CircuitPython CLUE I Ching Caster

Created by Carter Nelson

https://learn.adafruit.com/clue-i-ching-caster

Last updated on 2023-08-29 04:23:48 PM EDT
Table of Contents

Overview 3

I Ching Summary 4
• Yin and Yang
• Trigrams and Hexagrams
• Consulting the I Ching

CircuitPython on CLUE 6
• Set up CircuitPython Quick Start!

Code 9
• Libraries
• Usage
• Customizing
• A Note On Translation

Drawing the Hexagram 13
• Sprite Sheet
• Using a Tile Grid for the Hexagram
• Using Scale to Embiggen
Overview

In this guide we will show you how you can use your Adafruit CLUE to consult the ancient Chinese text known as the I Ching, or Book Of Change. With a simple shake of your CLUE board, you can tap into the mystic forces of the universe and come up with a "random" hexagram that forms the basis of a reading used to consult the I Ching.

So grab a CLUE, load the code, balance your chi, and have some fun! Ommmmmmmm.

[Adafruit CLUE - nRF52840 Express with Bluetooth LE](https://www.adafruit.com/product/4500)

Do you feel like you just don't have a CLUE? Well, we can help with that - get a CLUE here at Adafruit by picking up this sensor-packed development board. We wanted to build some...
I Ching Summary

The I Ching () is an ancient text. As such, there are numerous descriptions of what it is and what it has become. To keep things fun and simple, we are going to use a very brief description here.

The I Ching is comprised of 64 separate entries. Each entry is associated with a hexagram along with a collection of text that reads like poetry. Let's cover a few details.

Yin and Yang

The most fundamental basis for the I Ching is the concept of Yin and Yang - the two intertwined forces that drive the universe. Can't have one without the other. Light and dark. Sweet and sour. Etc.

A common way to represent Yin and Yang is with line segments like this:
These two lines are put together to create further symbols called trigrams, a stack of 3, and hexagrams, a stack of 6.

Trigrams and Hexagrams

The two lines representing Yin and Yang are put together in groups of 3 to comprise 8 trigrams which are considered to comprise the basic elements, like "Earth" and "Mountain". The trigrams are then stacked in groups of two to create the 64 unique hexagrams.

The basis here is essentially binary, 0=Yin and 1=Yang. The prefix "tri" means 3 and "hex" means 6. So the hexagrams are like a 6 bit value ($2^6 = 64$). They are read from bottom to top, so you end up with something like this:

```
1 0 0 1 0 1
```

This binary interpretation is not really used in the I Ching. But it is useful for our code. We don't need any fancy storage mechanism to correlate a specific hexagram with a given number from 0 to 63 (64 total). We can simply use the binary representation of that number to generate the hexagram. For example, let's say we somehow came up with 42 as our "reading". Then we can do something like:
and use each of the bits to generate a corresponding line for the hexagram. Each 0 becomes a dashed line Yin and each 1 becomes a solid line Yang.

Consulting the I Ching

For each of the 64 hexagrams there is an entry in the I Ching with a bunch of vague poetic like text. These 64 entries comprise the I Ching - it's basically a reference text.

Since the I Ching itself is nothing but a collection of 64 entries, how is it used? To "consult" the I Ching you must come up with a number from 1 to 64. And then you look up the I Ching entry for that hexagram number. While this process should be something intrinsically random, the idea is that you should be involved in this randomness and thus give a "reading" that is unique to you. This is referred to a [cleromancy](#).

There are approaches that involve shaking a bundle of sticks, flipping coins, etc. But once you have the value, you then just look it up in the I Ching.

Of course making sense of what the actual words are trying to tell you is another story. Seek out an old mystic living in a cave on a mountain for help.

CircuitPython on CLUE

[CircuitPython](#) is a derivative of [MicroPython](#) designed to simplify experimentation and education on low-cost microcontrollers. It makes it easier than ever to get prototyping by requiring no upfront desktop software downloads. Simply copy and edit files on the CIRCUITPY flash drive to iterate.

The following instructions will show you how to install CircuitPython. If you've already installed CircuitPython but are looking to update it or reinstall it, the same steps work for that as well!

Set up CircuitPython Quick Start!

Follow this quick step-by-step for super-fast Python power :)
Download the latest version of CircuitPython for CLUE from circuitpython.org

Click the link above to download the latest version of CircuitPython for the CLUE.

Download and save it to your desktop (or wherever is handy).

Plug your CLUE into your computer using a known-good USB cable.

A lot of people end up using charge-only USB cables and it is very frustrating! So make sure you have a USB cable you know is good for data sync.

Double-click the Reset button on the top (magenta arrow) on your board, and you will see the NeoPixel RGB LED (green arrow) turn green. If it turns red, check the USB cable, try another USB port, etc. Note: The little red LED next to the USB connector will pulse red. That's ok!

