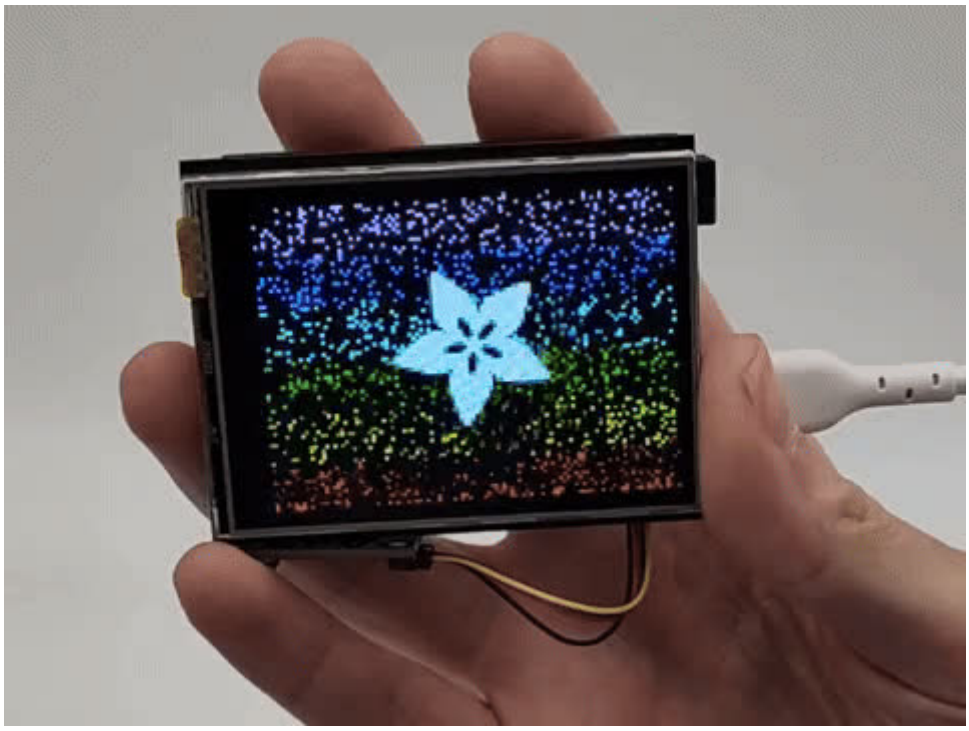
PixelDust Digital Sand for CircuitPython

Created by Tim C



https://learn.adafruit.com/pixeldust-digital-sand-for-circuitpython

Last updated on 2026-09-29 09:57:34 PM UTC

Table of Contents

Overview 3

- Parts - Metro TFT Version
- Parts - Matrix Portal S3 Version

Metro RP2350 + TFT 5

- Parts

Install CircuitPython 6

- CircuitPython Quickstart
- Safe Mode
- Flash Resetting UF2

Assemble 11

Code 12

- Getting the Program's Files
- code.py
- sand_viper.py

Matrix Portal S3 31

- Parts

Install CircuitPython 32

- Set up CircuitPython Quick Start!
- Further Information

Code 34

- Getting the Program's Files
- code.py
- sand.py

Overview

There are several learn guides () that have demonstrations of simulated sand particles. This makes a great demonstration with an accelerometer and LED matrix or small display. Until now, all of these demos required Arduino or similar lower level compiled firmware, or even a full on Raspberry Pi. With CircuitPython shifting gears into Turbo () and the recent addition of Picogame (), it's possible to run projects like this in CircuitPython.

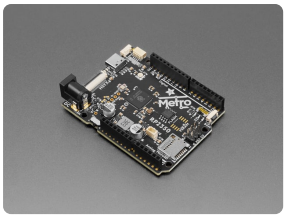
Turbo modules, compiled with Viper, are able to crunch numbers super fast. Numbers like the ones used to simulate gravity and collisions for a lot of particles. Picogame is more efficient at rendering canvas-style graphics of this sort than `displayio` which keeps the demo rendering smoothly on the Metro RP2350 + TFT Shield.

CircuitPython Turbo: Go Blinka!
By phillip torrone
Overview

https://learn.adafruit.com/circuitpython-turbo/overview

There are two versions of the hardware for this guide. Use the parts for your version: A Metro TFT or MatrixPortal S3.

Parts - Metro TFT Version

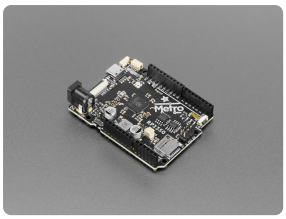


Featured

Adafruit Metro RP2350

Choo! Choo! This is the RP2350 Metro Line, making all station stops at "Dual Cortex M33 mountain", "528K RAM round-about" and "16 Megabytes of Flash...

https://www.adafruit.com/product/6003

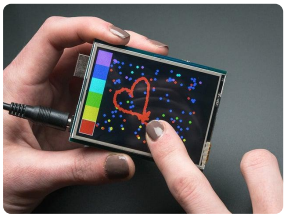


Featured

Adafruit Metro RP2350 with PSRAM

Choo! Choo! This is the RP2350 Metro Line, making all station stops at "Dual Cortex M33 mountain", "528K RAM round-about" and "16 Megabytes of Flash town"...

https://www.adafruit.com/product/6267

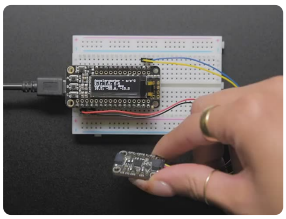


Featured

2.8" TFT Touch Shield for Arduino with Resistive Touch Screen v2

Spice up your Arduino project with a beautiful large touchscreen display shield with built in microSD card connection. This TFT display is big (2.8" diagonal) bright (4 white-LED...

https://www.adafruit.com/product/1651

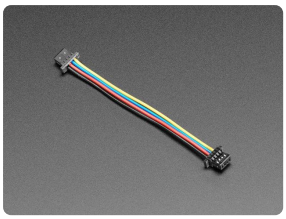


Featured

Adafruit MPU-6050 6-DoF Accel and Gyro Sensor - STEMMA QT Qwiic

I mew, you mew we all mew for IMU! The MPU-6050 is what we call "an oldie but goodie" - this popular triple-axis accelerometer plus gyro combo has been a...

https://www.adafruit.com/product/3886



Featured

STEMMA QT / Qwiic JST SH 4-Pin Cable - 50mm Long

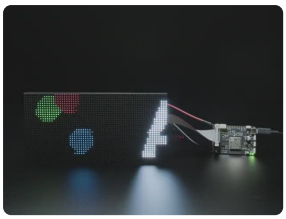
This 4-wire cable is 50mm / 1.9" long and fitted with JST SH female 4-pin connectors on both ends. Compared with the chunkier JST PH these are 1mm pitch instead of 2mm, but...

https://www.adafruit.com/product/4399

Mounting Putty

1x Small Glob

Parts - Matrix Portal S3 Version



Featured

Adafruit Matrix Portal S3 CircuitPython Powered Internet Display

Folks love our wide selection of RGB matrices and accessories for making custom colorful LED displays... and our RGB Matrix...

https://www.adafruit.com/product/5778

Featured

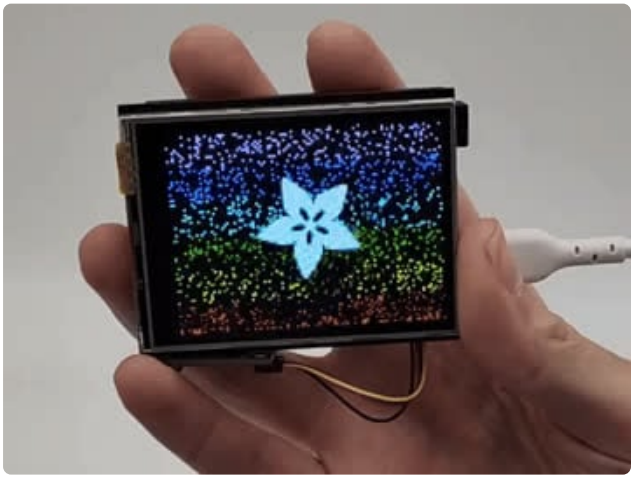
64x64 RGB LED Matrix Panel - 2mm Pitch

Wintertime can be rough in the city. The sky is gray. The weather is unpredictable. So slough off those seasonal blues with some Times Square razzle dazzle from this...

https://www.adafruit.com/product/5362

Or a 64x64 RGB LED Matrix Panel with 2.5mm, or 3mm pitch.

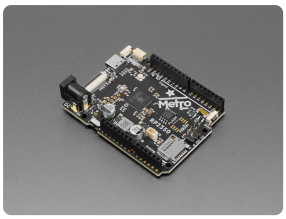
Metro RP2350 + TFT



This section of the guide covers a version of the Pixel Dust demo that runs on the Metro RP2350 with a TFT shield and accelerometer breakout. It simulates more sand particles than the Matrix Portal version.

