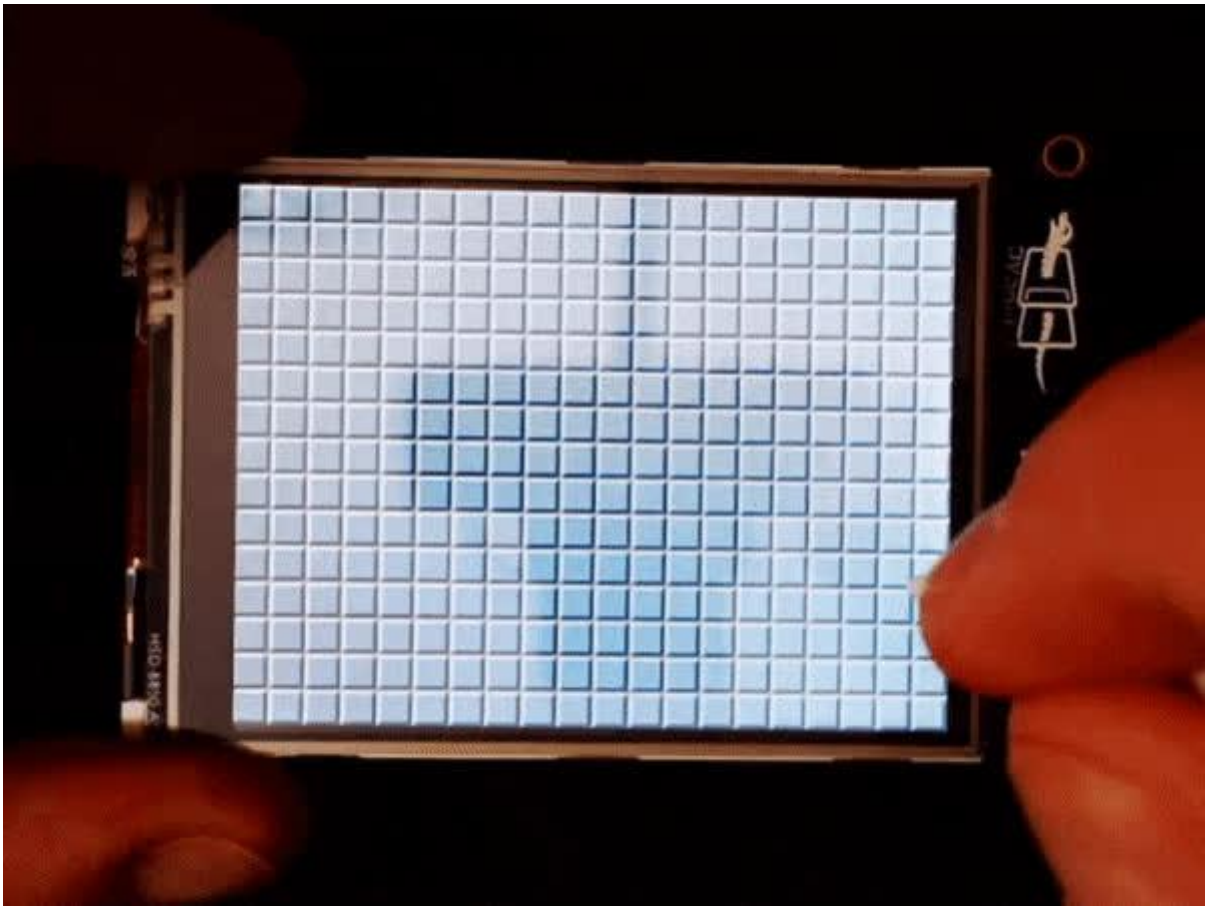
CircuitPython Minesweeper Game

Created by Dave Astels

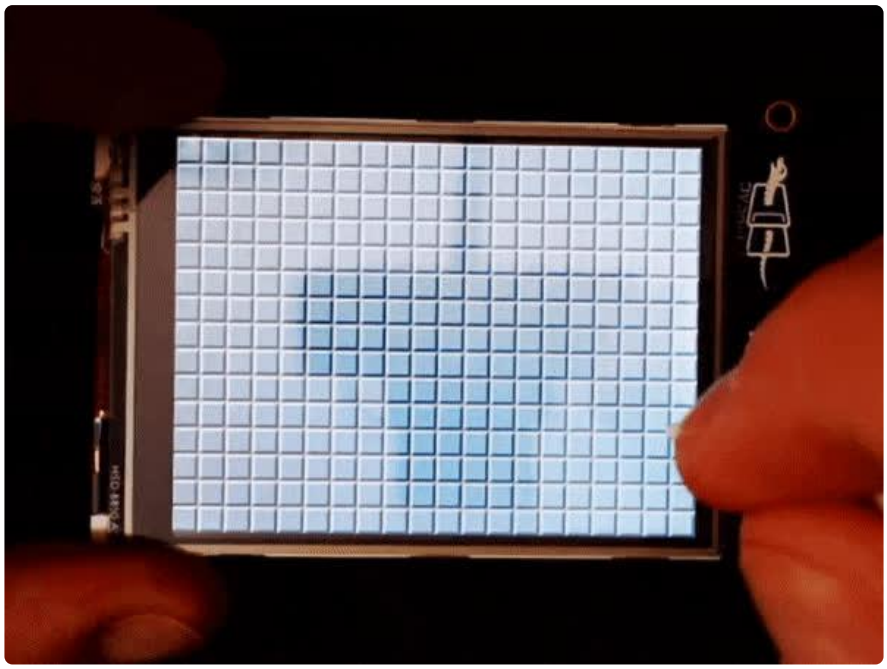


https://learn.adafruit.com/circuitpython-pyportal-minesweeper-game

Last updated on 2022-12-01 03:41:14 PM EST

Table of Contents

Overview



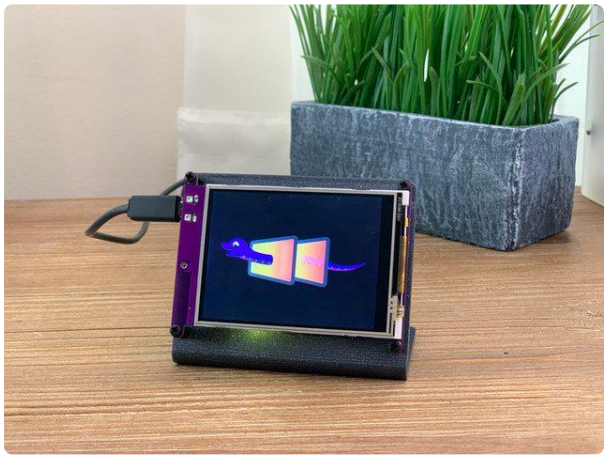
According to www.freeminesweeper.org () (whose game is shown in the guide cover image)

> Minesweeper () is a favourite of office and late night shift workers worldwide.

It's addictive, requires you to think, and every click/tap is fraught with tension. In short, an ideal time waster. In fact, one might think that minesweeper and solitaire were the primary motivation behind Microsoft developing Windows.

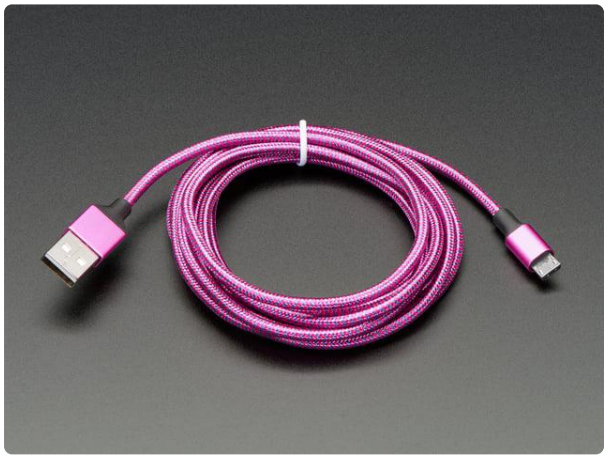
In this guide is a (slightly simplified) implementation of the classic Minesweeper game for the PyPortal and its touchscreen.

Because of the small 16x16 pixel game board squares, a stylus is recommended. One such as that used with a Nintendo DS type system will work fine.



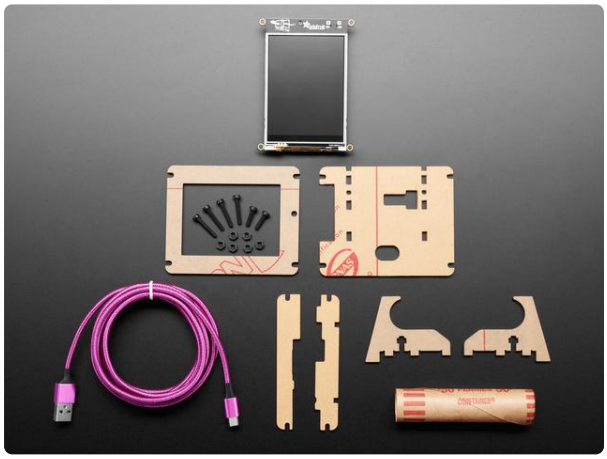
Adafruit PyPortal - CircuitPython Powered Internet Display

PyPortal, our easy-to-use IoT device that allows you to create all the things for the “Internet of Things” in minutes. Make custom touch screen interface...
https://www.adafruit.com/product/4116



Pink and Purple Braided USB A to Micro B Cable - 2 meter long

This cable is super-fashionable with a woven pink and purple Blinka-like pattern! First let's talk about the cover and over-molding. We got these in custom colors,...
https://www.adafruit.com/product/4148



PyPortal Starter Kit

Copy change Why not trick out your fresh new board with some accessories? The Pi 2 is a big deal - a big, big deal. It has an upgraded ARMv7 multicore...
https://www.adafruit.com/product/4303

Setting Up CircuitPython

CircuitPython is a programming language based on Python, one of the fastest growing programming languages in the world. It is specifically designed to simplify experimenting and learning to code on low-cost microcontroller boards. Here is a guide which covers the basics:

- Welcome to CircuitPython! ()

Be sure you have the latest CircuitPython for your board loaded onto your board, as described here (). You will need at least version 4.1.

