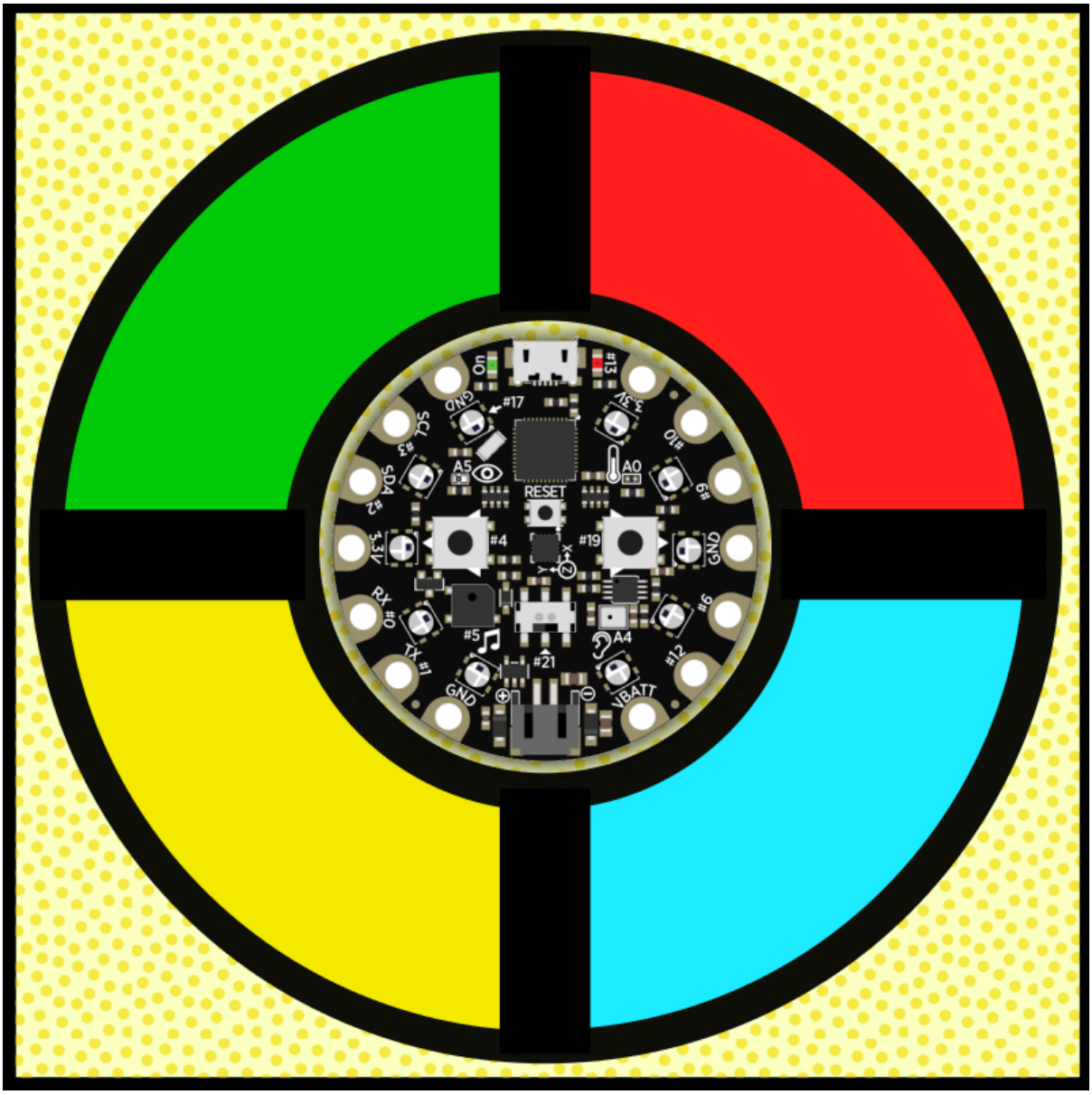
# Circuit Playground Simple Simon

Created by Carter Nelson




https://learn.adafruit.com/circuit-playground-simple-simon

Last updated on 2022-12-30 10:20:59 PM EST

# Table of Contents

# Overview

This is a Circuit Playground rendition of the classic Simon game () from the late 70s and early 80s. The idea is to repeat an ever growing sequence of lights (each with a tone). Here we use the NeoPixels to represent the colors, the capacitive touch pads provide our buttons, and the speaker to play the distinctive tones. The Circuit Playground is even round like the classic game!

