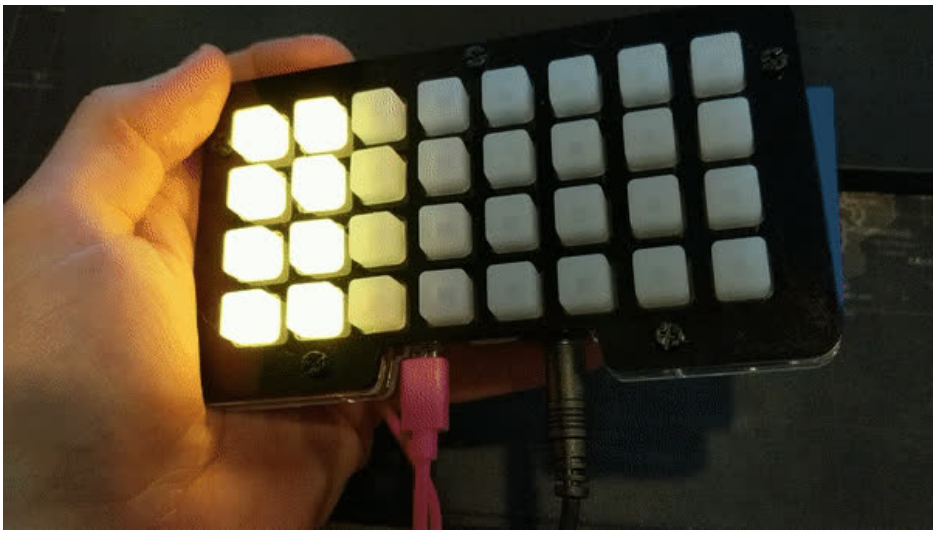
- 

Assuming you've done everything correctly, your mounted USB folder on your NeoTrellis M4 should look something that what's on the left, with the **code.py** and **water-click.wav** file in place.

If this is the case, you should be up and running - now go play!

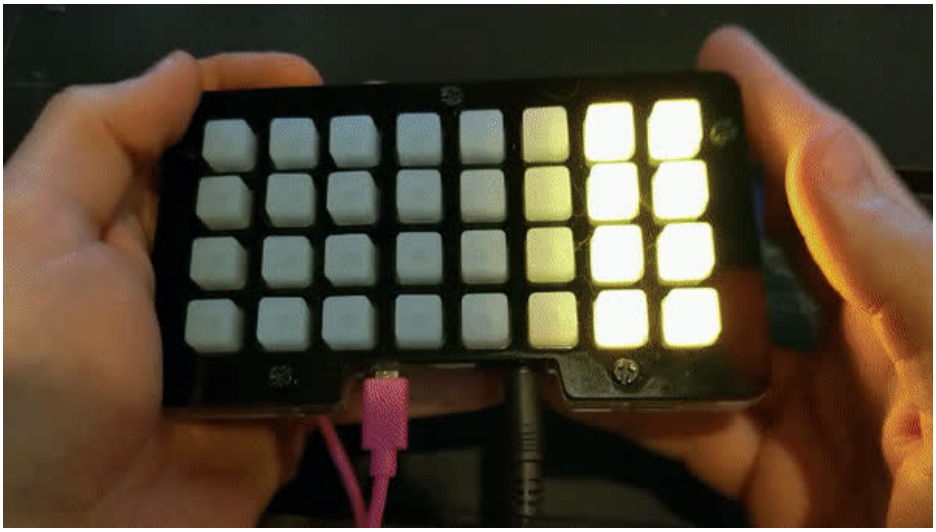
Play!

The main thing the 'grains of sand' code does is act like...grains of sand! If you tilt the TrellisM4 in any direction, the motion is detected by the accelerometer and the sand grains 'flow' in the direction of tilt.