This project was written with CircuitPython 4.1 or greater in mind.

CircuitPython is easiest to use within the Mu Editor. If you haven't previously used Mu, this guide will get you started ().

Libraries

Plug your board into your computer via a USB cable. Please be sure the cable is a good power+data cable so the computer can talk to the board.

A new disk should appear in your computer's file explorer/finder called CIRCUITPY. This is the place we'll copy the code and code library. If you can only get a drive named PORTALBOOT, load CircuitPython per the guide mentioned above ().

Create a new directory on the CIRCUITPY drive named lib.

Download the latest CircuitPython driver package to your computer using the green button below. Match the library you get to the version of CircuitPython you are using. Save to your computer's hard drive where you can find it.

Go to GitHub to get the latest CircuitPython library bundle

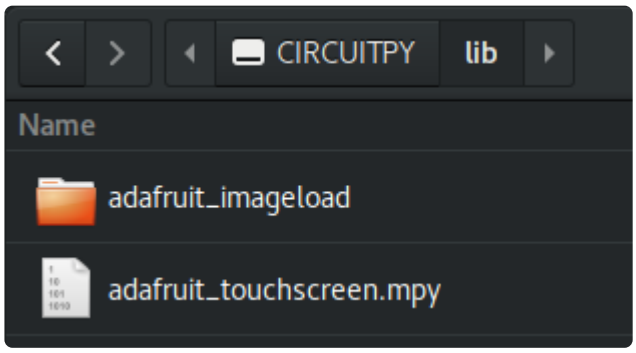
Download the adafruit-circuitpython-bundle-4.x-mpy-*.zip bundle zip file, and unzip a folder of the same name. Inside you'll find a lib folder. You have two options:

- You can add the lib folder to your CIRCUITPY drive. This will ensure you have all the drivers. But it will take a bunch of space on the 8 MB disk
- Add each library as you need it, this will reduce the space usage but you'll need to put in a little more effort.

You need the following libraries for minesweeper. So grab them and copy them into CIRCUITPY/lib now. The other libraries required are part of CircuitPython.

- adafruit_imageload
- adafruit_touchscreen.mpy

Your CIRCUITPY/lib directory should look like:



Once the library files are loaded, the code and other game files will be loaded on the next page.

Code

The code in its entirety is shown below. You should click on Download: Zip in order to get the graphics file SpriteSheet.bmp and sound files (win.wav, lose.wav) used by the game.

Ensure your PyPortal is plugged into your computer via a known good USB cable. The PyPortal should show up as a flash drive called CIRCUITPY.

Open the zip file and copy all the files listed below to CIRCUITPY.

- code.py
- SpriteSheet.bmp
- win.wav
- lose.wav

Note: The boot_out.txt file you may see on the CIRCUITPY drive is generated by CircuitPython and is not part of the needed files for this project.

```
# SPDX-FileCopyrightText: 2019 Dave Astels for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""
PyPortal MineSweeper

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) 2019 Adafruit Industries
Licensed under the MIT license.

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

import time
from random import seed, randint
import board
import digitalio
import displayio
import audioio
try:
    from audioio import WaveFile
except ImportError:
    from audiocore import WaveFile

import adafruit_imageload
import adafruit_touchscreen

seed(int(time.monotonic()))

# Set up audio
```