This project can run on the Metro RP2350 with, or without, PSRAM. The project uses an MPU-6050 accelerometer breakout to determine the direction that gravity should move within the simulation. It can be easily adapted to other accelerometer breakouts by swapping in a different library.

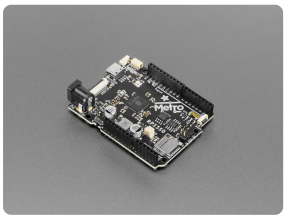
Parts



Adafruit Metro RP2350

Choo! Choo! This is the RP2350 Metro Line, making all station stops at "Dual Cortex M33 mountain", "528K RAM round-about" and "16 Megabytes of Flash...

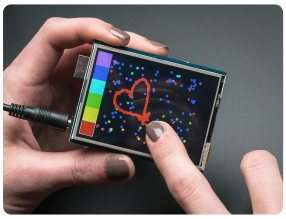
https://www.adafruit.com/product/6003



Adafruit Metro RP2350 with PSRAM

Choo! Choo! This is the RP2350 Metro Line, making all station stops at "Dual Cortex M33 mountain", "528K RAM round-about" and "16 Megabytes of Flash town"...

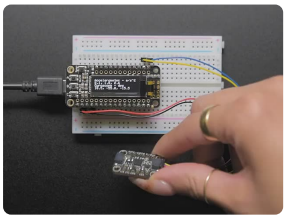
https://www.adafruit.com/product/6267



2.8" TFT Touch Shield for Arduino with Resistive Touch Screen v2

Spice up your Arduino project with a beautiful large touchscreen display shield with built in microSD card connection. This TFT display is big (2.8" diagonal) bright (4 white-LED...

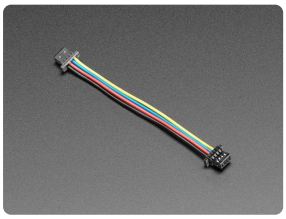
https://www.adafruit.com/product/1651



Adafruit MPU-6050 6-DoF Accel and Gyro Sensor - STEMMA QT Qwiic

I mew, you mew we all mew for IMU! The MPU-6050 is what we call "an oldie but goodie" - this popular triple-axis accelerometer plus gyro combo has been a...

https://www.adafruit.com/product/3886



STEMMA QT / Qwiic JST SH 4-Pin Cable - 50mm Long

This 4-wire cable is 50mm / 1.9" long and fitted with JST SH female 4-pin connectors on both ends. Compared with the chunkier JST PH these are 1mm pitch instead of 2mm, but...

https://www.adafruit.com/product/4399

Install CircuitPython

CircuitPython () is a derivative of MicroPython () designed to simplify experimentation and education on low-cost microcontrollers. It makes it easier than ever to get

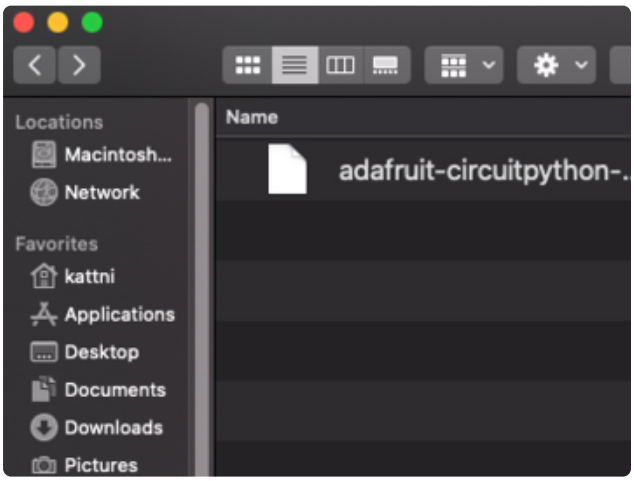
prototyping by requiring no upfront desktop software downloads. Simply copy and edit files on the **CIRCUITPY** drive to iterate.

CircuitPython Quickstart

Follow this step-by-step to quickly get CircuitPython running on your board.

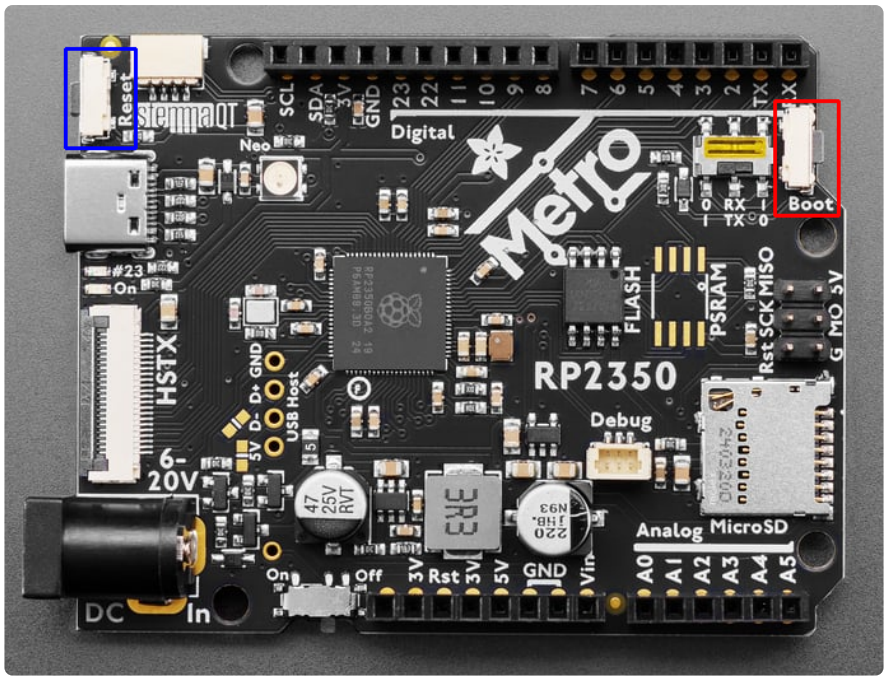
Text emphasized with a blue info:

This project requires both turbo module support and picogame enabled. On the Metro RP2350 Picogame was enabled in PR 11474 on 9/29/26. It's not in any released versions of CircuitPython yet. The button below links directly to the firmware built from #11474.



Click the link above to download the latest CircuitPython UF2 file.

Save it wherever is convenient for you.

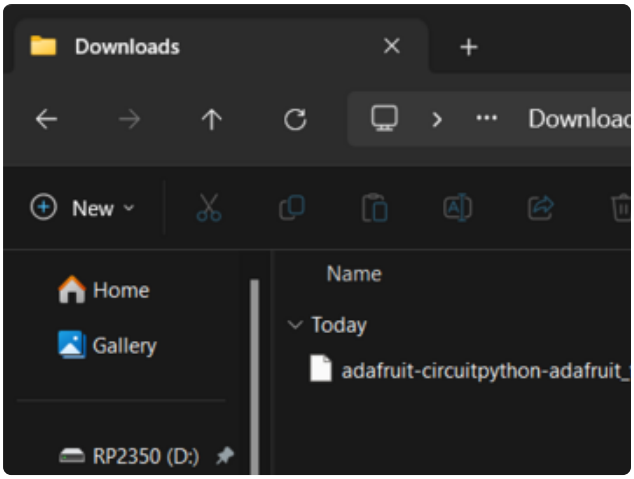


To enter the bootloader, hold down the **BOOT/BOOTSEL button** (highlighted in red above), and while continuing to hold it (don't let go!), press and release the **reset button** (highlighted in red or blue above). **Continue to hold the BOOT/BOOTSEL button until the RP2350 drive appears!**

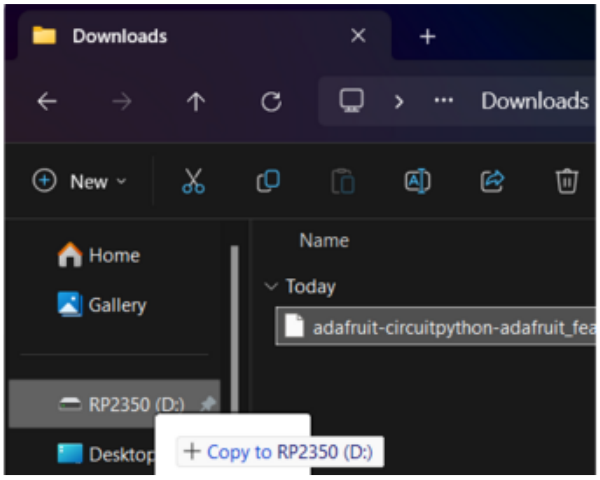
If the drive does not appear, release all the buttons, and then repeat the process above.

You can also start with your board unplugged from USB, press and hold the BOOTSEL button (highlighted in red above), continue to hold it while plugging it into USB, and wait for the drive to appear before releasing the button.