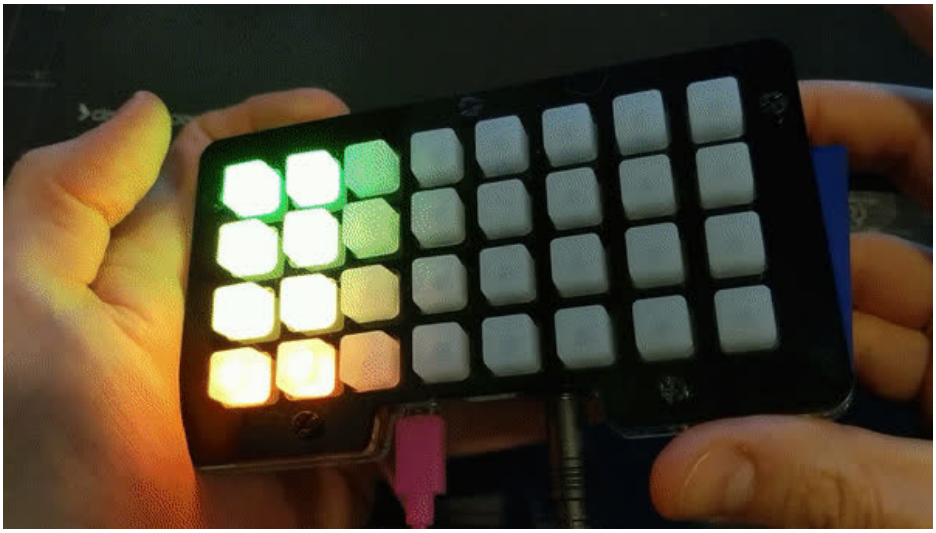


A sharp 'tap' anywhere on the NeoTrellis will switch the color mode - there are three modes built in:

- Rapidly cycles through random colors (default)
- A gradient based on the pixel index
- Solid color that changes with a button press



In the solid color mode, pushing any button on the NeoTrellis will rapidly change the color randomly. You can hold down for a random color effect, or quickly press to change to a new random color.



Don't forget the sound!

If you plug headphones into the jack on the NeoTrellis M4, or alternatively plug it into a set of powered speakers, the 'water drop' sound will play each time a sand grain moves.

Code Walkthrough

The code used here was a modification of the CircuitPython Digital Sand code for the sino:bit (https://adafru.it/DSe) by Tony DiCola (https://adafru.it/nYb), which was originally developed by Dave Astels (https://adafru.it/DSf) for the LSM303 FeatherWing (https://adafru.it/Ckh) as a CircuitPython port of the LED Sand (https://adafru.it/Cjb) code by Phillip Burgess (https://adafru.it/iPc).

For the specifics of how the sand grain code works, please refer to the guides above.

I will talk about the specific modifications needed to use this code on the NeoTrellis M4 and to take advantage of its features.

Setting up for the NeoTrellis M4

```
import time
import board
import audioio
import audiocore
import busio
import adafruit_trellism4
import math
import random
import adafruit_adxl34x

N_GRAINS = 8 # Number of grains of sand
WIDTH = 8 # Display width in pixels
HEIGHT = 4 # Display height in pixels
```

This bit of the code imports the libraries specifically needed for the NeoTrellis M4, and sets up dimensions of the matrix (8x4 pixels). One of the easiest things you can do to change the code is to modify the number of grains of sand in the demo by changing the `N_GRAINS` variable.

Setting up the board and accelerometer

```
trellis = adafruit_trellism4.TrellisM4Express(rotation=0)
i2c = busio.I2C(board.ACCELEROMETER_SCL, board.ACCELEROMETER_SDA)
sensor = adafruit_adxl34x.ADXL345(i2c)

sensor.enable_tap_detection(threshold=50)
```

This sets up the NeoTrellis M4 in the state we want, and sets up `sensor` to refer to the ADXL343 on the board.

We also let this sensor know to enable tap detection, since we'll use that to change color modes. This is one of the great features of this sensor, as there are a few additional modes of interaction beyond just reporting acceleration values.