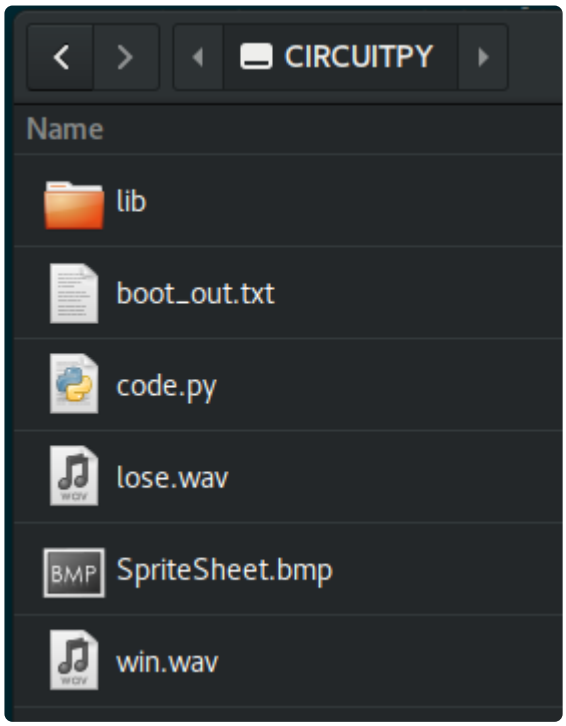
```
speaker_enable = digitalio.DigitalInOut(board.SPEAKER_ENABLE)
speaker_enable.switch_to_output(False)
if hasattr(board, 'AUDIO_OUT'):
    audio = audioio.AudioOut(board.AUDIO_OUT)
elif hasattr(board, 'SPEAKER'):
    audio = audioio.AudioOut(board.SPEAKER)
else:
    raise AttributeError('Board does not have a builtin speaker!')


NUMBER_OF_BOMBS = 15

# Board pieces

OPEN0 = 0
OPEN1 = 1
OPEN2 = 2
OPEN3 = 3
OPEN4 = 4
OPEN5 = 5
OPEN6 = 6
OPEN7 = 7
OPEN8 = 8
BLANK = 9
BOMBDEATH = 10
BOMBFLAGGED = 11
BOMBMISFLAGGED = 12
BOMBQUESTION = 13
BOMB = 14

sprite_sheet, palette = adafruit_imageload.load("/SpriteSheet.bmp",
                                                bitmap=displayio.Bitmap,
                                                palette=displayio.Palette)

display = board.DISPLAY
group = displayio.Group()
touchscreen = adafruit_touchscreen.Touchscreen(board.TOUCH_XL, board.TOUCH_XR,
                                               board.TOUCH_YD, board.TOUCH_YU,
                                               calibration=((9000, 59000),
                                                            (8000, 57000)),
                                               size=(display.width, display.height))
tilegrid = displayio.TileGrid(sprite_sheet, pixel_shader=palette,
                              width=20, height=15,
                              tile_height=16, tile_width=16,
                              default_tile=BLANK)
group.append(tilegrid)



display.show(group)

board_data = bytearray(b'\x00' * 300)

#pylint:disable=redefined-outer-name
def get_data(x, y):
    return board_data[y * 20 + x]

def set_data(x, y, value):
    board_data[y * 20 + x] = value
#pylint:disable=redefined-outer-name

def seed_bombs(how_many):
    for _ in range(how_many):
        while True:
            bomb_x = randint(0, 19)
            bomb_y = randint(0, 14)
            if get_data(bomb_x, bomb_y) == 0:
                set_data(bomb_x, bomb_y, 14)
                break
```

```
def compute_counts():
    """For each bomb, increment the count in each non-bomb square around it"""
    for y in range(15):
        for x in range(20):
            if get_data(x, y) != 14:
                continue                  # keep looking for bombs
            for dx in (-1, 0, 1):
                if x + dx < 0 or x + dx >= 20:
                    continue              # off screen
                for dy in (-1, 0, 1):
                    if y + dy < 0 or y + dy >= 15:
                        continue          # off screen
                    count = get_data(x + dx, y + dy)
                    if count == 14:
                        continue          # don't process bombs
                    set_data(x + dx, y + dy, count + 1)

def reveal():
    for x in range(20):
        for y in range(15):
            if tilegrid[x, y] == BOMBFLAGGED and get_data(x, y) != BOMB:
                tilegrid[x, y] = BOMBMISFLAGGED
            else:
                tilegrid[x, y] = get_data(x, y)

#pylint:disable=too-many-nested-blocks
def expand_uncovered(start_x, start_y):
    number_uncovered = 1
    stack = [(start_x, start_y)]
    while len(stack) > 0:
        x, y = stack.pop()
        if tilegrid[x, y] == BLANK:
            under_the_tile = get_data(x, y)
            if under_the_tile <= OPEN8:
                tilegrid[x, y] = under_the_tile
                number_uncovered += 1
                if under_the_tile == OPEN0:
                    for dx in (-1, 0, 1):
                        if x + dx < 0 or x + dx >= 20:
                            continue              # off screen
                        for dy in (-1, 0, 1):
                            if y + dy < 0 or y + dy >= 15:
                                continue          # off screen
                            if dx == 0 and dy == 0:
                                continue          # don't process where the bomb
                            stack.append((x + dx, y + dy))
    return number_uncovered
#pylint:enable=too-many-nested-blocks

def check_for_win():
    """Check for a complete, winning game. That's one with all squares uncovered
    and all bombs correctly flagged, with no non-bomb squares flaged.
    """
    # first make sure everything has been explored and decided
    for x in range(20):
        for y in range(15):
            if tilegrid[x, y] == BLANK or tilegrid[x, y] == BOMBQUESTION:
                return None               #still ignored or question squares
    # then check for mistagged bombs
    for x in range(20):
        for y in range(15):
            if tilegrid[x, y] == BOMBFLAGGED and get_data(x, y) != BOMB:
                return False               #misflagged bombs, not done
    return True               #nothing unexplored, and no misflagged bombs

#pylint:disable=too-many-branches
# This could be broken apart but I think it's more understandable
# with it all in one place
```