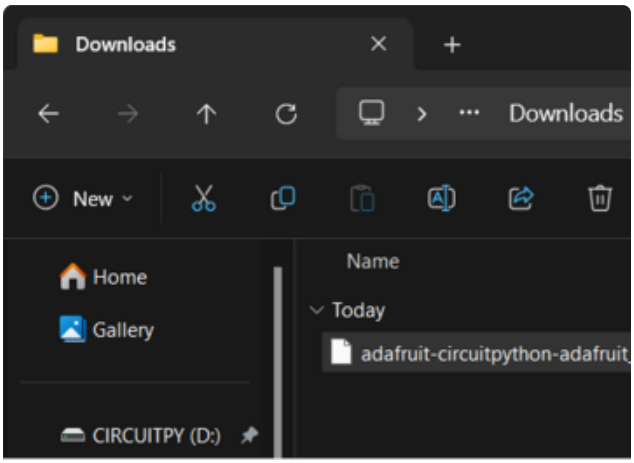
A lot of people end up using charge-only USB cables and it is very frustrating! **Make sure you have a USB cable you know is good for data sync.**



You will see a new disk drive appear called **RP2350**.



Drag the **adafruit-circuitpython-boardname-language-version.uf2** file to **RP2350.**



The **RP2350** drive will disappear and a new disk drive called **CIRCUITPY** will appear.

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

Safe Mode

You want to edit your **code.py** or modify the files on your **CIRCUITPY** drive, but find that you can't. Perhaps your board has gotten into a state where **CIRCUITPY** is read-only. You may have turned off the **CIRCUITPY** drive altogether. Whatever the reason, safe mode can help.

Safe mode in CircuitPython does not run any user code on startup, and disables auto-reload. This means a few things. First, safe mode bypasses any code in **boot.py** (where you can set **CIRCUITPY** read-only or turn it off completely). Second, it does not run the code in **code.py**. And finally, it does not automatically soft-reload when data is written to the **CIRCUITPY** drive.

Therefore, whatever you may have done to put your board in a non-interactive state, safe mode gives you the opportunity to correct it without losing all of the data on the **CIRCUITPY** drive.

Entering Safe Mode

To enter safe mode when using CircuitPython, plug in your board or hit reset (highlighted in red above). Immediately after the board starts up or resets, it waits 1000ms. On some boards, the onboard status LED (highlighted in green above) will blink yellow during that time. If you press reset during that 1000ms, the board will start up in safe mode. It can be difficult to react to the yellow LED, so you may want to think of it simply as a slow double click of the reset button. (Remember, a fast double click of reset enters the bootloader.)

In Safe Mode

If you successfully enter safe mode on CircuitPython, the LED will intermittently blink yellow three times.

If you connect to the serial console, you'll find the following message.

```
Auto-reload is off.
Running in safe mode! Not running saved code.

CircuitPython is in safe mode because you pressed the reset button during boot. Press again to

Press any key to enter the REPL. Use CTRL-D to reload.
```

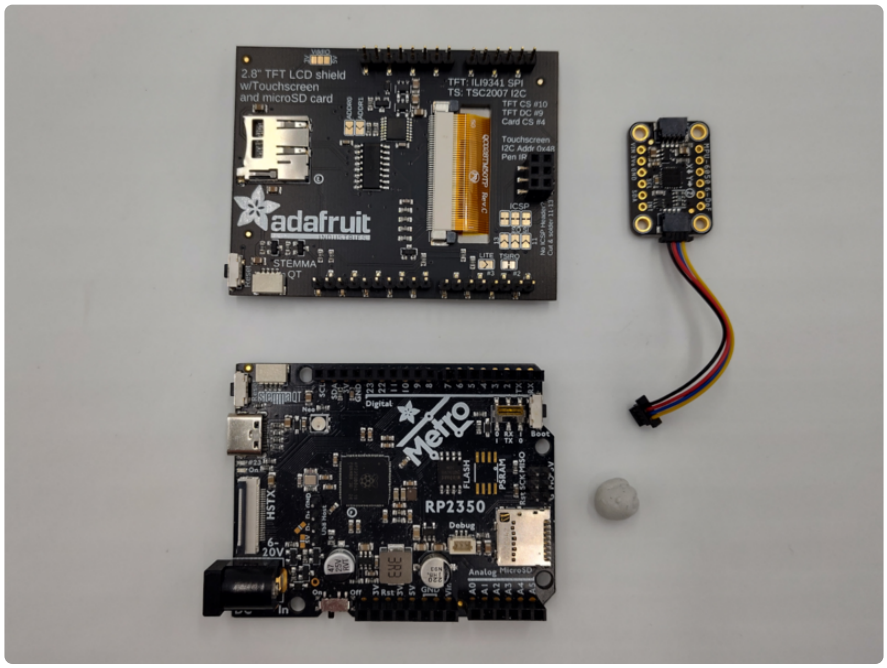
You can now edit the contents of the **CIRCUITPY** drive. Remember, your code will not run until you press the reset button, or unplug and plug in your board, to get out of safe mode.

Flash Resetting UF2

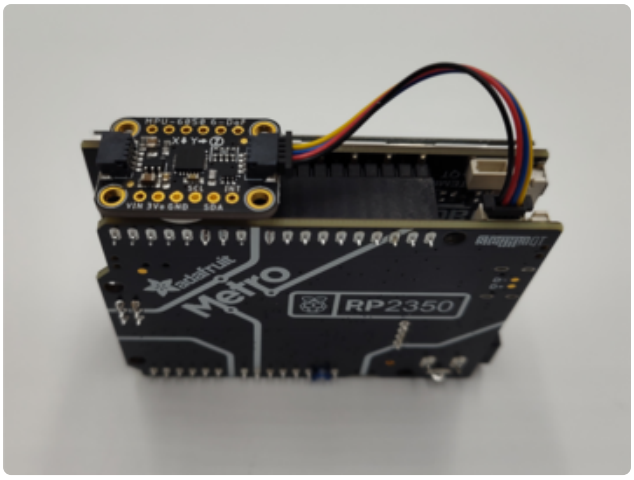
If your board ever gets into a really weird state and CIRCUITPY doesn't show up as a disk drive after installing CircuitPython, try loading this 'nuke' UF2 to RP2350. which

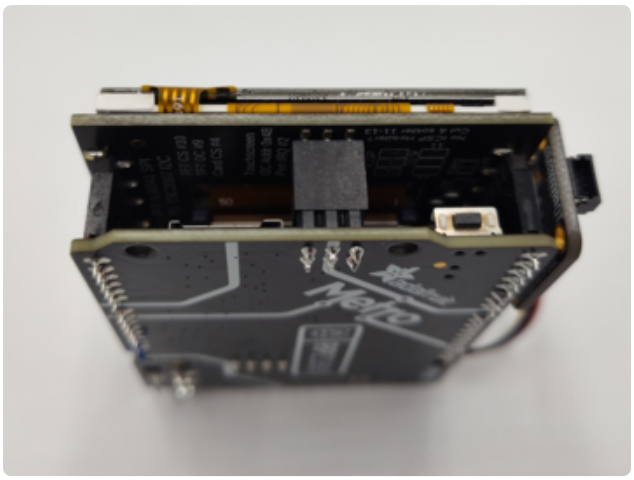
will do a 'deep clean' on your Flash Memory. **You will lose all the files on the board**, but at least you'll be able to revive it! After loading this UF2, follow the steps above to re-install CircuitPython.

Assemble



Assembly for this project is very simple thanks to the plug and play nature of the TFT shield and STEMMA QT connectors for the accelerometer breakout.





First place the TFT shield on top of the Metro RP2350 and gently press it down into place. It should be fully seated and aligned properly on all of the header pins along the two edges and the 2x3 SPI pin connector near the short side, next to the SD card slot and boot button.

Next, connect the accelerometer to the Metro using a small STEMMA QT cable.

Finally, use a small amount of mounting putty to stick the accelerometer to the side of the Metro and TFT shield. Put the putty on the back of the accelerometer breakout, then press it firmly onto the side of the Metro's header pins.

Code

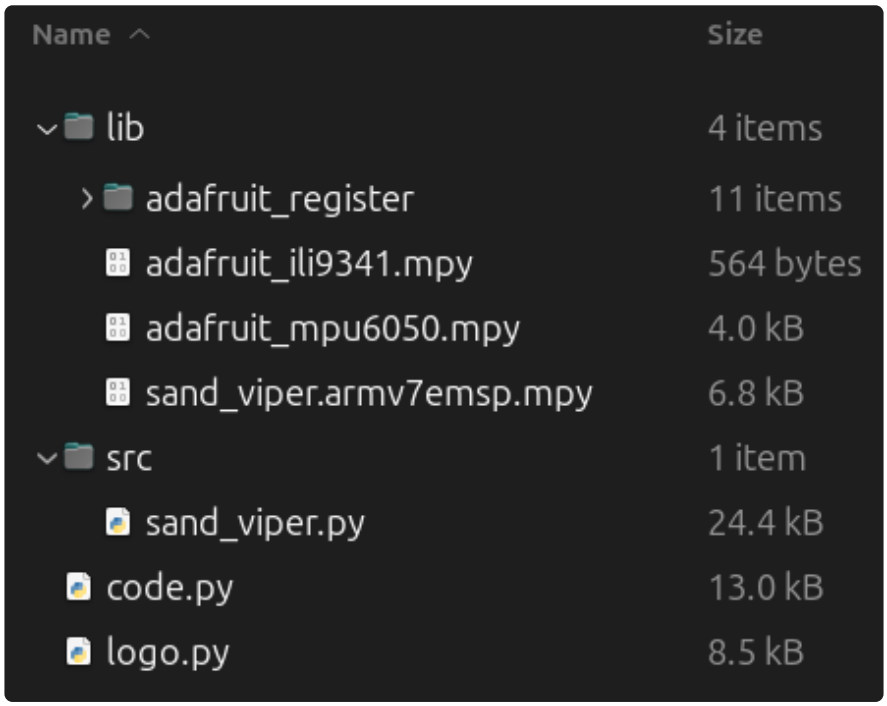
Getting the Program's Files

Viper and native compiled .mpy files are on the cutting edge of support in CircuitPython. The Learn project bundler has not yet been adapted to work with them. Use the green button below to download the project files instead of the usual "Download Project Bundle" button embedded with the code contents.