If double-clicking doesn't work the first time, try again. Sometimes it can take a few tries to get the rhythm right!
You will see a new disk drive appear called CLUEBOOT.

Drag the adafruit-circuitpython-clue-etc.uf2 file to CLUEBOOT.

The LED will flash. Then, the CLUEBOOT drive will disappear and a new disk drive called CIRCUITPY will appear.

If this is the first time you're installing CircuitPython or you're doing a completely fresh install after erasing the filesystem, you will have two files - boot_out.txt, and code.py, and one folder - lib on your CIRCUITPY drive.

If CircuitPython was already installed, the files present before reloading CircuitPython should still be present on your CIRCUITPY drive. Loading CircuitPython will not create new files if there was already a CircuitPython filesystem present.

That's it, you're done! :}

©Adafruit Industries
**Code**

The code for the CLUE I Ching Caster is listed below. You can copy it out and save it to a local file. Once you've done that, you can save the file as `code.py` in your CIRCUITPY folder and it will run automatically when powered up or reset. You'll also need to grab the font file `christopher_done_24.bdf`.

You CIRCUITPY folder should look something like this:

![CIRCUITPY folder content](image)

**Libraries**

You'll need to make sure the following libraries are installed in your CIRCUITPY/lib folder:

- adafruit_apds9960
- adafruit_bitmap_font
- adafruit_bus_device
- adafruit_display_text
- adafruit_register
- adafruit_bmp280.mpy
- adafruit_clue.mpy
- adafruit_lis3mdl.mpy
- adafruit_lsm6ds.mpy
- adafruit_sht31d.mpy
- neopixel.mpy

You can find more info on downloading and installing libraries here:

[Link to CircuitPython Libraries]

©Adafruit Industries
When you're done, your CIRCUITPY/lib folder should look like this:

```python
# SPDX-FileCopyrightText: 2020 Carter Nelson for Adafruit Industries
# SPDX-License-Identifier: MIT

import time
import random
import displayio
from adafruit_bitmap_font import bitmap_font
from adafruit_display_text import label
from adafruit_clue import clue

#--| User Config |-------------------------------
BACKGROUND_COLOR = 0xCFBC17
HEXAGRAM_COLOR   = 0xBB0000
FONT_COLOR       = 0x005500
SHAKE_THRESHOLD  = 20
MELODY = ( (1000, 0.1),  # (freq, duration)
           (1200, 0.1),
           (1400, 0.1),
           (1600, 0.2))

#--| User Config |-------------------------------

# Defined in order treating each hexagram as a 6 bit value.

# Grab the CLUE's display
display = clue.display

displayio.Bitmap(display.width, display.height, 1)

# Background fill
bg_bitmap = displayio.Bitmap(displayio.Palette(1))
bg_palette[0] = BACKGROUND_COLOR
background = displayio.TileGrid(bg_bitmap, pixel_shader=bg_palette)

# Hexagram setup
```
sprite_sheet = displayio.Bitmap(11, 4, 2)
palette = displayio.Palette(2)
palette.make_transparent(0)
palette[0] = 0x000000
palette[1] = HEXAGRAM_COLOR

for x in range(11):
    sprite_sheet[x, 0] = 1 # - - 0 YIN
    sprite_sheet[x, 1] = 0
    sprite_sheet[x, 2] = 1 # --- 1 YANG
    sprite_sheet[x, 3] = 0

sprite_sheet[5, 0] = 0

tile_grid = displayio.TileGrid(sprite_sheet, pixel_shader=palette,
                               width = 1,
                               height = 6,
                               tile_width = 11,
                               tile_height = 2)

hexagram = displayio.Group(x=60, y=15, scale=10)
hexagram.append(tile_grid)

# Hexagram name label
# font credit: https://www.instagram.com/cove703/
font = bitmap_font.load_font("/christopher_done_24.bdf")
font.load_glyphs(b'ABCDEFGHIJKLMNOPQRSTUVWXYZ')
hexname = label.Label(font, text=" "*40, color=FONT_COLOR)
# this will initially hold the "shake for reading" message
hexname.text = " SHAKE\n FOR\n READING"
hexname.anchor_point = (0.5, 0.5)
hexname.anchored_position = (120, 120)

# Set up main display group (splash)
splash = displayio.Group()
display.show(splash)

# Add background and text label
splash.append(background)
splash.append(hexname)

def show_hexagram(number):
    for i in range(6):
        tile_grid[5-i] = (number >> i) & 0x01

def show_name(number):
    hexname.text = HEXAGRAMS[number]
    hexname.anchored_position = (120, 180)

# MAIN CODE
#===================================
print("shake")
wait for shake
while not clue.shake(shake_threshold=SHAKE_THRESHOLD):
    pass

# calibrate the mystic universe
x, y, z = clue.acceleration
random.seed(int(time.monotonic() + abs(x) + abs(y) + abs(z)))

# cast a reading
reading = random.randrange(64)
print("reading = ", reading, HEXAGRAMS[reading])

# play a melody
for note, duration in MELODY:
    clue.play_tone(note, duration)

# prompt to show
```python
display.auto_refresh = False
hexname.text = "GOT IT
PRESS BUTTON
TO SEE"
hexname.anchored_position = (120, 120)
display.auto_refresh = True
while not clue.button_a and not clue.button_b:
    pass

# and then show it
display.auto_refresh = False
splash.append(hexagram)
show_hexagram(reading)
show_name(reading)
display.auto_refresh = True

# hold here until reset
while True:
    pass
```

Usage

Here's how to use.