Audio File Setup

```
f = open("water-click.wav", "rb")
wav = audiocore.WaveFile(f)
print("%d channels, %d bits per sample, %d Hz sample rate " %
(wav.channel_count, wav.bits_per_sample, wav.sample_rate))
audio = audioio.AudioOut(board.A1)
```

Here is the sound file and setup the audio output for the NeoTrellis M4

Tap Detection to Change Color Mode

```
if sensor.events['tap']: color_mode += 1
if color_mode > 2: color_mode = 0
```

This just watches for 'tap' events from the accelerometer and switches the color mode when one is detected. Who would have thought it would take just a few lines of code to set up a tap sensor on a device?

Color Modes and Button Presses

```
if color_mode == 0: trellis.pixels[(i%8, i//8)] = wheel(random.randint(1, 254)) if occupied_bits[i] else (0, 0, 0
)

if color_mode == 1: trellis.pixels[(i%8, i//8)] = wheel(i*2) if occupied_bits[i] else (0, 0, 0)

if color_mode == 2: trellis.pixels[(i%8, i//8)] = wheel(color) if occupied_bits[i] else (0, 0, 0)

pressed = set(trellis.pressed_keys)
for press in pressed - current_press:
if press:
print("Pressed:", press)
color = random.randint(1, 254)
print("Color:", color)
```

This sets up the three different color modes, which simply define each of the NeoTrellis M4's NeoPixels to a different color in slightly different ways.

It also watches for pressed buttons to change the color of the solid color mode.

Accelerometer Value Correction

I struggled the most to port the accelerometer calculation part of the code to the NeoTrellis M4. Ironically, that's because the accelerometer on the NeoTrellis M4 is *easier to use!*

Instead of the raw accelerometer values that this code expects to see, the ADXL343 gives you very nice corrected values. I found the easiest route to port the code was to emulate uncorrected values.

```
f_x = int(f_x * 9.80665 * 16704/1000)
f_y = int(f_y * 9.80665 * 16704/1000)
f_z = int(f_z * 9.80665 * 16704/1000)
```

I also had to change the axes around to match the way I set the direction of the matrix up.

```
ax2 = ax
ax = -ay
ay = ax2
```

Playing the Sound at the Right Time

I also struggled a bit with deciding when was the right time to play the sound. After some trial and error, I ended up simply doing it every time the 'grains of sand' matrix was modified, and after all the collision detection.

```
if oldidx != newidx: audio.play(wav) # If there's an update, play the sound
```

...and that's it!

When you have a feature-rich board, it can sometimes be a pain to explore and develop code for each of the features. However, one of the remarkable things this code shows is how easy this is to do on the NeoTrellis M4 using CircuitPython.

With just a few lines of code, you have:

- Set up an audio file and played it
- Enabled 'tap' detection on the accelerometer and used it
- Enabled button presses
- Made multi-mode light color changes

That's pretty remarkable - now it's your turn!

Next Steps

There are a lot of ways you can make this code your own - I encourage you to play around with it and add new features.

Change the colors

There are lots of great NeoPixel CircuitPython code examples (https://adafru.it/DSg) out there. Modify the code to do different kinds of color cycling or animation!

It's simple to add new color modes to this code, just don't forget to change the following line to increase the number of modes available:

```
if color_mode > 2: color_mode = 0
```

Add new interactions

The ADXL343 is capable of detecting more than just taps - you can just as simply detect double-taps and free-fall as well. Can you use these modes for more interactivity?

We've also just used the buttons in a simple way. Can you change the button presses to be more sophisticated, and perhaps change the color of the grains in cleverer ways?

Add more sounds

I just used one sound - can you add more sounds for different types of grain interactions? Maybe a sound for grain collisions?

Make it faster

This is not very optimized code, as has been noted before, this should perhaps be called 'grains of snow' rather than 'grains of sand' due to their sluggish movement. Can you make it run faster?

Break it!

Most of all, I encourage you just to get into the code and break things. Failure is the greatest teacher, and many of us got started in electronics and coding by breaking things others have built and having to put them back together (sorry Mom and Dad). I'd love to see what you do with this code, and please drop me a line if you do something with it, or simply just enjoy it!