# Special Thanks

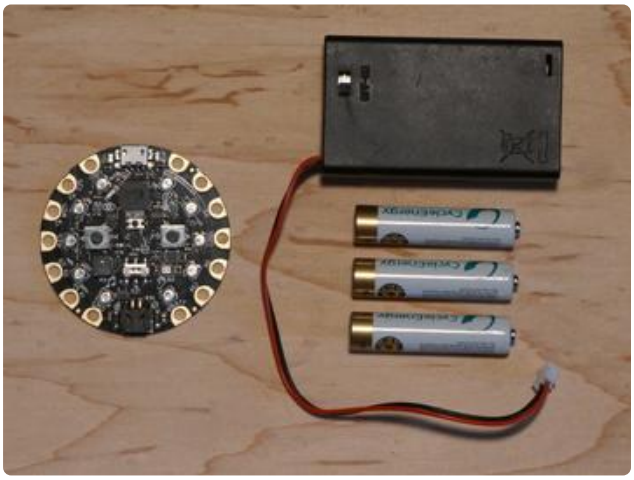
The technical information provided on this page () was very useful for creating the game. Thanks waitingforfriday () for the reverse engineering!

Another version of this game is available in this guide () by Mike Barela.

Also, forum user bbunderson posted another version of Simon you can play on Circuit Playground. Check it out here ().

# Required Parts

This project uses the hardware already included on the Circuit Playground so no additional electronics or soldering are required. If you want to play the game without being attached to your computer, you will also need some batteries and a holder for the batteries.



Circuit Playground
Classic (http://adafru.it/3000)
Express (http://adafru.it/3333)
3 x AAA Battery Holder (http://adafru.it/727)
3 x AAA Batteries (NiMH work great!)

# Before Starting

If you are new to the Circuit Playground, you may want to first read these overview guides.

## Circuit Playground Classic

- Overview ()
- Lesson #0 ()

## Circuit Playground Express

- Overview ()

---

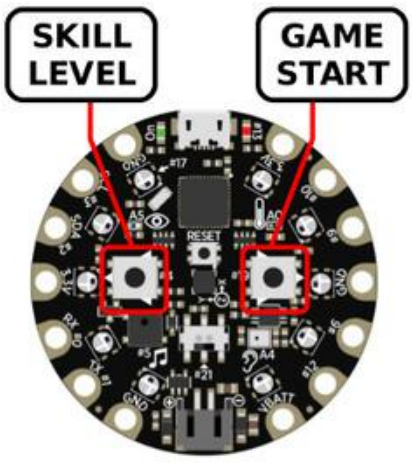
# Playing the Game

Here's the code for the Simple Simon game. Download the zip file, extract it, and load the sketch to your Circuit Playground using the Arduino IDE.

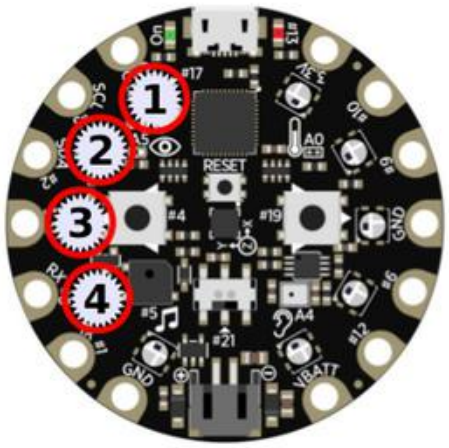
SimpleSimon.zip

# Starting a New Game

You can press the RESET button at anytime to start a new game. When the game starts, you must first select a game difficulty level using the SKILL LEVEL button. Then press the GAME START button to start playing.