Save the program as code.py
Press RESET
Should see screen that say "SHAKE FOR READING"

You should hear a little melody play
The screen will change to "PRESS BUTTON TO SEE"
Once you press either A or B button, you should see your hexagram. If you want to try again, press RESET to start over.

Customizing

There are various settings you can change at the top of the code. Look for these lines:

```python
#--| User Config |-------------------------------
BACKGROUND_COLOR = 0xCFBC17
HEXAGRAM_COLOR   = 0xBB0000
FONT_COLOR       = 0x005500
SHAKE_THRESHOLD  = 20
MELODY = ( (1000, 0.1),  # (freq, duration)
          (1200, 0.1),
          (1400, 0.1),
          (1600, 0.2))
#--| User Config |-------------------------------
```

You can change the colors, the shake threshold, as well as the little melody that plays.

A Note On Translation

The original text of the I Ching was written in ancient Chinese. The translation into English used here is based on the text by Kerson and Rosemary Huang (ISBN 978-0894803192).

Drawing the Hexagram

Let's go into some detail about the approach taken to draw the hexagram. This is a handy little trick that maybe you can reuse in one of your projects. It uses a very simple and tiny sprite sheet `Bitmap` to source the two basic line symbols used to generate the hexagram. These are then used in a `TileGrid` to generate the actual hexagram. To render this large on the display, the `displayio Group`'s scale feature is used.
Sprite Sheet

For a more in depth overview of what a sprite sheet is, see this section of the displayio guide:

Look again at the two lines that comprise the hexagram. They are very basic - just a line and dashed line. We can represent these with a simple bitmap like this:

This bitmap is only 11 pixels wide by 4 pixels high. Each blue square is a pixel. The two blank lines are there for padding. The width was chosen for basic aesthetic reasons, generating a nice overall aspect ratio for the lines. This bitmap is so simple that it is generated programmatically. That's what these lines of code are doing:

```python
for x in range(11):
    sprite_sheet[x, 0] = 1 # - - 0 YIN
    sprite_sheet[x, 1] = 0
    sprite_sheet[x, 2] = 1 # --- 1 YANG
    sprite_sheet[x, 3] = 0
    sprite_sheet[5, 0] = 0
```

Using a Tile Grid for the Hexagram

The sprite sheet can then be used along with a TileGrid to generate any of the 64 hexagrams. The TileGrid will be 1 column by 6 rows - one row for each line of the hexagram. Each cell is 11 pixels wide by 2 pixels high, corresponding to one of the symbols from the sprite sheet.

Then it's a simple matter of setting the source index for each cell of the TileGrid to point to one of the two available bitmaps from the sprite sheet. For example, to make the 3rd line down a dashed (Yin) symbol, we would do something like this:
To generate the entire hexagram we just set each cell in the TileGrid to a source from the sprite sheet. Like this:

```
def show_hexagram(number):
    for i in range(6):
        tile_grid[5-i] = (number >> i) & 0x01
```

Some thought has been put into the arrangement of the sprite sheet. By making the dashed (Yin) symbol be first, we can then generate a hexagram from any integer 0 to 63 (64 total) by simply using each of its bits to set the TileGrid's sources.

In the example above, that would be `0b100101 = 37`. Remember, the least significant bit is on the bottom.

That general approach is what is being down in the `show_hexagram` function:
Using Scale to Embiggen

In its raw form, the sprite sheet is only 11 pixels wide by 4 pixels high. The hexagram created from the sprite sheet ends up being 11 pixels wide by 12 pixels high. That's pretty small. So how can we make it bigger on the display? By using `scale`.

The `scale` parameter is an option for the `displayio.Group()` class. It is used as a multiplier when rendering the contents of the Group. Every pixel is drawn "scale" number of times. So if scale=3, then every 1 pixel would become 3x3 pixels on the display.

A scale factor of 10 is used for the hexagram, which is specified in this line of code:

```python
hexagram = displayio.Group(max_size=1, x=60, y=15, scale=10)
```

So the 11x12 hexagram ends up being drawn at an actual size of 110x120 pixels.

There's no fancy smoothing done, which can lead to a very coarse look to the scaled up bitmap. However, since our shapes are blocky rectangles, this works out just fine.