Connect your board to your computer via a known good data+power USB cable. The board should show up in your File Explorer/Finder (depending on your operating system) as a flash drive named **CIRCUITPY**.

Extract the contents of the zip file, copy the **lib** directory files to **CIRCUITPY/lib**. Copy the **code.py** and **logo.py** files to your **CIRCUITPY** drive. The program should self start.

After loading the project on your device the **CIRCUITPY** drive should contain these files. The **src/** folder is optional. When the code runs it will use the **.mpy** version in the **lib/** directory. Keeping a copy of the plain source code on the device is nice as well if you've got the flash storage space to spare, it makes it easy if you ever want to modify the code and re-compile it.



code.py

This is the main code entry point. It initializes the display and accelerometer. It runs the main loop, deferring to **sand_viper.mpy** to carry out the heavy math for each step of the simulation. This version uses `picogame` to render on the TFT display. It's more efficient than `displayio` for rendering graphics like the ones in this simulation.

If your accelerometer is mounted differently from mine, it may cause the particles to move too much while the Metro is flat. To fix it, you can set `LEVEL` to `None` in the

code. Run it while the Metro RP2350 is sitting flat on a desk or other surface. Look in the serial console to find the calibration values printed from the accelerometer. Copy those values and put them in place as the value of `LEVEL`.

```
# SPDX-FileCopyrightText: 2026 Tim Cocks for Adafruit Industries
#
# SPDX-License-Identifier: MIT
"""Accelerometer sand on a picogame Scene, physics in viper (lib/sand_viper.mpy).
"""

import gc
import math
import random
import time

import board
import displayio
import fourwire
import picogame
import adafruit_ili9341

import sand_viper
import logo

# Grain size: each grain is SCALE x SCALE pixels. The grid is the screen
# divided by SCALE (320x240 at 3 -> 106x80 cells, centered), and the grain
# count is FILL of that, so larger grains also mean fewer of them.
SCALE = 3
FILL = 0.15  # fraction of cells holding a grain
# Grains start as this many horizontal rainbow bands, red at the top through
# blue to magenta at the bottom. 0 scatters the rainbow colors at random.
STRIPES = 6
# Top speed in cells per frame; 1 G tilt adds 1/8 cell per frame each frame,
# so grains accelerate for 8 frames per cell of top speed. Faster grains move
# in one-cell substeps, so they can't jump over one another.
MAX_SPEED = 5
MAX_FPS = 45
SPI_HZ = 75000000
BACKGROUND_RGB = (0, 0, 0)
BACKGROUND_COLOR = picogame.rgb565(*BACKGROUND_RGB)
# The logo is one cell per logo pixel (40x40 cells, 120x120 pixels at SCALE 3),
# drawn in LOGO_RGB blended over BACKGROUND_RGB by the logo's coverage. Grains
# can't enter its solid part, but can cover its faint antialiased edge.
LOGO_RGB = (255, 255, 255)

# The accelerometer's reading, in m/s^2, while resting in the position
# that should count as flat. Every reading is turned by the smallest rotation
# that takes this one to straight along the sensor's z axis, so the sensor can
# be mounted at any angle. (0, 0, 9.8) is a sensor mounted flat. None measures
# it at startup instead (power up with the rig level) and prints the value to
# paste here.
LEVEL = (-9.25, -0.33, -1.32)

# Screen x and y from the leveled accelerometer axes, as (axis index, sign).
# LED_Sand.ino's mapping, (1, -1) and (0, 1), sent the sand uphill with this
```

```
# mounting, so both signs are negated; checked by tilting the rig in every
# direction. For a different mounting, negate an axis that runs the wrong way,
# and swap the axis indexes if left/right tilts move the sand up/down.
SCREEN_X = (1, 1)
SCREEN_Y = (0, -1)

displayio.release_displays()
spi_bus = fourwire.FourWire(
    board.SPI(), command=board.D9, chip_select=board.D10, baudrate=SPI_HZ
)
display = adafruit_ili9341.ILI9341(spi_bus, width=320, height=240)
display.auto_refresh = False
use_dma = False
target = (
    picogame.Display(display)
    if use_dma and picogame.FAST_DISPLAY_SUPPORTED
    else display
)
print("picogame backend:", "DMA Display" if target is not display else "BusDisplay")
strip_bytes = display.width * picogame.STRIP_H * 2  # 2 bytes per RGB565 pixel
scene = picogame.Scene(
    target, bytearray(strip_bytes), bytearray(strip_bytes), background=BACKGROUND_COLOR
)
screen_width, screen_height = display.width, display.height

# -- sand ----------------------------------------------------------------

GRID_WIDTH, GRID_HEIGHT = screen_width // SCALE, screen_height // SCALE
GRAIN_COUNT = int(GRID_WIDTH * GRID_HEIGHT * FILL)


gc.collect()
# The four buffers the viper kernels work on; sand_viper.py describes each.
# grains holds 5 ints per grain: x, y, velocity x, velocity y, color. params
# holds the settings, indexed by the sand_viper.PARAM_* constants.
grains, grid, pixels, params = sand_viper.new(
    GRID_WIDTH,
    GRID_HEIGHT,
    GRAIN_COUNT,
    SCALE,
    BACKGROUND_COLOR,
    seed=random.getrandbits(30),
    backdrop_cells=logo.WIDTH * logo.HEIGHT,
)
params[sand_viper.PARAM_MAX_SPEED] = int(MAX_SPEED * 256)  # 256 grain units per cell


# Alpha-blend the logo over the background.
# step() repaints a cell the logo covers with its color
# from params when a grain leaves it.
def blend(coverage):
    """LOGO_RGB over BACKGROUND_RGB, by coverage from 0 (none) to 255 (solid)."""
    logo_weight = coverage + 1
    background_weight = 257 - logo_weight
    return picogame.rgb565(
        *(
```

```
            (logo_channel * logo_weight + background_channel * background_weight) >> 8
            for logo_channel, background_channel in zip(LOGO_RGB, BACKGROUND_RGB)
        )
    )


logo_width, logo_height = logo.WIDTH, logo.HEIGHT
mask_row_bytes = (
    logo_width + 7
) // 8  # the mask packs 8 cells per byte, leftmost in the top bit
sand_viper.backdrop(
    grid,
    params,
    (GRID_WIDTH - logo_width) // 2,
    (GRID_HEIGHT - logo_height) // 2,
    logo_width,
    logo_height,
    [blend(coverage) for coverage in logo.GRAY],
    bytes(
        logo.MASK[y * mask_row_bytes + x // 8] & (0x80 >> (x & 7))
        for y in range(logo_height)
        for x in range(logo_width)
    ),
)


def rainbow(hue):
    """Fully saturated color at hue in [0, 1): red, yellow, green, cyan, blue, magenta."""
    scaled_hue = hue * 6
    sector = int(scaled_hue)  # which sixth of the color wheel
    rising = int((scaled_hue - sector) * 255)  # how far through that sixth, 0 to 255
    falling = 255 - rising
    red, green, blue = (
        (255, rising, 0),
        (falling, 255, 0),
        (0, 255, rising),
        (0, falling, 255),
        (rising, 0, 255),
        (255, 0, falling),
    )[sector % 6]
    return picogame.rgb565(red, green, blue)


canvas = scene.add(
    picogame.Canvas(GRID_WIDTH * SCALE, GRID_HEIGHT * SCALE, buffer=pixels)
)
canvas.move(
    (screen_width - GRID_WIDTH * SCALE) // 2, (screen_height - GRID_HEIGHT * SCALE) // 2
)

if STRIPES:
    # A grain's row is only known once place() has scattered it, and place()
    # paints with the colors already in grains. So place once for positions,
    # color by row, then restore the random state and place again to paint.
    saved_random_state = params[sand_viper.PARAM_RANDOM_STATE]
    sand_viper.place(grains, grid, pixels, params)
```

```
    for grain in range(GRAIN_COUNT):
        cell_y = grains[5 * grain + 1] >> 8  # grain units to cells: divide by 256
        stripe = cell_y * STRIPES // GRID_HEIGHT
        grains[5 * grain + 4] = rainbow(stripe / max(STRIPES - 1, 1) * 5 / 6)
    params[sand_viper.PARAM_RANDOM_STATE] = saved_random_state
else:
    # place() scatters grains over random cells, so spreading the hues over
    # the grain indices spreads the whole rainbow evenly across the sand.
    for grain in range(GRAIN_COUNT):
        grains[5 * grain + 4] = rainbow(grain / GRAIN_COUNT)
sand_viper.place(grains, grid, pixels, params)

# -- accelerometer ------------------------------------------------------


def make_accelerometer():
    i2c = board.STEMMA_I2C()
    try:
        import adafruit_mpu6050  # pylint: disable=import-outside-toplevel

        mpu6050 = adafruit_mpu6050.MPU6050(i2c)
        mpu6050.accelerometer_range = adafruit_mpu6050.Range.RANGE_4_G
        print("using MPU6050")
        return mpu6050
    except ValueError as error:
        print("no MPU6050:", error)
    try:
        import adafruit_lis3dh  # pylint: disable=import-outside-toplevel

        lis3dh = adafruit_lis3dh.LIS3DH_I2C(i2c, address=0x18)
        lis3dh.range = adafruit_lis3dh.RANGE_4_G
        print("using LIS3DH")
        return lis3dh
    except ValueError as error:
        print("no LIS3DH:", error)
    print("no accelerometer, simulating one")
    return None


accelerometer = make_accelerometer()
GRAVITY = 9.80665  # 1 G in m/s^2


def level_rotation(rest_reading):
    """Rows of the smallest rotation that takes rest_reading to straight
    along the sensor's z axis, +z or -z, whichever is nearer."""
    rest_x, rest_y, rest_z = rest_reading
    length = math.sqrt(rest_x * rest_x + rest_y * rest_y + rest_z * rest_z)
    unit_x, unit_y, unit_z = rest_x / length, rest_y / length, rest_z / length
    target_z = 1.0 if unit_z >= 0 else -1.0
    # Rodrigues' rotation formula, turning the unit vector u to t = (0, 0,
    # target_z): with v = u x t (the axis to turn about, as long as the sine
    # of the angle; its z is 0) and c = u . t >= 0 (the cosine),
    # R = I + [v]x + [v]x^2 / (1 + c).
    axis_x, axis_y = unit_y * target_z, -unit_x * target_z
    factor = 1 / (1 + unit_z * target_z)
```

```
    return (
        (1 - axis_y * axis_y * factor, axis_x * axis_y * factor, axis_y),
        (axis_x * axis_y * factor, 1 - axis_x * axis_x * factor, -axis_x),
        (-axis_y, axis_x, 1 - (axis_x * axis_x + axis_y * axis_y) * factor),
    )


if accelerometer:
    if LEVEL is None:
        totals = [0.0, 0.0, 0.0]
        for _ in range(64):
            reading = accelerometer.acceleration
            totals[0] += reading[0]
            totals[1] += reading[1]
            totals[2] += reading[2]
            time.sleep(0.002)
        LEVEL = (totals[0] / 64, totals[1] / 64, totals[2] / 64)
        print("measured LEVEL = (%.2f, %.2f, %.2f)" % LEVEL)
    # Each row, dotted with a raw reading, gives one leveled axis.
    rotation_x_row, rotation_y_row, rotation_z_row = level_rotation(LEVEL)
    raw_x, raw_y, raw_z = accelerometer.acceleration
    print(
        "leveled reading now (%.2f, %.2f, %.2f) G"
        % tuple(
            (row[0] * raw_x + row[1] * raw_y + row[2] * raw_z) / GRAVITY
            for row in (rotation_x_row, rotation_y_row, rotation_z_row)
        )
    )


def read_acceleration():
    """Return (accel_x, accel_y, jitter_range) in grain units per frame."""
    if accelerometer:
        _raw_x, _raw_y, _raw_z = accelerometer.acceleration
        leveled = (
            rotation_x_row[0] * _raw_x
            + rotation_x_row[1] * _raw_y
            + rotation_x_row[2] * _raw_z,
            rotation_y_row[0] * _raw_x
            + rotation_y_row[1] * _raw_y
            + rotation_y_row[2] * _raw_z,
            rotation_z_row[0] * _raw_x
            + rotation_z_row[1] * _raw_y
            + rotation_z_row[2] * _raw_z,
        )
    else:
        angle = time.monotonic() * 0.4  # simulated gravity turns 0.4 radians per second
        leveled = (
            0.9 * GRAVITY * math.cos(angle),
            -0.9 * GRAVITY * math.sin(angle),
            0.45 * GRAVITY,
        )
    # The original divides raw 4 G-range counts by 256, so 1 G is 32 units.
    _accel_x = int(leveled[SCREEN_X[0]] * SCREEN_X[1] * 32 / GRAVITY)
    _accel_y = int(leveled[SCREEN_Y[0]] * SCREEN_Y[1] * 32 / GRAVITY)
    # Held flat (|z| near 1 G) there is little jitter; tilted up, more.
```

```
    flatness = int(abs(leveled[2]) * 4 / GRAVITY)
    jitter = 1 if flatness >= 3 else 4 - flatness
    # step() adds 0 .. 2 * jitter to each axis; subtracting jitter here
    # centers that on zero.
    return _accel_x - jitter, _accel_y - jitter, 2 * jitter + 1


# -- main loop ---------------------------------------------------------

print(
    f"{screen_width}x{screen_height}, {GRAIN_COUNT} grains on {GRID_WIDTH}x{GRID_HEIGHT} cells"
)
frame_ns = 1000000000 // MAX_FPS
next_frame_ns = time.monotonic_ns()
# Totals since the last report, which is every 2 s.
frames = step_ns = draw_ns = pixels_sent = 0
report_ns = next_frame_ns + 2000000000

while True:
    accel_x, accel_y, jitter_range = read_acceleration()
    params[sand_viper.PARAM_ACCEL_X] = accel_x
    params[sand_viper.PARAM_ACCEL_Y] = accel_y
    params[sand_viper.PARAM_JITTER_RANGE] = jitter_range

    step_start_ns = time.monotonic_ns()
    if sand_viper.step(grains, grid, pixels, params):
        # step() wrote pixels straight into the canvas buffer. Rewriting the
        # dirty rectangle's two corner pixels with their own colors makes the
        # canvas mark exactly that rectangle for the next refresh.
        canvas.pixel(
            params[sand_viper.PARAM_DIRTY_LEFT] * SCALE,
            params[sand_viper.PARAM_DIRTY_TOP] * SCALE,
            params[sand_viper.PARAM_TOP_LEFT_COLOR],
        )
        canvas.pixel(
            params[sand_viper.PARAM_DIRTY_RIGHT] * SCALE - 1,
            params[sand_viper.PARAM_DIRTY_BOTTOM] * SCALE - 1,
            params[sand_viper.PARAM_BOTTOM_RIGHT_COLOR],
        )
    step_end_ns = time.monotonic_ns()
    sent_rect = (
        scene.refresh()
    )  # (left, top, right, bottom) sent to the screen, if anything
    draw_end_ns = time.monotonic_ns()
    if sent_rect:
        pixels_sent += (sent_rect[2] - sent_rect[0]) * (sent_rect[3] - sent_rect[1])

    frames += 1
    step_ns += step_end_ns - step_start_ns
    draw_ns += draw_end_ns - step_end_ns
    if draw_end_ns >= report_ns:
        print(
            "%d fps, step %d us, draw %d us, %d px sent/frame"
            % (
                frames // 2,
                step_ns // frames // 1000,
```

```
                draw_ns // frames // 1000,
                pixels_sent // frames,
            )
        )
        frames = step_ns = draw_ns = pixels_sent = 0
        report_ns += 2000000000

    next_frame_ns += frame_ns
    wait_ns = next_frame_ns - time.monotonic_ns()
    if wait_ns > 0:
        time.sleep(wait_ns / 1e9)
    else:
        next_frame_ns = time.monotonic_ns()  # running behind: don't try to catch up
```

https://github.com/adafruit/Adafruit_Learning_System_Guides/blob/main/CircuitPython_Turbo/sand_simulation/metro_rp2350/code.py

sand_viper.py

This is the code for the Viper module that handles the physics and collision math for the simulation. Viper functions are written as Python, but they are their own weird flavor of Python with an extra set of rules that must be followed.

Local variables and arguments must be `int`, `uint` (32-bit only), `bool`, `ptr`, `ptr8`, `ptr16`, `ptr32`, or `object`. They cannot use `floats` like normal Python numbers do. Parameters get passed to the Viper function as a list of values, and are accessed via their index within that list. There is no overflow protection. Integers will silently wrap, out-of-range indexes can corrupt memory or crash instead of raising a clean `IndexError`. Variables cannot change types. Mixing of typed values with `object` values will fall back to slower runtime calls.

