NeoTrellis M4 Animated Dice Roller

Created by Dave Astels

https://learn.adafruit.com/neotrellis-dice

Last updated on 2024-06-03 02:33:22 PM EDT

Table of Contents

Overview

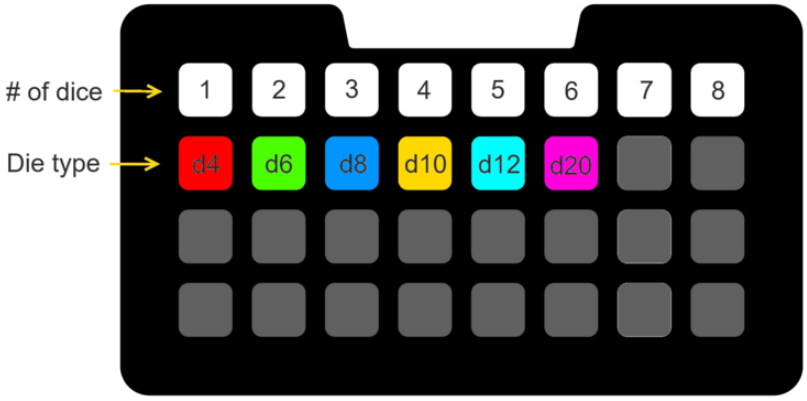
In this project we'll take the NeoTrellisM4 and use it to make a dice roller.

We want to be able to select between 1 and 8 dice to roll, and the type of dice: 4, 6, 8, 10, 12, or 20 sided. To keep the interface simply, we can't mix types of dice.

The number of dice is selected using the top row of buttons: 1-8 starting from the left. As the count is selected, a bar fills in from the left.

The type of die is selected by buttons on the second row. Each type of die is assigned a unique color that is used to display the result of rolls as well as the associated selection button when a die is selected.

Button use and color is shown below.



Once the number and type of dice are selected, shaking the Trellis will roll the dice and display the result.

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

Adafruit suggests using the Mu editor to edit your code and have an interactive REPL in CircuitPython. You can learn about Mu and its installation in this tutorial (https://adafru.it/ANO).

There's a guide to get you up and running with CircuitPython specifically for the NeoTrellisM4 (https://adafru.it/C-O). You should read it before starting to get the most recent CircuitPython build for the NeoTrellisM4 installed and running along with the required libraries.

Parts

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...
https://www.adafruit.com/product/4020

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...

https://www.adafruit.com/product/592

Code Walkthrough

As is usual for a Python script, we start by importing what is needed, and initializing the hardware. Most of what we need is in the (Neo)TrellisM4 Express class. However, the accelerometer has to be set up separately.

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

# Set up the trellis and accelerometer

trellis = adafruit_trellism4.TrellisM4Express()

i2c = busio.I2C(board.ACCELEROMETER_SCL, board.ACCELEROMETER_SDA)
accelerometer = adafruit_adxl34x.ADXL345(i2c)
```

Since we want to display numbers on the Trellis's pixels, we need to define the bitmaps. The approach used is to use a string of three characters (the width of each glyph). Here we use spaces and asterisks to denote unlit/lit pixels. A 3x4 grid isn't much to work with, but this font does ok. See http://pixeljoint.com/pixelart/24172.htm (https://adafru.it/D3v) by Adam Simpson.

```
number_patterns = [
    [" * ", "* *", "* *", "***"],         # 0
    [" * ", "** ", " * ", "***"],         # 1
    ["** ", "  *", " * ", "***"],         # 2
    ["** ", "  *", " **", "***"],         # 3
    ["* *", "* *", "***", "  *"],         # 4
    ["***", "*  ", " **", "***"],         # 5
    [" **", "*  ", "***", "***"],         # 6
    ["***", "  *", " * ", " * "],         # 7
    [" * ", "* *", "***", "***"],         # 8
    [" * " ,"* *", " **", "  *"]          # 9
]
```

Now that the look of the digits is decided, we need to display them. First we need a function to display a single digit. Since this will be used for both the ones and tens place, it will need to have a parameter telling it where to display its digit; that's `offset`. Since each type of die will have it's own color, that needs to be passed in as well. Finally there's a flag to specify whether a zero will be displayed or left blank. Why? If the number is less than 10 it would be nice to suppress the leading 0.

The `display_digit` function simply gets the pixel pattern for the value of the `number` parameter, and loops through the rows and columns setting each pixel based on the pattern.