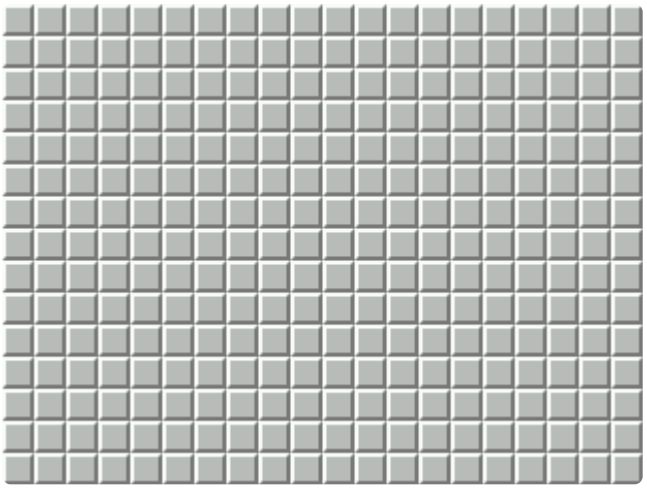
```
def play_a_game():
    number_uncovered = 0
    touch_x = -1
    touch_y = -1
    touch_time = 0
    wait_for_release = False
    while True:
        now = time.monotonic()
        if now >= touch_time:
            touch_time = now + 0.2
            # process touch
            touch_at = touchscreen.touch_point
            if touch_at is None:
                wait_for_release = False
            else:
                if wait_for_release:
                    continue
                wait_for_release = True
                touch_x = max(min([touch_at[0] // 16, 19]), 0)
                touch_y = max(min([touch_at[1] // 16, 14]), 0)
                if tilegrid[touch_x, touch_y] == BLANK:
                    tilegrid[touch_x, touch_y] = BOMBQUESTION
                elif tilegrid[touch_x, touch_y] == BOMBQUESTION:
                    tilegrid[touch_x, touch_y] = BOMBFLAGGED
                elif tilegrid[touch_x, touch_y] == BOMBFLAGGED:
                    under_the_tile = get_data(touch_x, touch_y)
                    if under_the_tile == 14:
                        set_data(touch_x, touch_y, BOMBDEATH) #reveal a red bomb
                        tilegrid[touch_x, touch_y] = BOMBDEATH
                        return False          #lost
                    elif under_the_tile > OPEN0 and under_the_tile <= OPEN8:
                        tilegrid[touch_x, touch_y] = under_the_tile
                    elif under_the_tile == OPEN0:
                        tilegrid[touch_x, touch_y] = BLANK
                        number_uncovered += expand_uncovered(touch_x, touch_y)
                    else:                     #something bad happened
                        raise ValueError('Unexpected value on board')
            status = check_for_win()
            if status is None:
                continue
            return status
#pylint:enable=too-many-branches

def reset_board():
    for x in range(20):
        for y in range(15):
            tilegrid[x, y] = BLANK
            set_data(x, y, 0)
    seed_bombs(NUMBER_OF_BOMBS)
    compute_counts()

def play_sound(file_name):
    try:
        board.DISPLAY.refresh(target_frames_per_second=60)
    except AttributeError:
        board.DISPLAY.wait_for_frame()
    wavfile = open(file_name, "rb")
    wavedata = WaveFile(wavfile)
    speaker_enable.value = True
    audio.play(wavedata)
    return wavfile

def wait_for_sound_and_cleanup(wavfile):
    while audio.playing:
        pass
    wavfile.close()
    speaker_enable.value = False

def win():
```

```
    print('You won')
    wait_for_sound_and_cleanup(play_sound('win.wav'))

def lose():
    print('You lost')
    wavfile = play_sound('lose.wav')
    for _ in range(10):
        tilegrid.x = randint(-2, 2)
        tilegrid.y = randint(-2, 2)
        try:
            board.DISPLAY.refresh(target_frames_per_second=60)
        except AttributeError:
            board.DISPLAY.refresh_soon()
            board.DISPLAY.wait_for_frame()
    tilegrid.x = 0
    tilegrid.y = 0
    wait_for_sound_and_cleanup(wavfile)

while True:
    reset_board()
    if play_a_game():
        win()
    else:
        reveal()
        lose()
    time.sleep(5.0)
```

Looking at the CIRCUITPY drive in your operating system file explorer/finder, you should see a listing similar to this with all the files loaded from this page and the previous page:



The Board



The initial state of the board is shown above. All is hidden, a morass of mystery, intrigue, and danger. Most squares are harmless, but some hide a bomb. Your goal as a player is to find those bombs.

The first thing is to set up the screen structures.

A `TileGrid` is used for the game board, along with a spritesheet that is loaded from a file. Read more about `TileGrid`, spritesheets, and palettes in this guide ().

```
NUMBER_OF_BOMBS = 15

# Board pieces

OPEN0 = 0
OPEN1 = 1
OPEN2 = 2
OPEN3 = 3
OPEN4 = 4
OPEN5 = 5
OPEN6 = 6
OPEN7 = 7
OPEN8 = 8
BLANK = 9
BOMBDEATH = 10
BOMBFLAGGED = 11
BOMBMISFLAGGED = 12
BOMBQUESTION = 13
BOMB = 14
```

```
sprite_sheet, palette = adafruit_imageload.load("/SpriteSheet.bmp",
                                                bitmap=displayio.Bitmap,
                                                palette=displayio.Palette)

display = board.DISPLAY
group = displayio.Group()
touchscreen = adafruit_touchscreen.Touchscreen(board.TOUCH_XL, board.TOUCH_XR,
                                               board.TOUCH_YD, board.TOUCH_YU,
                                               calibration=((9000, 59000),
                                                            (8000, 57000)),
                                               size=(display.width, display.height))
tilegrid = displayio.TileGrid(sprite_sheet, pixel_shader=palette,
                              width=20, height=15,
                              tile_height=16, tile_width=16,
                              default_tile=BLANK)
group.append(tilegrid)



display.show(group)
```