This code gets compiled by `mpy-cross` for the specific architecture of the chip that it will run on. The **sand_viper.mpy** file, included in the bundle, is compiled for the RP2350 architecture `armv7emsp`. The code cannot be directly copied to another type of board (re. ESP-32 board).

```
# SPDX-FileCopyrightText: 2026 Tim Cocks for Adafruit Industries (algorithm)
# SPDX-FileCopyrightText: 2017 Phillip Burgess for Adafruit Industries (algorithm)
# SPDX-License-Identifier: MIT
"""Accelerometer 'sand' kernels as @micropython.viper, ported from LED_Sand.ino.

Positions are in grain units, 256 to a cell as in the original, so grains move
in sub-cell steps. Everything is integer because viper has no float type; the
only divisions are in the velocity clamp and in splitting up fast moves,
and only grains above one cell per frame take either path.

All state lives in caller-owned buffers, so the kernels allocate nothing.
new() returns them in this order:

  grains  array('i')   5 ints per grain: x, y, velocity x, velocity y, color
```

```
  grid    bytearray    one byte per cell, row by row: 0 = empty, 1 = grain,
                       2 = obstacle
  pixels  bytearray    RGB565 pixels, cell_size x cell_size per cell; hand
                       this to picogame.Canvas(buffer=...)
  params  array('i')   settings in and results out, one int per PARAM_* slot
                       below, then the backdrop's cell colors

The kernels take buffers rather than Python objects because viper reads and
writes a ptr8/ptr16/ptr32 argument directly as memory, with no object
overhead. That is also why the settings travel in one int array instead of
as separate arguments: a PARAM_* name is a const(), so viper compiles
params[PARAM_GRID_WIDTH] to a load from a fixed offset.

The kernels write pixels directly, behind the Canvas's back. step() reports
the rectangle it touched (the "dirty" rectangle) so the caller can tell the
Canvas to redraw it.

A backdrop is a picture, one color per cell, that shows wherever no
grain is, in place of the background color: backdrop() sets it up, and marks
the cells of it that grains can't enter (as Adafruit_PixelDust's setPixel()
does). Since grains never enter an obstacle, the kernels never repaint one.
"""

import array

import micropython
from micropython import const

# pylint: disable=undefined-variable, too-many-statements, too-many-locals, too-many-nested-blocks, too-
many-branches

# The slots of params, as indexes into it.
PARAM_GRID_WIDTH = const(0)  # grid width in cells
PARAM_GRID_HEIGHT = const(1)  # grid height in cells
PARAM_GRAIN_COUNT = const(2)  # number of grains, < grid width * height
PARAM_CELL_SIZE = const(3)  # pixels per cell edge
PARAM_BACKGROUND = const(4)  # background color
PARAM_RANDOM_STATE = const(5)  # random number generator's state, updated in place
PARAM_ACCEL_X = const(6)  # acceleration this frame, grain units per frame
PARAM_ACCEL_Y = const(7)
PARAM_JITTER_RANGE = const(8)  # each axis's velocity gets 0 .. jitter range - 1 added
PARAM_DIRTY_LEFT = const(9)  # out: dirty cells are [left, right) x [top, bottom)
PARAM_DIRTY_TOP = const(10)
PARAM_DIRTY_RIGHT = const(11)
PARAM_DIRTY_BOTTOM = const(12)
PARAM_TOP_LEFT_COLOR = const(13)  # out: pixel color at the dirty rect's top-left corner
PARAM_BOTTOM_RIGHT_COLOR = const(14)  # out: pixel color at its bottom-right corner
PARAM_MAX_SPEED = const(15)  # top speed, grain units per frame; 256 is one cell
PARAM_BACKDROP_X = const(16)  # backdrop's top-left cell; may be off the grid
PARAM_BACKDROP_Y = const(17)
PARAM_BACKDROP_WIDTH = const(18)  # backdrop size in cells, 0 for none
PARAM_BACKDROP_HEIGHT = const(19)
# Backdrop colors from here to the end, row by row:
# params[PARAM_BACKDROP_COLORS + row * backdrop_width + column]
PARAM_BACKDROP_COLORS = const(20)

