adafruit learning system

NeoTrellis M4 Memory Game

Created by Dave Astels

Last updated on 2019-03-27 02:49:33 PM UTC

Overview

Matching pairs of tiles is a very common memory game that has appeared at various time in various motifs. It's also a good game for the NeoTrellis. In this context we can match colors on the buttons.

At the beginning of a game, all buttons are dark. The player presses a dark button (pressing lit buttons is ignored) and it's color is shown. Now the player presses another dark button with the goal of matching the color of the first. If they don't match, both go dark once more. If they do match, both buttons flash and turn white. Once all 16 pairs have been found, the game is over and there's a small light show.

There are 6 different colors used, so there are multiple pairs for each color.

There's also a demo mode that runs when the code starts.

This project was inspired by a comment Yannick Mauray made in Discord during an episode of John Park's Workshop (https://adafru.it/Djf).

Parts

Your browser does not support the video tag.

Adafruit NeoTrellis M4 with Enclosure and Buttons Kit Pack

$59.95
IN STOCK

ADD TO CART

USB cable - USB A to Micro-B

$2.95
OUT OF STOCK

OUT OF STOCK

5V 2.5A Switching Power Supply with 20AWG MicroUSB Cable

$7.50
IN STOCK

ADD TO CART

Code

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 NeoTrellis M4 (https://adafru.it/C-O). You should read it before starting to get the most recent CircuitPython build for the NeoTrellis M4 installed and running along with the required libraries.

Setting up your the NeoTrellis

To get your NeoTrellis M4 setup to run this project's code, first follow these steps:

1) Update the bootloader for NeoTrellis (https://adafru.it/C-N)from the NeoTrellis M4 guide

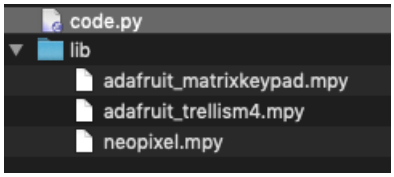
2) Install the latest CircuitPython for NeoTrellis (https://adafru.it/C-O)from the NeoTrellis M4 guide

3) Get the latest 4.0 library pack (https://adafru.it/zB-), unzip it, and drag the libraries you need over into the **/lib** folder on **CIRCUITPY**. The latest library package includes support for NeoTrellis.
https://github.com/adafruit/Adafruit_CircuitPython_Bundle/releases/ (https://adafru.it/zB-)

For this project you will need the following libraries:

- **adafruit_trellism4.mpy**
- **neopixel.mpy**
- **adafruit_matrixkeypad.mpy**

When the libraries and code is installed **CIRCUITPY** should look like:



To complete your setup, click **code.py** in the GitHub window heading below and save the file to your NeoTrellis

CIRCUITPY drive in the main/root directory as **code.py**. The code will start immediately and play the game.