The code above handles what's on the screen. That's what the player sees, but we also need to know where the bombs are and how many bombs are next to each square. There aren't many options, so a byte array works fine. There are two `get`/`set` functions, so we can access it using the same x/y coordinates that we use with the tilegrid.

```
board_data = bytearray(b'\x00' * 300)

#pylint:disable=redefined-outer-name
def get_data(x, y):
    return board_data[y * 20 + x]

def set_data(x, y, value):
    board_data[y * 20 + x] = value
#pylint:disable=redefined-outer-name
```

Next are two functions to initialize the bytearray. The first randomly places bombs. The loop is used to repeated generate a random square until a clear one is found.

```
def seed_bombs(how_many):
    for _ in range(how_many):
        while True:
            bomb_x = randint(0, 19)
            bomb_y = randint(0, 14)
            if get_data(bomb_x, bomb_y) == 0:
                set_data(bomb_x, bomb_y, 14)
                break
```

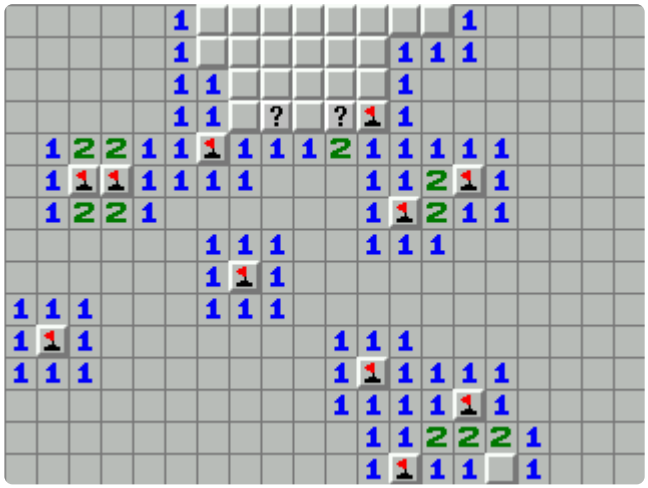
The next function sets the counts in squares adjacent to the bombs. It scans for bombs and when it finds one it increments the value (initialized to 0, see above) in each of the eight squares surrounding the bomb. Of course, if it's another bomb, it's skipped. There's code in there to ignore any squares that are outside the board. This is an issue when a bomb is along the edge of the board.

```
def compute_counts():
    """For each bomb, increment the count in each non-bomb square around it"""
    for y in range(15):
        for x in range(20):
            if get_data(x, y) != 14:
                continue                # keep looking for bombs
            for dx in (-1, 0, 1):
                if x + dx &lt; 0 or x + dx &gt;= 20:
                    continue            # off screen
                for dy in (-1, 0, 1):
                    if y + dy &lt; 0 or y + dy &gt;= 15:
                        continue        # off screen
                    count = get_data(x + dx, y + dy)
                    if count == 14:
                        continue        # don't process bombs
                    set_data(x + dx, y + dy, count + 1)
```

To start a game, the board must be initialized using the above functions. This is handled by the `reset_board` function:

```
def reset_board():
    for x in range(20):
        for y in range(15):
            tilegrid[x, y] = BLANK
            set_data(x, y, 0)
    seed_bombs(NUMBER_OF_BOMBS)
    compute_counts()
```

Playing



The image above shows a game under way. Many squares are revealed, some are marked as bombs (the flags), and some remain a mystery.

We'll start with some support functions.

First is one that is used when the player uncovers an empty square. Starting at that square, more are uncovered, up to squares that are next to bombs. It works much like

the bucket tool in most painting applications. The uncovering spreads until it runs in to a non-empty square (which is uncovered) or a previously uncovered square

The way it works is to use a stack to store squares to be visited. This is initialized to hold the square the player uncovered. The `while` loop continues until the stack is empty, which is the case when we're run out of squares to examine. Each time through the loop, a square is popped from the stack. This is the square to examine. If it's already been uncovered, it gets ignored.

If the square needs to be uncovered, we check what's in the board bytearray at that location. If is adjacent to a bomb (it contains a number), it's uncovered and the next iteration starts. However, if it's an empty square (not adjacent to a bomb) it is uncovered and each of the squares around it (skipping any that are off the edge of the board) are pushed onto the stack. The next iteration then starts.

So, this causes the code to visit a square, then the squares surrounding it, until one with a number (i.e. adjacent to a bomb) is encountered (it's neighbors aren't pushed onto the stack). Once it has visited as many as it can, it's done.

```
def expand_uncovered(start_x, start_y):
    number_uncovered = 1
    stack = [(start_x, start_y)]
    while len(stack) &gt; 0:
        x, y = stack.pop()
        if tilegrid[x, y] == BLANK:
            under_the_tile = get_data(x, y)
            if under_the_tile &lt;= OPEN8:
                tilegrid[x, y] = under_the_tile
                number_uncovered += 1
                if under_the_tile == OPEN0:
                    for dx in (-1, 0, 1):
                        if x + dx &lt; 0 or x + dx &gt;= 20:
                            continue              # off screen
                        for dy in (-1, 0, 1):
                            if y + dy &lt; 0 or y + dy &gt;= 15:
                                continue          # off screen
                            if dx == 0 and dy == 0:
                                continue          # don't process where the bomb
                            stack.append((x + dx, y + dy))
    return number_uncovered
```

Another helper function examines the board to determine if the game is over and whether the player has won or lost.