```

```
71
72  def new(
73      grid_width,
74      grid_height,
75      grain_count,
76      cell_size,
77      background=0,
78      seed=12345,
79      backdrop_cells=0,
80  ):
81      """Allocate the four state buffers, with room in params for a backdrop of
82      up to `backdrop_cells` cells. Fill in grain colors, maybe backdrop(), then
83      place()."""
84      if not 0 < grain_count < grid_width * grid_height:
85          raise ValueError(
86              "grain_count must be between 1 and grid_width * grid_height - 1"
87          )
88      # Largest first, each at its final size: a 320x240 canvas is 150 KB of
89      # contiguous heap, and growing an array element by element leaves holes.
90      pixels = bytearray(grid_width * cell_size * grid_height * cell_size * 2)
91      grid = bytearray(grid_width * grid_height)
92      grains = array.array("i", [0]) * (5 * grain_count)
93      params = array.array("i", [0]) * (PARAM_BACKDROP_COLORS + backdrop_cells)
94      params[PARAM_GRID_WIDTH] = grid_width
95      params[PARAM_GRID_HEIGHT] = grid_height
96      params[PARAM_GRAIN_COUNT] = grain_count
97      params[PARAM_CELL_SIZE] = cell_size
98      params[PARAM_BACKGROUND] = background
99      params[PARAM_RANDOM_STATE] = seed & 0x3FFFFFFF
100     params[PARAM_JITTER_RANGE] = 1
101     params[PARAM_MAX_SPEED] = 256
102     return grains, grid, pixels, params
103
104
105 def backdrop(grid, params, x, y, width, height, colors, solid):
106     """Show a width x height picture with its top-left cell at (x, y), which
107     may be off the grid. colors and solid go through the picture's cells row
108     by row, giving each cell's color and whether grains are kept out of it.
109     Call before place(), which paints it."""
110     if len(params) < PARAM_BACKDROP_COLORS + width * height:
111         raise ValueError(
112             "params has no room for the backdrop: pass new() backdrop_cells=width*height"
113         )
114     grid_width = params[PARAM_GRID_WIDTH]
115     grid_height = params[PARAM_GRID_HEIGHT]
116     # Clear any earlier backdrop's obstacles.
117     for cell in range(grid_width * grid_height):
118         if grid[cell] == 2:
119             grid[cell] = 0
120     params[PARAM_BACKDROP_X] = x
121     params[PARAM_BACKDROP_Y] = y
122     params[PARAM_BACKDROP_WIDTH] = width
123     params[PARAM_BACKDROP_HEIGHT] = height
124     for row in range(height):
125         for column in range(width):
126             index = row * width + column
```

```
            params[PARAM_BACKDROP_COLORS + index] = colors[index]
            if (
                solid[index]
                and 0 <= x + column < grid_width
                and 0 <= y + row < grid_height
            ):
                grid[(y + row) * grid_width + x + column] = 2
    obstacle_count = sum(1 for occupancy in grid if occupancy == 2)
    if params[PARAM_GRAIN_COUNT] >= grid_width * grid_height - obstacle_count:
        raise ValueError("more grains than free cells")


# The random source is a 30-bit LCG. 30 bits, not 32, so every constant is a small int
# and so the masked result is identical under viper's wrapping arithmetic and CPython's bigints.
#
# Shifting a grain unit position right by 8 (>> 8) divides by 256, giving the
# cell it is in. A cell's index in grid is cell_y * grid_width + cell_x.
#
# Each grain is five ints in grains, starting at grains[offset]:
# offset + 0 and + 1 are its x and y, + 2 and + 3 its velocity, + 4 its color.


@micropython.viper
def place(grains: ptr32, grid: ptr8, pixels: ptr16, params: ptr32):
    """Scatter the grains over distinct free cells at rest and paint the
    canvas: background, backdrop, then grains. Obstacles stay in place."""
    grid_width = params[PARAM_GRID_WIDTH]
    grid_height = params[PARAM_GRID_HEIGHT]
    grain_count = params[PARAM_GRAIN_COUNT]
    cell_size = params[PARAM_CELL_SIZE]
    background = params[PARAM_BACKGROUND]
    random_state = params[PARAM_RANDOM_STATE]
    backdrop_x = params[PARAM_BACKDROP_X]
    backdrop_y = params[PARAM_BACKDROP_Y]
    backdrop_width = params[PARAM_BACKDROP_WIDTH]
    backdrop_height = params[PARAM_BACKDROP_HEIGHT]
    cell_count = grid_width * grid_height
    row_stride = grid_width * cell_size  # pixels in one row of the canvas

    # Take the grains off the grid, leaving obstacles where they are.
    cell = 0
    while cell < cell_count:
        if grid[cell] == 1:
            grid[cell] = 0
        cell += 1
    cell_y = 0
    while cell_y < grid_height:
        cell_x = 0
        while cell_x < grid_width:
            color = background
            backdrop_col = cell_x - backdrop_x
            backdrop_row = cell_y - backdrop_y
            if (
                0 <= backdrop_col < backdrop_width
                and 0 <= backdrop_row < backdrop_height
            ):
```

```
                color = params[
                    PARAM_BACKDROP_COLORS + backdrop_row * backdrop_width + backdrop_col
                ]
            # Fill the cell's square of pixels, one row at a time.
            pixel_index = cell_y * cell_size * row_stride + cell_x * cell_size
            row = 0
            while row < cell_size:
                row_end = pixel_index + cell_size
                while pixel_index < row_end:
                    pixels[pixel_index] = color
                    pixel_index += 1
                pixel_index += row_stride - cell_size  # to the next row's start
                row += 1
            cell_x += 1
        cell_y += 1

    grain = 0
    offset = 0
    while grain < grain_count:
        random_state = (random_state * 1664525 + 1013904223) & 0x3FFFFFFF
        cell = (random_state >> 4) % cell_count
        while grid[cell] != 0:
            random_state = (random_state * 1664525 + 1013904223) & 0x3FFFFFFF
            cell = (random_state >> 4) % cell_count
        grid[cell] = 1
        cell_x = cell % grid_width
        cell_y = cell // grid_width
        # A random spot inside the cell, and not moving.
        grains[offset] = cell_x * 256 + ((random_state >> 12) & 255)
        grains[offset + 1] = cell_y * 256 + ((random_state >> 20) & 255)
        grains[offset + 2] = 0
        grains[offset + 3] = 0
        color = grains[offset + 4]
        pixel_index = cell_y * cell_size * row_stride + cell_x * cell_size
        row = 0
        while row < cell_size:
            row_end = pixel_index + cell_size
            while pixel_index < row_end:
                pixels[pixel_index] = color
                pixel_index += 1
            pixel_index += row_stride - cell_size
            row += 1
        grain += 1
        offset += 5
    params[PARAM_RANDOM_STATE] = random_state


@micropython.viper
def step(grains: ptr32, grid: ptr8, pixels: ptr16, params: ptr32) -> int:
    """Advance one frame. Returns how many grains changed cell; when that is
    nonzero, PARAM_DIRTY_LEFT .. PARAM_DIRTY_BOTTOM bound the cells repainted
    and PARAM_TOP_LEFT_COLOR / PARAM_BOTTOM_RIGHT_COLOR hold the colors of
    that rectangle's corner pixels."""
    grid_width = params[PARAM_GRID_WIDTH]
    grid_height = params[PARAM_GRID_HEIGHT]
    grain_count = params[PARAM_GRAIN_COUNT]
```

```
    cell_size = params[PARAM_CELL_SIZE]
    background = params[PARAM_BACKGROUND]
    random_state = params[PARAM_RANDOM_STATE]
    accel_x = params[PARAM_ACCEL_X]
    accel_y = params[PARAM_ACCEL_Y]
    jitter_range = params[PARAM_JITTER_RANGE]
    max_speed = params[PARAM_MAX_SPEED]
    max_speed_squared = max_speed * max_speed
    backdrop_x = params[PARAM_BACKDROP_X]
    backdrop_y = params[PARAM_BACKDROP_Y]
    backdrop_width = params[PARAM_BACKDROP_WIDTH]
    backdrop_height = params[PARAM_BACKDROP_HEIGHT]
    max_x = grid_width * 256 - 1  # largest position, in grain units
    max_y = grid_height * 256 - 1
    row_stride = grid_width * cell_size  # pixels in one row of the canvas

    # The rectangle of cells repainted this frame, left and top inclusive,
    # right and bottom exclusive.
    dirty_left = grid_width
    dirty_top = grid_height
    dirty_right = 0
    dirty_bottom = 0
    grains_moved = 0
    grain = 0
    offset = 0
    while grain < grain_count:
        # Apply the acceleration vector, plus a little randomness so tall
        # stacks topple, then clip speed to max_speed. The clip is on the 2D
        # vector so diagonal motion isn't faster. The randomness is 15 random
        # bits (0 to 32767) scaled to 0 .. jitter_range - 1.
        random_state = (random_state * 1664525 + 1013904223) & 0x3FFFFFFF
        velocity_x = (
            grains[offset + 2]
            + accel_x
            + ((((random_state >> 15) & 0x7FFF) * jitter_range) >> 15)
        )
        random_state = (random_state * 1664525 + 1013904223) & 0x3FFFFFFF
        velocity_y = (
            grains[offset + 3]
            + accel_y
            + ((((random_state >> 15) & 0x7FFF) * jitter_range) >> 15)
        )
        # Compare squares, so the square root is only needed when too fast.
        speed_squared = velocity_x * velocity_x + velocity_y * velocity_y
        if speed_squared > max_speed_squared:
            # Integer Newton square root: after the first guess every guess
            # is >= floor(sqrt(speed_squared)), and they fall until they
            # reach it. That is >= |velocity_x|, |velocity_y|, so neither
            # component comes out above max_speed.
            speed = (max_speed + speed_squared // max_speed) >> 1
            while True:
                next_guess = (speed + speed_squared // speed) >> 1
                if next_guess >= speed:
                    break
                speed = next_guess