```
def display_digit(number, offset, color, force_zero):
    """Display a digit.
    number     -- the number (0-9) to display
    offset     -- the left-most column of the displayed digit
    color      -- the RGB color to use to display the digit
    force_zero -- whether to leave a 0 blank (False) or display it
    """
    bits = number_patterns[number]
    for row in range(4):
        for col in range(3):
            if bits[row][col] == " " or (number == 0 and not force_zero):
                trellis.pixels[col + offset, row] = (0, 0, 0)
            else:
                trellis.pixels[col + offset, row] = color
```

Now that a single digit can be displayed, that function gets used in another that is responsible for displaying the entire number. This displays the ones digit, and then the tens digit (suppressing it if it's a leading zero). Note that if the number is 100 or more, the tens digit is always displayed.

What happens next is a bit different. Since a roll with several larger dice could result in a number greater than will fit in two digits, the hundreds place uses the otherwise unused pixels on the far left to indicate 100, 200, 300, or 400. This is just a matter of scanning through, lighting the appropriate one.

That's it other than a bit of range checking on the number being displayed.

```
def display_number(number, color):
    """Display a multi-digit number.
    If the number is &gt; 99, the hundreds digit (1-4) is indicated on the far left
column:
        100 is the top pixel, 400 is the bottom.
    number      -- the number to display
    color       -- the RGB color to use to display the digit
    force_zeros -- whether to leave a 0 in the tens place blank (False) or display
it
    """
    if number &gt;= 500 or number &lt; 0:
        return False
    display_digit(number % 10, 5, color, True)
    display_digit((number // 10) % 10, 1, color, number &gt;= 100)
    hundreds = number // 100
    for h in range(4):
```

```
        if h + 1 == hundreds:
            trellis.pixels[0, h] = (255, 255, 255)
        else:
            trellis.pixels[0, h] = (0, 0, 0)
    return True
```

While we could just slap a number onto the display without fanfare, it's far more interesting to have some animation suggesting rolling dice.

That's what the `animate_to` function does. Simply, it displays 10 random two digit numbers with a slight delay between them. After that's done, the number to be displayed is shown.

```
def animate_to(number, color):
    """Perform an animation (displaying random numbers) before displaying the
requested number.
    number -- the number to eventually display
    color  -- the color to use (indicates the type of dice used)
    """
    for _ in range(10):
        trellis.pixels.fill((0, 0, 0))
        display_number(random.randint(10, 99), color, True)
        time.sleep(0.1)
    trellis.pixels.fill((0, 0, 0))
    display_number(number, color)
```

Now that we can display numbers we need numbers to display.

That's where the roll function comes in. It has parameters for the number of dice to roll, as well as the number of sides those dice have. It generates the specified number of random integers between 1 and the number of sides and returns the sum.

```
def roll(number, sides):
    """Generate a random dice roll.
    Returns the total of the roll.
    number -- the number of dice to roll
    sides  -- the number of side on dice to roll (4, 6, 8, 10, 12, 20)
    """
    total = 0
    for _ in range(number):
        total += random.randint(1, sides + 1)
    return total
```

As mentioned earlier, we want to roll the dice when the Trellis is shaken. To know when that happens we need to use the accelerometer. The approach to detecting a shake is inspired by this article (https://adafru.it/19yb). Put simply, look for sudden significant changes in all accelerometer axis. To do that we keep track of the previous reading, and compare the new one to it.

The `bound` variable is the threshold for axis change.