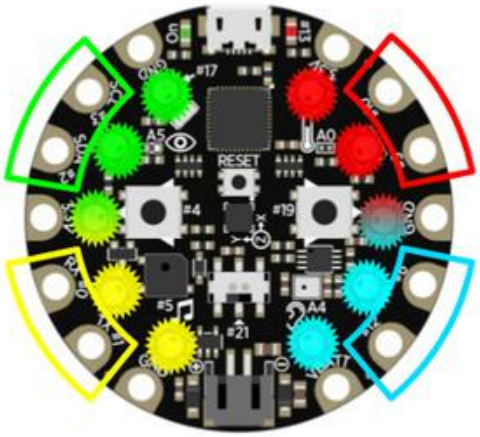

SKILL LEVEL: Use to select game difficulty
GAME START: Use to start a game at selected skill level




The SKILL LEVEL sets the total length (N) of the game sequence.

1 -> N=8
2 -> N=14
3 -> N=20
4 -> N=31

The game is played by repeating the sequence of lights shown using the associated touch pads. There are four different sections of lights, each with its own color and tone.




Use the touch pads shown for each color. You can press one or both, it doesn't matter.

The sequence starts with only one light. It then grows by one light each time you successfully complete the sequence. Thus the game starts easy, and gets more difficult as it progresses. To make it even more difficult, the sequence playback speeds up when the sequence becomes longer.

# Winning

To game is won by completing the entire sequence for the chosen skill level without making any mistakes. A super special sound and light show called a 'razz' is displayed for the victor.

# Losing

There are two was to lose the game:

1. Pick an incorrect light in the sequence.
2. Take too long (>3 secs) to pick the next light.

If either of the above happens, the game ends. The light that was suppose to be next is shown.

# Playing Again

Simply press RESET to start a new game.

# Example Game Play

The video below demonstrates selecting a skill level, starting a new game, general game play, and successfully completing the sequence.

---

# Code Walk Through

Let's go through the code chunk by chunk to see how it works.

## chooseSkillLevel()

```
void chooseSkillLevel() {
  while (!CircuitPlayground.rightButton()) {
    if (CircuitPlayground.leftButton()) {
      skillLevel = skillLevel + 1;
      if (skillLevel &gt; 4) skillLevel = 1;

      CircuitPlayground.clearPixels();
      for (int p=0; p&lt;skillLevel; p++) {
        CircuitPlayground.setPixelColor(p, 0xFFFFFF);
      }

      delay(DEBOUNCE);
    }
  }
}
```

The outer `while()` loop looks for the right button press, which starts the new game. Inside the loop is an `if` statement that looks for the left button press and increments

the skill level for each press. The value is checked and reset back to 1 if it exceeds the max level, which is 4. The current level is shown using the first 4 NeoPixels.

### newGame()

```
void newGame() {
  // Set game sequence length based on skill level
  switch (skillLevel) {
    case 1:
      sequenceLength = 8;
      break;
    case 2:
      sequenceLength = 14;
      break;
    case 3:
      sequenceLength = 20;
      break;
    case 4:
      sequenceLength = 31;
      break;
  }

  // Populate the game sequence
  for (int i=0; i&lt;sequenceLength; i++) {
    simonSequence[i] = random(4);
  }

  // We start with the first step in the sequence
  currentStep = 1;
}
```

The entire game sequence is created before the game is started. The game sequence length is set based on the skill level in the `switch` block. The game sequence is then filled with random numbers between 0-3 to represent one of the four game pads. The currentStep is set to the starting position of 1.