            # Scale both components by max_speed / speed. Negative ones are
```

```
            # made positive first, so they round toward zero too.
            if velocity_x < 0:
                velocity_x = 0 - ((0 - velocity_x) * max_speed // speed)
            else:
                velocity_x = velocity_x * max_speed // speed
            if velocity_y < 0:
                velocity_y = 0 - ((0 - velocity_y) * max_speed // speed)
            else:
                velocity_y = velocity_y * max_speed // speed

        # Collisions are only right for moves of at most one cell per axis,
        # so a faster grain moves in `substeps` equal steps of (step_x,
        # step_y), the first extra_x (extra_y) of them one unit longer so
        # they add up to exactly (velocity_x, velocity_y). Each substep is
        # the original single move. An axis that bounces stops for the rest
        # of the frame, as it would in one move; its extra_x or extra_y
        # becomes -1 to say so.
        substeps = 1
        step_x = velocity_x
        step_y = velocity_y
        extra_x = 0
        extra_y = 0
        if speed_squared > 65536:
            # Maybe faster than one cell (256 units) per frame on some axis.
            # (speed_squared is from before the clamp, so this also catches
            # grains clamped to max_speed.) Speed here means velocity
            # without its sign.
            fastest_speed = velocity_x
            if fastest_speed < 0:
                fastest_speed = 0 - fastest_speed
            speed_y = velocity_y
            if speed_y < 0:
                speed_y = 0 - speed_y
            if speed_y > fastest_speed:
                fastest_speed = speed_y
            if fastest_speed > 256:
                substeps = (fastest_speed + 255) >> 8  # cells to cover, rounded up
                step_x = velocity_x // substeps
                extra_x = velocity_x - step_x * substeps
                step_y = velocity_y // substeps
                extra_y = velocity_y - step_y * substeps

        # Move the grain, treating all the others as stationary. A bounce
        # is `velocity = (0 - velocity) >> 1`, reversing at half speed.
        start_x = grains[offset]
        start_y = grains[offset + 1]
        start_cell = (start_y >> 8) * grid_width + (start_x >> 8)
        grain_x = start_x
        grain_y = start_y
        current_cell = start_cell
        substep = 0
        while substep < substeps:
            next_x = grain_x + step_x
            if substep < extra_x:
                next_x += 1
            next_y = grain_y + step_y
```

```
            if substep < extra_y:
                next_y += 1
            # Bounce off the edges of the grid.
            if next_x > max_x:
                next_x = max_x
                velocity_x = (0 - velocity_x) >> 1
                step_x = 0
                extra_x = -1
            elif next_x < 0:
                next_x = 0
                velocity_x = (0 - velocity_x) >> 1
                step_x = 0
                extra_x = -1
            if next_y > max_y:
                next_y = max_y
                velocity_y = (0 - velocity_y) >> 1
                step_y = 0
                extra_y = -1
            elif next_y < 0:
                next_y = 0
                velocity_y = (0 - velocity_y) >> 1
                step_y = 0
                extra_y = -1

            # Bounce off whatever is in the cell it would move into.
            next_cell = (next_y >> 8) * grid_width + (next_x >> 8)
            if next_cell != current_cell and grid[next_cell] != 0:
                cell_distance = next_cell - current_cell
                if cell_distance < 0:
                    cell_distance = 0 - cell_distance
                if cell_distance == 1:  # blocked left or right: cancel x
                    next_x = grain_x
                    velocity_x = (0 - velocity_x) >> 1
                    step_x = 0
                    extra_x = -1
                    next_cell = current_cell
                elif cell_distance == grid_width:  # blocked above or below: cancel y
                    next_y = grain_y
                    velocity_y = (0 - velocity_y) >> 1
                    step_y = 0
                    extra_y = -1
                    next_cell = current_cell
                else:
                    # Diagonal. Try sliding along one axis alone, faster
                    # first; either one alone is sure to change cell.
                    current_speed_x = velocity_x
                    if current_speed_x < 0:
                        current_speed_x = 0 - current_speed_x
                    current_speed_y = velocity_y
                    if current_speed_y < 0:
                        current_speed_y = 0 - current_speed_y
                    if current_speed_x >= current_speed_y:
                        next_cell = (grain_y >> 8) * grid_width + (next_x >> 8)
                        if grid[next_cell] == 0:  # x alone is free
                            next_y = grain_y
                            velocity_y = (0 - velocity_y) >> 1
```

```
                    step_y = 0
                    extra_y = -1
                else:
                    next_cell = (next_y >> 8) * grid_width + (grain_x >> 8)
                    if grid[next_cell] == 0:  # y alone is free
                        next_x = grain_x
                        velocity_x = (0 - velocity_x) >> 1
                        step_x = 0
                        extra_x = -1
                    else:  # neither: stay put, bounce on both axes
                        next_x = grain_x
                        next_y = grain_y
                        velocity_x = (0 - velocity_x) >> 1
                        velocity_y = (0 - velocity_y) >> 1
                        step_x = 0
                        extra_x = -1
                        step_y = 0
                        extra_y = -1
                        next_cell = current_cell
            else:
                next_cell = (next_y >> 8) * grid_width + (grain_x >> 8)
                if grid[next_cell] == 0:  # y alone is free
                    # The original bounces vy here, against its own
                    # comment; x is the axis being cancelled.
                    next_x = grain_x
                    velocity_x = (0 - velocity_x) >> 1
                    step_x = 0
                    extra_x = -1
                else:
                    next_cell = (grain_y >> 8) * grid_width + (next_x >> 8)
                    if grid[next_cell] == 0:  # x alone is free
                        next_y = grain_y
                        velocity_y = (0 - velocity_y) >> 1
                        step_y = 0
                        extra_y = -1
                    else:  # neither: stay put, bounce on both axes
                        next_x = grain_x
                        next_y = grain_y
                        velocity_x = (0 - velocity_x) >> 1
                        velocity_y = (0 - velocity_y) >> 1
                        step_x = 0
                        extra_x = -1
                        step_y = 0
                        extra_y = -1
                        next_cell = current_cell
        grain_x = next_x
        grain_y = next_y
        current_cell = next_cell
        if step_x == 0 and extra_x <= 0 and step_y == 0 and extra_y <= 0:
            break  # stopped on both axes, so the other substeps would do nothing
        substep += 1
    # Rebound no faster than the original's top speed of one cell per
    # frame allows. v /= -2 from three cells per frame would bounce
    # grains nine times as high, and the sand would take longer to settle.
    if extra_x < 0:  # bounced in x this frame
        if velocity_x > 128:
```

```
            velocity_x = 128
        elif velocity_x < -128:
            velocity_x = -128
    if extra_y < 0:  # bounced in y this frame
        if velocity_y > 128:
            velocity_y = 128
        elif velocity_y < -128:
            velocity_y = -128
    grains[offset] = grain_x
    grains[offset + 1] = grain_y
    grains[offset + 2] = velocity_x
    grains[offset + 3] = velocity_y

    # The grain's own start cell is still marked in grid while it moves,
    # which is harmless: each axis only moves one way, so no substep can
    # return to it.
    if current_cell != start_cell:
        grid[start_cell] = 0
        grid[current_cell] = 1
        grains_moved += 1
        # Erase the old cell, to the backdrop if it's on it...
        cell_x = start_x >> 8
        cell_y = start_y >> 8
        if cell_x < dirty_left:
            dirty_left = cell_x
        if cell_x >= dirty_right:
            dirty_right = cell_x + 1
        if cell_y < dirty_top:
            dirty_top = cell_y
        if cell_y >= dirty_bottom:
            dirty_bottom = cell_y + 1
        color = background
        backdrop_col = cell_x - backdrop_x
        backdrop_row = cell_y - backdrop_y
        if (
            0 <= backdrop_col < backdrop_width
            and 0 <= backdrop_row < backdrop_height
        ):
            color = params[
                PARAM_BACKDROP_COLORS + backdrop_row * backdrop_width + backdrop_col
            ]
        pixel_index = cell_y * cell_size * row_stride + cell_x * cell_size
        row = 0
        while row < cell_size:
            row_end = pixel_index + cell_size
            while pixel_index < row_end:
                pixels[pixel_index] = color
                pixel_index += 1
            pixel_index += row_stride - cell_size
            row += 1
        # ...and draw the new one.
        cell_x = grain_x >> 8
        cell_y = grain_y >> 8
        if cell_x < dirty_left:
            dirty_left = cell_x
        if cell_x >= dirty_right:
```