If the game does not play, check your NeoTrellis setup at the top of the page and ensure the three library **.mpy** files are in the **/lib** directory on the **CIRCUITPY** drive.

```
"""
NeoTrellis M4 Express Memory Game

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=stop-iteration-return

import time
import random
import adafruit_trellism4

COLORS = [0xFF0000, 0xFFFF00, 0x00FF00, 0x00FFFF, 0x0000FF, 0xFF00FF]

trellis = adafruit_trellism4.TrellisM4Express(rotation=0)
trellis.pixels.brightness = 0.1
trellis.pixels.fill(0)

pixel_colors = [None] * 32
found_pairs = 0
previously_pressed = set([])
first_pixel = None

key_pressed = None

def index_of(coord):
    x, y = coord
    return y * 8 + x

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))

def cycle_sequence(seq):
    while True:
        for elem in seq:
```

```
            yield elem

def rainbow_lamp(seq):
    g = cycle_sequence(seq)
    while True:
        trellis.pixels.fill(wheel(next(g)))
        yield

def splash():
    rainbow = rainbow_lamp(range(0, 256, 8))
    for _ in range(64):
        next(rainbow)
        time.sleep(0.005)

def assign_colors():
    unassigned = [(x, y) for x in range(8) for y in range(4)]
    while unassigned:
        first_of_pair = random.choice(unassigned)
        unassigned.remove(first_of_pair)
        second_of_pair = random.choice(unassigned)
        unassigned.remove(second_of_pair)
        random_color = random.choice(COLORS)
        pixel_colors[index_of(first_of_pair)] = random_color
        pixel_colors[index_of(second_of_pair)] = random_color

def handle_key(key, _found_pairs, _first_pixel):
    if key is None:
        return _found_pairs, _first_pixel
    key_color = pixel_colors[index_of(key)]
    if key_color is not None:
        trellis.pixels[key] = pixel_colors[index_of(key)]
        time.sleep(0.5)
        if _first_pixel and _first_pixel != key:
            if key_color == pixel_colors[index_of(_first_pixel)]:
                pixel_colors[index_of(_first_pixel)] = None
                pixel_colors[index_of(key)] = None
                for _ in range(5):
                    trellis.pixels[_first_pixel] = 0xFFFFFF
                    trellis.pixels[key] = 0xFFFFFF
                    time.sleep(0.1)
                    trellis.pixels[_first_pixel] = 0x000000
                    trellis.pixels[key] = 0x000000
                    time.sleep(0.1)
                trellis.pixels[_first_pixel] = 0x444444
                trellis.pixels[key] = 0x444444
                return _found_pairs + 1, None
            else:
                trellis.pixels[_first_pixel] = 0x000000
                trellis.pixels[key] = 0x000000
                return _found_pairs, None
        else:
            return _found_pairs, key
    return _found_pairs, None

def check_for_key(last_pressed):
    now_pressed = set(trellis.pressed_keys)
    new_presses = now_pressed - last_pressed
    if new_presses:
        return now_pressed, list(new_presses)[0]
    return now pressed, None
```

```
demo_mode_enabled = True
while True:
    trellis.pixels.fill(0x000000)
    assign_colors()
    found_pairs = 0
    first_pixel = None
    remaining = [(x, y) for x in range(8) for y in range(4)]
    while found_pairs < 16:
        if demo_mode_enabled:
            previously_pressed, key_pressed = check_for_key(previously_pressed)
            if key_pressed:
                demo_mode_enabled = False
                break
            first = random.choice(remaining)
            remaining.remove(first)
            found_pairs, first_pixel = handle_key(first, found_pairs, first_pixel)
            previously_pressed, key_pressed = check_for_key(previously_pressed)
            if key_pressed:
                demo_mode_enabled = False
                break
            c = pixel_colors[index_of(first)]
            match = random.choice([x for x in remaining if pixel_colors[index_of(x)] == c])
            found_pairs, first_pixel = handle_key(match, found_pairs, first_pixel)
            remaining.remove(match)
        else:
            previously_pressed, key_pressed = check_for_key(previously_pressed)
            found_pairs, first_pixel = handle_key(key_pressed, found_pairs, first_pixel)
    if found_pairs == 16:
        splash()
```

Code Walkthrough

Setup

As is always the case with a Python program, we start with imports and initialization:

```
import time
import random
import adafruit_trellism4

COLORS = [0xFF0000, 0xFFFF00, 0x00FF00, 0x00FFFF, 0x0000FF, 0xFF00FF]

trellis = adafruit_trellism4.TrellisM4Express(rotation=0)
trellis.pixels.brightness = 0.1
trellis.pixels.fill(0)

pixel_colors = [None] * 32
found_pairs = 0
previously_pressed = set([])
first_pixel = None
```

`COLORS` contains the possible colors for the pairs. Feel free to add to or change these as you desire, but keep in mind that they should be clearly differentiated.

`pixel_colors` holds the assigned color of each pixel on the NeoTrellis, `found_pairs` tracks how many pairs have been found in the current game, `first_pixel` contains the coordinates of the first pixel of a potential pair (and currently lit in its color), None if there isn't one yet. `previously_pressed` keeps track of what buttons were pressed the last time we checked (so we can notice new presses). We'll see how these are used below.