### indicateButton()

```
void indicateButton(uint8_t b, uint16_t duration) {
  CircuitPlayground.clearPixels();
  for (int p=0; p&lt;3; p++) {
    CircuitPlayground.setPixelColor(simonButton[b].pixel[p], simonButton[b].color);
  }
  CircuitPlayground.playTone(simonButton[b].freq, duration);
  CircuitPlayground.clearPixels();
}
```

This function displays the NeoPixels associated with the "button" passed in by `b`, plays the tone associated with this "button" for the length of time passed in by `duration`. After the tone is played, the lights are turned off.

## showSequence()

```
void showSequence() {
  // Set tone playback duration based on current sequence length
  uint16_t toneDuration;
  if (currentStep&lt;=5) {
    toneDuration = 420;
  } else if (currentStep&lt;=13) {
    toneDuration = 320;
  } else {
    toneDuration = 220;
  }

  // Play back sequence up to current step
  for (int i=0; i&lt;currentStep; i++) {
    delay(50);
    indicateButton(simonSequence[i], toneDuration);
  }
}
```

The game sequence is played back up to the current game play location. The duration of the tone is set in the `if` statement based on the current location. This sets the play back speed of the sequence. Then, the sequence is played back up to its current location.

## getButtonPress()

```
uint8_t getButtonPress() {
  for (int b=0; b&lt;4; b++) {
    for (int p=0; p&lt;2; p++) {
      if (CircuitPlayground.readCap(simonButton[b].capPad[p]) &gt; CAP_THRESHOLD) {
        indicateButton(b, DEBOUNCE);
        return b;
      }
    }
  }
  return NO_BUTTON;
}
```

The 2 touch pads associated with each "button" are scanned to see if either are currently being touched. If so, it is indicated and its index is returned, otherwise `NO_BUTTON` is returned.

## gameLost()

```
void gameLost(int b) {
  // Show button that should have been pressed
  for (int p=0; p&lt;3; p++) {
    CircuitPlayground.setPixelColor(simonButton[b].pixel[p], simonButton[b].color);
  }

  // Play sad sound :(
  CircuitPlayground.playTone(FAILURE_TONE, 1500);
```

```
  // And just sit here until reset
  while (true) {}
}
```

The "button" that is passed in via `b` is shown, the sad tone is played, and the game ends at an infinite loop.

## gameWon()

```
void gameWon() {
  // Play 'razz' special victory signal
  for (int i=0; i&lt;3; i++) {
    indicateButton(3, 100);  // RED
    indicateButton(1, 100);  // YELLOW
    indicateButton(2, 100);  // BLUE
    indicateButton(0, 100);  // GREEN
  }
  indicateButton(3, 100);  // RED
  indicateButton(1, 100);  // YELLOW

  // Change tones to failure tone
  for (int b=0; b&lt;4; b++) simonButton[b].freq = FAILURE_TONE;

  // Continue for another 0.8 seconds
  for (int i=0; i&lt;2; i++) {
    indicateButton(2, 100);  // BLUE
    indicateButton(0, 100);  // GREEN
    indicateButton(3, 100);  // RED
    indicateButton(1, 100);  // YELLOW
  }

  // Change tones to silence
  for (int b=0; b&lt;4; b++) simonButton[b].freq = 0;

  // Loop lights forever
  while (true) {
    indicateButton(2, 100);  // BLUE
    indicateButton(0, 100);  // GREEN
    indicateButton(3, 100);  // RED
    indicateButton(1, 100);  // YELLOW
  }
}
```

Plays the super special victory 'razz' and ends with an infinite loop of blinking lights.

## setup()

```
void setup() {
  // Initialize the Circuit Playground
  CircuitPlayground.begin();

  // Set play level
  skillLevel = 1;
  CircuitPlayground.clearPixels();
  CircuitPlayground.setPixelColor(0, 0xFFFFFF);
  chooseSkillLevel();

  // Sowing the seeds of random
```

```
  randomSeed(millis());

  // Create game
  newGame();
}
```

Per the rules of Arduino, this is the first thing called. It initializes the Circuit Playground, gets the skill level, seeds the pseudo random number generator, and sets up a new game.