This is more straight forward.

First, we check every square in the tilegrid to see whether it is still covered or has been marked as questionable. If any are, the game isn't over and `None` is returned.

If all squares have been uncovered or flagged as a bomb, we check if any that have been marked as covering a bomb actually don't. If there are any such squares the game is lost, otherwise it is won. False or True is returned, respectively.

```
def check_for_win():
    """Check for a complete, winning game. That's one with all squares uncovered
    and all bombs correctly flagged, with no non-bomb squares flaged.
    """
    # first make sure everything has been explored and decided
    for x in range(20):
        for y in range(15):
            if tilegrid[x, y] == BLANK or tilegrid[x, y] == BOMBQUESTION:
                return None               #still ignored or question squares
    # then check for mistagged bombs
    for x in range(20):
        for y in range(15):
            if tilegrid[x, y] == BOMBFLAGGED and get_data(x, y) != BOMB:
                return False               #misflagged bombs, not done
    return True               #nothing unexplored, and no misflagged bombs
```

Finally, we have the core game loop.

After initializing some local variables, the loop begins. While the loop runs continuously, it only processes input from the player every 200 milliseconds. This avoids too much noise on the touch input, essentially downsampling.

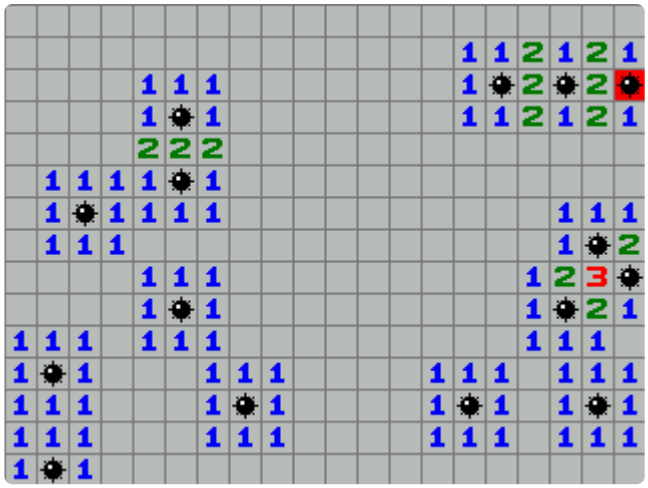
Actions happen on the detection of an initial touch. Subsequent touch detection has to be ignored until a release has been seen (i.e. lack of touch). The `wait_for_release` flag is used to accomplish this. It gets `set` when the first touch is detected. Subsequent touch detections are ignored if the flag is set. Once no touch is detected, the flag is cleared, allowing the next touch to be processed.

Once an initial touch is detected, its coordinates are divided by 16 (the width and height of each tile/square) to give the coordinates of the touched square.

What happens next depends on what is displayed in the tilegrid in the square the player touched.

- If it's blank, set it to a question mark, indicating that the player thinks there might be a bomb there.
- If it's a question mark, set it to a flag, indicating that the player thinks there is a bomb there.
- If it's a flag there's a bit more work involved, depending on what's in that square in the bytearray:
    - If it's a bomb, the game is lost. This is indicated by placing a red square with a bomb at that location and returning `False`

- If it's a tile adjacent to a bomb (i.e. it contains a number) it is simply revealed.
- If it's empty, `expand_uncovered` is called to recursively uncover the space around it.
- Anything else is invalid and causes an error to be raised.



After the touched square is processed (and there hasn't been a game loss due to uncovering a bomb) the board is checked for a win or lose state. If the game isn't over yet, the loop keeps running, otherwise the win/lose state is returned.