Helper

There's an `index_of` helper function that converts the `(x, y)` coordinate tuple of a pixel to it's NeoPixel index.

```
def index_of(coord):
    x, y = coord
    return y * 8 + x
```

Completion Splash

Next is a group of functions and generators that together generate a rainbow color splash that happens when all pairs are found. A page (https://adafru.it/Bek) in the Hacking Ikea Lamps with Circuit Playground Express (https://adafru.it/Djg) guide describes how this code works.

```
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))

def cycle_sequence(seq):
    while True:
        for elem in seq:
            yield elem

def rainbow_lamp(seq):
    g = cycle_sequence(seq)
    while True:
        trellis.pixels.fill(wheel(next(g)))
        yield

def splash():
    rainbow = rainbow_lamp(range(0, 256, 8))
    for _ in range(64):
        next(rainbow)
        time.sleep(0.005)
```

Generating Pairs to Find

Now we get into the components of the game itself. The `assign_colors` function assigns random color pairs. It does this by constructing a list of all possible pixel coordinates using a list comprehension. This is the list of pixels remaining unassigned. While it's not empty, we grab a random coordinate and remove it. Then we grab another random coordinate from what's left, and remove it. A random color is picked and applied to the two pixels. This continues until all random pairs of pixels have been assigned.

```
def assign_colors():
    remaining = [(x, y) for x in range(8) for y in range(4)]
    while remaining:
        first = random.choice(remaining)
        remaining.remove(first)
        second = random.choice(remaining)
        remaining.remove(second)
        c = random.choice(COLORS)
        pixel_colors[index_of(first)] = c
        pixel_colors[index_of(second)] = c
```

Handling Input

The crux of the game happens in response to button presses by the player (or the demo mode). The `handle_key` function handles them and implements the game mechanics. It takes the coordinate tuple of the pixel button that was pressed (`key`), and the current value of the `found_pairs` and `first_pixel` variables we created earlier. The function returns new values for those variables.

The first thing that happens is that the key is checked to see if it's an actual press. If `key` is `None`, then it isn't which causes an immediate return, changing nothing.

Since we now know it's a legitimate key press, its color is checked. `None` means it's part of a pair that has been found already, and so it's ignored.

If we get to this point we know that a valid unmatched key has been pressed. The color of the corresponding NeoTrellis NeoPixel is set to the pixel's color for half a second.

Now we check to see if this is the first pixel of a pair, or a possibly second one. Note that we handle the case of pressing the same key twice.

If this is the second in a possible pair and the colors match, it is noted that the pair of pixels has been matched and both are flashed white 5 times before being set to white. Values are returned indicating that another pair was found, and resetting `first_key` to `None`.

If the colors didn't match, both NeoPixels are turned off and the return values clear `first_pixel`.

Finally if this is the first key of a potential pair, it is assigned to `first_pixel` via the function's return values. The final return handles the case where a pixel from a previously matched pair is pressed: `first_pixel` is cleared.

```
def handle_key(key, _found_pairs, _first_pixel):
    if key is None:
        return _found_pairs, _first_pixel
    key_color = pixel_colors[index_of(key)]
    if key_color is not None:
        trellis.pixels[key] = pixel_colors[index_of(key)]
        time.sleep(0.5)
        if _first_pixel and _first_pixel != key:
            if key_color == pixel_colors[index_of(_first_pixel)]:
                pixel_colors[index_of(_first_pixel)] = None
                pixel_colors[index_of(key)] = None
                for _ in range(5):
                    trellis.pixels[_first_pixel] = 0xFFFFFF
                    trellis.pixels[key] = 0xFFFFFF
                    time.sleep(0.1)
                    trellis.pixels[_first_pixel] = 0x000000
                    trellis.pixels[key] = 0x000000
                    time.sleep(0.1)
                trellis.pixels[_first_pixel] = 0x444444
                trellis.pixels[key] = 0x444444
                return _found_pairs + 1, None
            else:
                trellis.pixels[_first_pixel] = 0x000000
                trellis.pixels[key] = 0x000000
                return _found_pairs, None
        else:
            return _found_pairs, key
    return _found_pairs, None
```