```
previous_reading = [None, None, None]
bound = 4.0

def shaken():
    """Detect when the Trellis is shaken.
    See http://www.profoundlogic.com/docs/display/PUI/Accelerometer+Test+for+Shaking
    TL;DR one or more axis experiences a significant (set by bound) change very
quickly
    Returns whether a shake was detected.
    """
    global previous_reading
    result = False
    x, y, z = accelerometer.acceleration
    if previous_reading[0] is not None:
        result = (math.fabs(previous_reading[0] - x) &gt; bound and
                  math.fabs(previous_reading[1] - y) &gt; bound and
                  math.fabs(previous_reading[2] - z) &gt; bound)
    previous_reading = (x, y, z)
    return result
```

Finally, we have the main loop.

Each time through, we start by clearing the display and initializing the shake detector.

Next we have another loop that cycles until a shake is detected. In this loop we check for presses in the top two rows and process them into the number and type of die to roll. By looping until a shake is detected, we can keep adjusting count and die until we're ready to roll.

Once a shake is detected, we confirm that both count and die have, indeed, been selected. If so the dice are rolled and the result displayed in the appropriate color. It's displayed for 5 seconds, or until any key is pressed.

Then the outer loop restarts.

```
selected_count = -1
selected_die = -1

while True:
    trellis.pixels.fill((0, 0, 0))
    previous_reading = accelerometer.acceleration
    while not shaken():
        # update selected count/die if a respective selection has been made
        pressed = trellis.pressed_keys    # Get the pressed buttons
        count_selector = [key for key in pressed if key[1] == 0]   # first row
presses
        die_selector = [key for key in pressed if key[1] == 1]     # second row
presses
        if len(count_selector) &gt; 0:
            selected_count = count_selector[0][0] + 1
        if len(die_selector) &gt; 0 and die_selector[0][0] &lt; 6:   # only 6 types
of dice
            selected_die = die_selector[0][0]

        # update the pixels to reflect the selections
        for i in range(8):
            if i &lt; selected_count:        # Display a bar for the count
                trellis.pixels[i, 0] = (128, 64, 16)
```

```
            else:
                trellis.pixels[i, 0] = (0, 0, 0)
            if i == selected_die:         # Just the selected dice, in the
appropriate color
                trellis.pixels[i, 1] = die_colors[selected_die]
            else:
                trellis.pixels[i, 1] = (0, 0, 0)

    # Only do the roll if both count and die have been selected
    if (selected_count &gt; -1) and (selected_die &gt; -1):
        animate_to(roll(selected_count, sides_per_die[selected_die]),
die_colors[selected_die])
        timeout = time.monotonic() + 5.0
        while len(trellis.pressed_keys) == 0 and time.monotonic() &lt; timeout:
            pass
```

The full code is below. Save it as **code.py** on your Trellis's **CIRCUITPY** drive.

```
# SPDX-FileCopyrightText: 2018 Dave Astels for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""
Dice roller for the NeoTrellisM4

Adafruit invests time and resources providing this open source code.
Please support Adafruit and open source hardware by purchasing
products from Adafruit!

Written by Dave Astels for Adafruit Industries
Copyright (c) 2018 Adafruit Industries
Licensed under the MIT license.

All text above must be included in any redistribution.
"""

# pylint: disable=global-statement

import math
import time
import random
import board
import adafruit_trellism4
import adafruit_adxl34x
import busio

# Set up the trellis and accelerometer

trellis = adafruit_trellism4.TrellisM4Express()

i2c = busio.I2C(board.ACCELEROMETER_SCL, board.ACCELEROMETER_SDA)
accelerometer = adafruit_adxl34x.ADXL345(i2c)


# a 3-wide, 4-tall fonts for 0-9
# Each string is a row with a space being unlit, and anything else being lit.

number_patterns = [
    [" * ", "* *", "* *", "***"],         # 0
    [" * ", "** ", " * ", "***"],         # 1
    ["** ", "  *", " * ", "***"],         # 2
    ["** ", "  *", " **", "***"],         # 3
    ["* *", "* *", "***", "  *"],         # 4
    ["***", "*  ", " **", "***"],         # 5
    [" **", "*  ", "***", "***"],         # 6
    ["***", "  *", " * ", " * "],         # 7
    [" * ", "* *", "***", "***"],         # 8
```