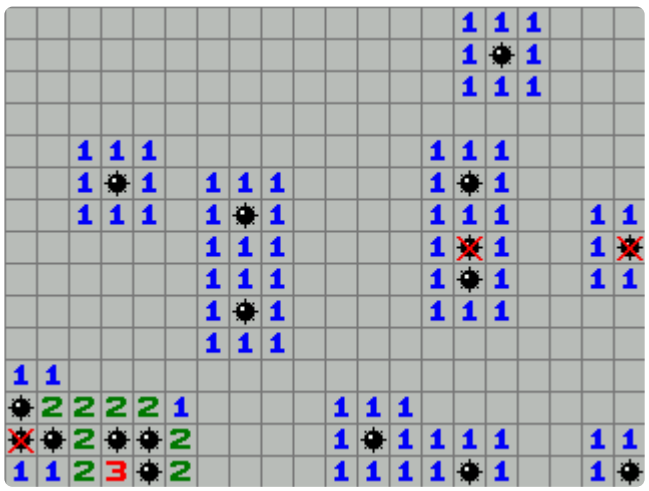
```
def play_a_game():
    number_uncovered = 0
    touch_x = -1
    touch_y = -1
    touch_time = 0
    wait_for_release = False
    while True:
        now = time.monotonic()
        if now &gt;= touch_time:
            touch_time = now + 0.2
            # process touch
            touch_at = touchscreen.touch_point
            if touch_at is None:
                wait_for_release = False
            else:
                if wait_for_release:
                    continue
                wait_for_release = True
                touch_x = max(min([touch_at[0] // 16, 19]), 0)
                touch_y = max(min([touch_at[1] // 16, 14]), 0)
                print('Touched (%d, %d)' % (touch_x, touch_y))
                if tilegrid[touch_x, touch_y] == BLANK:
                    tilegrid[touch_x, touch_y] = BOMBQUESTION
                elif tilegrid[touch_x, touch_y] == BOMBQUESTION:
                    tilegrid[touch_x, touch_y] = BOMBFLAGGED
                elif tilegrid[touch_x, touch_y] == BOMBFLAGGED:
                    under_the_tile = get_data(touch_x, touch_y)
                    if under_the_tile == 14:
                        set_data(touch_x, touch_y, BOMBDEATH)
                        tilegrid[touch_x, touch_y] = BOMBDEATH
                        return False          #lost
                    elif under_the_tile &gt; OPEN0 and under_the_tile &lt;= OPEN8:
```

```
                    tilegrid[touch_x, touch_y] = under_the_tile
                elif under_the_tile == OPEN0:
                    tilegrid[touch_x, touch_y] = BLANK
                    number_uncovered += expand_uncovered(touch_x, touch_y)
                else:                    #something bad happened
                    raise ValueError('Unexpected value on board')
        status = check_for_win()
        if status is None:
            continue
        return status
```

There are a handful of support functions.

First is a function that is called when the game is lost, it reveals the board to the player, if any squares were flagged as a bomb incorrectly, they are shown as a crossed out bomb.

```
def reveal():
    for x in range(20):
        for y in range(15):
            if tilegrid[x, y] == BOMBFLAGGED and get_data(x, y) != BOMB:
                tilegrid[x, y] = BOMBMISFLAGGED
            else:
                tilegrid[x, y] = get_data(x, y)
```



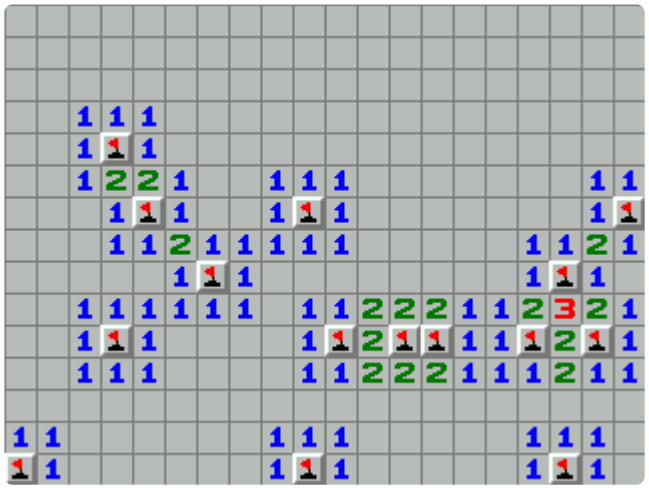
When a game is over, a sound is played. There is one function to start the sound and another to wait for it to finish playing then close it. This allows something else to be done while the sound plays.

```
def play_sound(file_name):
    board.DISPLAY.wait_for_frame()
    wavfile = open(file_name, "rb")
    wavedata = audiocore.WaveFile(wavfile)
    speaker_enable.value = True
    audio.play(wavedata)
    return wavfile

def wait_for_sound_and_cleanup(wavfile):
    while audio.playing:
        pass
```

```
    wavfile.close()
    speaker_enable.value = False
```

If the game was won, there's nothing to do other than play a fanfare:

```
def win():
    wait_for_sound_and_cleanup(play_sound('win.wav'))
```



If the game was lost, a bomb sound is played. While it plays we do a shake effect on the screen by manipulating the location of the tilegrid slightly. Here we see the reason that the sound start/end was split between two functions.

```
def lose():
    wavfile = play_sound('lose.wav')
    for _ in range(10):
        tilegrid.x = randint(-2, 2)
        tilegrid.y = randint(-2, 2)
        display.refresh_soon()
        display.wait_for_frame()
    tilegrid.x = 0
    tilegrid.y = 0
    wait_for_sound_and_cleanup(wavfile)
```

Finally we have the overall loop, that resets the board, plays the game, and signals whether it was won or lost. There's a 5 second delay before another game begins.

```
while True:
    reset_board()
    if play_a_game():
        win()
    else:
        reveal()
        lose()
    time.sleep(5.0)
```

Further Possibilities

This guide covers the basics of a Minesweeper game.

A intro/splash screen could be added as well as a settings screen to set the size of the board and/or the number of bombs.

There's no scoring now, so that's another capability that can be added.

Finally, the ESP32 coprocessor can be used to do something like tweeting your score.