### loop()

```
void loop() {
  // Show sequence up to current step
  showSequence();

  // Read player button presses
  for (int s=0; s&lt;currentStep; s++) {
    startGuessTime = millis();
    guess = NO_BUTTON;
    while ((millis() - startGuessTime &lt; GUESS_TIMEOUT) &amp;&amp;
(guess==NO_BUTTON)) {
      guess = getButtonPress();
    }
    if (guess != simonSequence[s]) {
      gameLost(simonSequence[s]);
    }
  }
  currentStep++;
  if (currentStep &gt; sequenceLength) {
    delay(SEQUENCE_DELAY);
    gameWon();
  }
  delay(SEQUENCE_DELAY);
}
```

Per the rules of Arduino, this is called over and over and over and over. First, the current sequence is played back. Then the player interaction is dealt with. For each step in the sequence, a button is read. If it's the wrong button, or the player took too long to press the button, the value is set to `NO_BUTTON`. The button value is then compared to the current sequence value. If it's correct, the game continues. If it's wrong, the game is over. Once the step values exceeds the game sequence length, the game is won.

---

# The Structure of struct

Have fun playing the game for a while before reading this section.

In the previous section, I didn't mention any of the lines of code at the top of the sketch. Most of these are just normal global variables and constants that are fairly common. However, this section of code may be new to you.

```
struct button {
  uint8_t capPad[2];
  uint8_t pixel[3];
  uint32_t color;
  uint16_t freq;
} simonButton[] = {
  { {3,2},    {0,1,2},  0x00FF00,   415 },  // GREEN
  { {0,1},    {2,3,4},  0xFFFF00,   252 },  // YELLOW
  { {12, 6},  {5,6,7},  0x0000FF,   209 },  // BLUE
  { {9, 10},  {7,8,9},  0xFF0000,   310 },  // RED
};
```

This is defining, and initializing, an array ( `simonButton[]` ) of something called a stru ct. A struct comes from the C programming language, but is also supported in C++ and thus Arduino. What it basically does is let us create a new variable that is a collection of information associated with something, like a Simon "button". This let's us pass around and access this information from a single reference. We don't need to pass around each individual piece of information separately.

Talking about structs can get pretty complex, and they're a precursor to the more complex `class` of C++ and the world of OOP. But they are very useful, and we are using it in a fairly simple way in our Simple Simon code. To better understand why a struct is useful, let's look at a more simple example - a 2D point.

# 2D Point Example

A 2D point is represented by an `x` and `y` value. We could create these as separate variables:

```
float x;
float y;
```

and then use them as normal:

```
x = 45.3;
y = -108.6;
```

But what if I wanted to pass this 2D point to a function? Or what if I wanted to create another 2D point? Things start getting a little messy. This is where a struct would help out. Instead of two separate variable, we create a single struct to represent the 2D point, and put the variables in there.

```
struct point {
  float x;
  float y;
}
```

And then to create a new point variable:

```
struct point someLocation;
```

And assign it values:

```
someLocation.x = 45.3;
someLocation.y = -108.6;
```

Now all one needs to do is pass the variable `someLocation` around. The values come along for the ride. And they don't get confused with some other values. If we need another 2D point, we just create another variable, or better yet, an array of them.

The one fancy thing being done in the Simple Simon code is to create and initialize the values of the struct in one step. For our 2D point example, this would look like:

```
struct point {
  float x;
  float y;
} someLocation = {
  45.3, -108.6
}
```

It's just a shortcut way of creating the variable and assigning it some initial values.

# The Simple Simon "button"

Getting back to our Simple Simon game, think about what each of the four "buttons" has. Most obvious is a color. But they also make a sound when they are pressed, and each has its own tone. Since we are trying to implement these on our Circuit Playground, we will use a set of NeoPixels to show off the color. Also, we plan to use capacitive touch for the buttons, with each button having a couple of pads. So, we have the following items for each "button":

- 2 x capacitive touch pads
- 3 x NeoPixels
- a color

- a tone

Hey, that should look familiar. Look again at the struct definition:

```
struct button {
  uint8_t capPad[2];
  uint8_t pixel[3];
  uint32_t color;
  uint16_t freq;
}
```

Just like the list. For each "button" we have the indices for 2 capacitive touch pads, the indices for 3 NeoPixels, the color to use for the NeoPixels, and the frequency of the tone for the button.