```
    [" * " ,"* *", " **", "  *"]          # 9
]

sides_per_die = [4, 6, 8, 10, 12, 20]
die_colors = [
    (255, 0, 0),                          # d4
    (0, 255, 0),                          # d6
    (0, 0, 255),                          # d8
    (255, 255, 0),                        # d10
    (0, 255, 255),                        # d12
    (255, 0, 255)]                        # d20

def display_digit(number, offset, color, force_zero):
    """Display a digit.
    number     -- the number (0-9) to display
    offset     -- the left-most column of the displayed digit
    color      -- the RGB color to use to display the digit
    force_zero -- whether to leave a 0 blank (False) or display it
    """
    bits = number_patterns[number]
    for row in range(4):
        for col in range(3):
            if bits[row][col] == " " or (number == 0 and not force_zero):
                trellis.pixels[col + offset, row] = (0, 0, 0)
            else:
                trellis.pixels[col + offset, row] = color


def display_number(number, color):
    """Display a multi-digit number.
    If the number is > 99, the hundreds digit (1-4) is indicated on the far left
column:
        100 is the top pixel, 400 is the bottom.
    number      -- the number to display
    color       -- the RGB color to use to display the digit
    force_zeros -- whether to leave a 0 in the tens place blank (False) or display
it
    """
    if number >= 500 or number < 0:
        return False
    display_digit(number % 10, 5, color, True)
    display_digit((number // 10) % 10, 1, color, number >= 100)
    hundreds = number // 100
    for h in range(4):
        if h + 1 == hundreds:
            trellis.pixels[0, h] = (255, 255, 255)
        else:
            trellis.pixels[0, h] = (0, 0, 0)
    return True


def animate_to(number, color):
    """Perform an animation (displaying random numbers) before displaying the
requested number.
    number -- the number to eventually display
    color  -- the color to use (indicates the type of dice used)
    """
    for _ in range(10):
        trellis.pixels.fill((0, 0, 0))
        display_number(random.randint(10, 99), color)
        time.sleep(0.1)
    trellis.pixels.fill((0, 0, 0))
    display_number(number, color)


def roll(number, sides):
    """Generate a random dice roll.
    Returns the total of the roll.
    number -- the number of dice to roll
```

```
    sides  -- the number of side on dice to roll (4, 6, 8, 10, 12, 20)
    """
    total = 0
    for _ in range(number):
        total += random.randint(1, sides + 1)
    return total


previous_reading = [None, None, None]
bound = 4.0

def shaken():
    """Detect when the Trellis is shaken.
    See http://www.profoundlogic.com/docs/display/PUI/Accelerometer+Test+for+Shaking
    TL;DR one or more axis experiences a significant (set by bound) change very
quickly
    Returns whether a shake was detected.
    """
    global previous_reading
    result = False
    x, y, z = accelerometer.acceleration
    if previous_reading[0] is not None:
        result = (math.fabs(previous_reading[0] - x) > bound and
                  math.fabs(previous_reading[1] - y) > bound and
                  math.fabs(previous_reading[2] - z) > bound)
    previous_reading = (x, y, z)
    return result


selected_count = -1
selected_die = -1

while True:
    trellis.pixels.fill((0, 0, 0))
    previous_reading = accelerometer.acceleration
    while not shaken():
        # update selected count/die if a respective selection has been made
        pressed = trellis.pressed_keys    # Get the pressed buttons
        count_selector = [key for key in pressed if key[1] == 0]   # first row
presses
        die_selector = [key for key in pressed if key[1] == 1]     # second row
presses
        if len(count_selector) > 0:
            selected_count = count_selector[0][0] + 1
        if len(die_selector) > 0 and die_selector[0][0] < 6:   # only 6 types of
dice
            selected_die = die_selector[0][0]

        # update the pixels to reflect the selections
        for i in range(8):
            if i < selected_count:        # Display a bar for the count
                trellis.pixels[i, 0] = (128, 64, 16)
            else:
                trellis.pixels[i, 0] = (0, 0, 0)
            if i == selected_die:         # Just the selected dice, in the
appropriate color
                trellis.pixels[i, 1] = die_colors[selected_die]
            else:
                trellis.pixels[i, 1] = (0, 0, 0)

    # Only do the roll if both count and die have been selected
    if (selected_count > -1) and (selected_die > -1):
        animate_to(roll(selected_count, sides_per_die[selected_die]),
die_colors[selected_die])
        timeout = time.monotonic() + 5.0
        while len(trellis.pressed_keys) == 0 and time.monotonic() < timeout:
            pass
```