Later, in the main game loop, we'll see lines like

```
found_pairs, first_pixel = handle_key(..., found_pairs, first_pixel)
```

Note that both `found_pairs` and `first_pixel` are global variables. Why not just change them directly in the `handle_key` function? That would be considered somewhat bad form. Globals are best avoided in general, and by default Python assumes you want to create a local variable the first time you assign to something inside a function. You can use the `global` statement to tell the compiler that what you really want to do is to assign to the global variable instead of creating a new local:

```
global first_pixel
```

Even though this is available, it's use is bad form and pylint will complain about it. This technique of returning new values for globals can be a handy way to avoid changing globals from inside a function. Additionally, you can see we've taken the next step of passing the values of those globals into function to avoid directly referencing the globals and completely decoupling `handle_key` from them.

Detecting Keypresses

The next function compares the currently pressed keys with those that were pressed last time it checked, returning the one that was new since last time. "One that was new" means an arbitrary one that wasn't pressed before but is now. Because we're using Sets for the comparison, the comparison result (set difference) is converted to a List, and the first one in that list is returned, if there is more than one now pressed the one that is returned depends on the Set to List conversion.

Similarly to `handle_key`, `check_for_key` takes the set of keys previously pressed as an argument and returns its new value along with a key pressed (or `None`)

```
def check_for_key(last_pressed):
    now_pressed = set(trellis.pressed_keys)
    new_presses = now_pressed - last_pressed
    if new_presses:
        return now_pressed, list(new_presses)[0]
    return now_pressed, None
```

Main Game Loop

This is actually a loop within a loop. Each pass through is a game: either demo or playable. The game starts in demo mode. The display is cleared, color pairs assigned, and game variables initialized.

Then the inner loop starts. if left to run to completion it continues until all pairs have been found. What happens in this loop depends on whether it's in demo or play mode.

Play mode is simple, if a button was pressed, it's handled (as detailed above).

Demo mode is a bit more involved. First it picks a random unmatched pixel, handles it, then picks a random pixel of the same color as the first and handles it. That will take care of matching pairs and the NeoPixel manipulation involved. Before each random pick, a keypress is checked for and if there is any, the demo stops and play mode is entered. If the inner loop completes (all 16 pairs matched) a color splash is played. The value of `found_pairs` is checked to see if all were found because the loop can be exited early when a button is pressed in demo mode (to switch into play mode).

```
demo_mode_enabled = True
while True:
    trellis.pixels.fill(0x000000)
    assign_colors()
    found_pairs = 0
    first_pixel = None
    remaining = [(x, y) for x in range(8) for y in range(4)]
    while found_pairs < 16:
        if demo_mode_enabled:
            previously_pressed, key_pressed = check_for_key(previously_pressed)
            if key_pressed:
                demo_mode_enabled = False
                break
            first = random.choice(remaining)
            remaining.remove(first)
            found_pairs, first_pixel = handle_key(first, found_pairs, first_pixel)
            previously_pressed, key_pressed = check_for_key(previously_pressed)
            if key_pressed:
                demo_mode_enabled = False
                break
            c = pixel_colors[index_of(first)]
            match = random.choice([x for x in remaining if pixel_colors[index_of(x)] == c])
            found_pairs, first_pixel = handle_key(match, found_pairs, first_pixel)
            remaining.remove(match)
        else:
            previously_pressed, key_pressed = check_for_key(previously_pressed)
            found_pairs, first_pixel = handle_key(key_pressed, found_pairs, first_pixel)
    if found_pairs == 16:
        splash()
```

That's it. Quite simple code when you look at the individual pieces, but a fairly complex and fun game. One interesting bit is how the same underlying code is used for bot play and demo, differing only by how "pressed" pixels are supplied.