We have more than one "button", so we create an array of them and initialize their values:

```
  simonButton[] = {
  { {3,2},    {0,1,2},  0x00FF00,   415 },  // GREEN
  { {0,1},    {2,3,4},  0xFFFF00,   252 },  // YELLOW
  { {12, 6},  {5,6,7},  0x0000FF,   209 },  // BLUE
  { {9, 10},  {7,8,9},  0xFF0000,   310 },  // RED
};
```

Each line is an entry in the array and represents one button. Each item in each line corresponds to the value of the struct and is used to define the specific buttons. For example, the first line defines the GREEN button as having the capacitive touch pads #3 and #2, the NeoPixels #0, #1, and #2, the color green (in hex), and the frequency 415Hz.

Then, if we need the frequency for the tone of the GREEN button, the syntax is:

```
simonButton[0].freq
```

The color for the BLUE button?

```
simonButton[2].color
```

The index of the 2nd NeoPixel for the RED button?

```
simonButton[3].pixel[1]
```

etc. etc.

Hopefully that makes some sense and you see how it is useful. If not, meh, just have fun playing the game....for now.

# So Why Didn't You Just Create a simonButton Class?

Just trying to keep it simple. But this is totally doable, since Arduino uses C++ under the hood.

---

# CircuitPython Version

Here is a CircuitPython () version of the Simple Simon code.

CircuitPython only works on the Circuit Playground Express and Circuit Playground Bluefruit.

```
# SPDX-FileCopyrightText: 2021 Carter Nelson for Adafruit Industries
#
# SPDX-License-Identifier: MIT

# Circuit Playground Express Simple Simon
#
# Game play based on information provided here:
# http://www.waitingforfriday.com/?p=586
#
# Author: Carter Nelson
# MIT License (https://opensource.org/licenses/MIT)
import time
import random
import math
import board
from analogio import AnalogIn
from adafruit_circuitplayground import cp

FAILURE_TONE        = 100
SEQUENCE_DELAY      = 0.8
GUESS_TIMEOUT       = 3.0
DEBOUNCE            = 0.250
SEQUENCE_LENGTH = {
  1 : 8,
  2 : 14,
  3 : 20,
  4 : 31
}
SIMON_BUTTONS = {
  1 : { 'pads':(4,5), 'pixels':(0,1,2), 'color':0x00FF00, 'freq':415 },
  2 : { 'pads':(6,7), 'pixels':(2,3,4), 'color':0xFFFF00, 'freq':252 },
  3 : { 'pads':(1, ), 'pixels':(5,6,7), 'color':0x0000FF, 'freq':209 },
  4 : { 'pads':(2,3), 'pixels':(7,8,9), 'color':0xFF0000, 'freq':310 },
}

def choose_skill_level():
    # Default
    skill_level = 1
```

```
    # Loop until button B is pressed
    while not cp.button_b:
        # Button A increases skill level setting
        if cp.button_a:
            skill_level += 1
            skill_level = skill_level if skill_level < 5 else 1
            # Indicate current skill level
            cp.pixels.fill(0)
            for p in range(skill_level):
                cp.pixels[p] = 0xFFFFFF
            time.sleep(DEBOUNCE)
    return skill_level

def new_game(skill_level):
    # Seed the random function with noise
    with AnalogIn(board.A4) as a4, AnalogIn(board.A5) as a5, AnalogIn(board.A6) as
a6:
        seed  = a4.value
        seed += a5.value
        seed += a6.value

        random.seed(seed)

        # Populate the game sequence
        return [random.randint(1,4) for i in range(SEQUENCE_LENGTH[skill_level])]

def indicate_button(button, duration):
    # Turn them all off
    cp.pixels.fill(0)
    # Turn on the ones for the given button
    for p in button['pixels']:
        cp.pixels[p] = button['color']
    # Play button tone
    if button['freq'] == None:
        time.sleep(duration)
    else:
        cp.play_tone(button['freq'], duration)
    # Turn them all off again
    cp.pixels.fill(0)

def show_sequence(sequence, step):
    # Set tone playback duration based on current location
    if step <= 5:
        duration = 0.420
    elif step <= 13:
        duration = 0.320
    else:
        duration = 0.220

    # Play back sequence up to current step
    for b in range(step):
        time.sleep(0.05)
        indicate_button(SIMON_BUTTONS[sequence[b]], duration)

def cap_map(b):
    if b == 1: return cp.touch_A1
    if b == 2: return cp.touch_A2
    if b == 3: return cp.touch_A3
    if b == 4: return cp.touch_A4
    if b == 5: return cp.touch_A5
    if b == 6: return cp.touch_A6
    if b == 7: return cp.touch_TX

def get_button_press():
    # Loop over all the buttons
    for button in SIMON_BUTTONS.values():
        # Loop over each pad
        for pad in button['pads']:
            if cap_map(pad):
```

```
                indicate_button(button, DEBOUNCE)
                return button
    return None

def game_lost(step):
    # Show button that should have been pressed
    cp.pixels.fill(0)
    for p in SIMON_BUTTONS[sequence[step]]['pixels']:
        cp.pixels[p] = SIMON_BUTTONS[sequence[step]]['color']

    # Play sad sound :(
    cp.play_tone(FAILURE_TONE, 1.5)

    # And just sit here until reset
    while True:
        pass

def game_won():
    # Play 'razz' special victory signal
    for i in range(3):
        indicate_button(SIMON_BUTTONS[4], 0.1)
        indicate_button(SIMON_BUTTONS[2], 0.1)
        indicate_button(SIMON_BUTTONS[3], 0.1)
        indicate_button(SIMON_BUTTONS[1], 0.1)
    indicate_button(SIMON_BUTTONS[4], 0.1)
    indicate_button(SIMON_BUTTONS[2], 0.1)

    # Change tones to failure tone
    for button in SIMON_BUTTONS.values():
        button['freq'] = FAILURE_TONE

    # Continue for another 0.8 seconds
    for i in range(2):
        indicate_button(SIMON_BUTTONS[3], 0.1)
        indicate_button(SIMON_BUTTONS[1], 0.1)
        indicate_button(SIMON_BUTTONS[4], 0.1)
        indicate_button(SIMON_BUTTONS[2], 0.1)

    # Change tones to silence
    for button in SIMON_BUTTONS.values():
        button['freq'] = None

    # Loop lights forever
    while True:
        indicate_button(SIMON_BUTTONS[3], 0.1)
        indicate_button(SIMON_BUTTONS[1], 0.1)
        indicate_button(SIMON_BUTTONS[4], 0.1)
        indicate_button(SIMON_BUTTONS[2], 0.1)

# Initialize setup
cp.pixels.fill(0)
cp.pixels[0] = 0xFFFFFF
skill_level = choose_skill_level()
sequence = new_game(skill_level)
current_step = 1

# Loop forever
while True:
    # Show sequence up to current step
    show_sequence(sequence, current_step)

    # Read player button presses
    for step in range(current_step):
        start_guess_time = time.monotonic()
        guess = None
        while (time.monotonic() - start_guess_time < GUESS_TIMEOUT) and (guess ==
None):
            guess = get_button_press()
        if not guess == SIMON_BUTTONS[sequence[step]]:
```

```
            game_lost(step)

    # Advance the game forward
    current_step += 1
    if current_step > len(sequence):
        game_won()

    # Small delay before continuing
    time.sleep(SEQUENCE_DELAY)
```