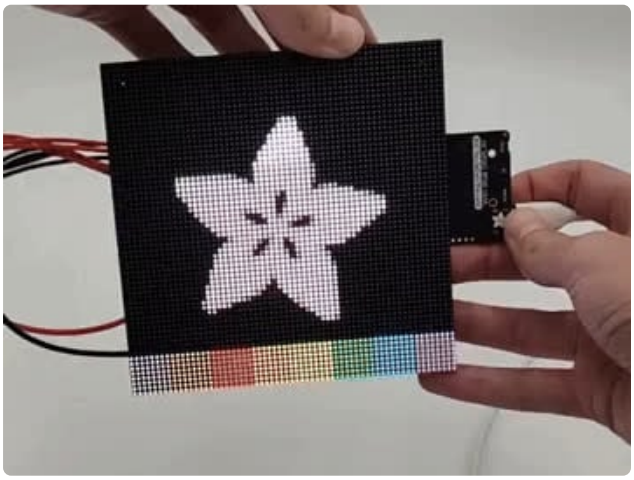
```
                dirty_right = cell_x + 1
            if cell_y < dirty_top:
                dirty_top = cell_y
            if cell_y >= dirty_bottom:
                dirty_bottom = cell_y + 1
            color = grains[offset + 4]
            pixel_index = cell_y * cell_size * row_stride + cell_x * cell_size
            row = 0
            while row < cell_size:
                row_end = pixel_index + cell_size
                while pixel_index < row_end:
                    pixels[pixel_index] = color
                    pixel_index += 1
                pixel_index += row_stride - cell_size
                row += 1
        grain += 1
        offset += 5

    params[PARAM_RANDOM_STATE] = random_state
    params[PARAM_DIRTY_LEFT] = dirty_left
    params[PARAM_DIRTY_TOP] = dirty_top
    params[PARAM_DIRTY_RIGHT] = dirty_right
    params[PARAM_DIRTY_BOTTOM] = dirty_bottom
    if grains_moved != 0:
        params[PARAM_TOP_LEFT_COLOR] = pixels[
            dirty_top * cell_size * row_stride + dirty_left * cell_size
        ]
        params[PARAM_BOTTOM_RIGHT_COLOR] = pixels[
            (dirty_bottom * cell_size - 1) * row_stride + dirty_right * cell_size - 1
        ]
    return grains_moved
```

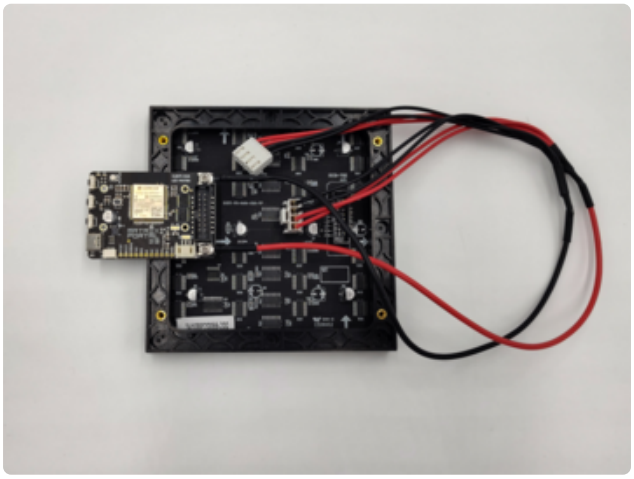
https://github.com/adafruit/Adafruit_Learning_System_Guides/blob/main/CircuitPython_Turbo/sand_simulation/metro_rp2350/sand_viper.py

Here is the command used to compile the module for the RP2350.

```
mpy-cross -march=armv7emsp sand_viper.py
```

Matrix Portal S3





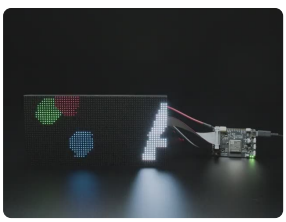
This section of the guide covers a version of the Pixel Dust demo that runs on the Matrix Portal S3. It's made for a single 64x64 LED matrix. The Matrix Portal has an LIS3DH accelerometer on-board that is used to determine the direction of gravity within the simulation. The only assembly required is hooking up the LED panel to the Matrix Portal with the HUB75 connector and power wires.

This version uses a smaller overall display size and fewer sand particles than the Metro version. I used the 2mm pitch matrix panels, but other sizes would work fine as well.

Text emphasized with a yellow exclamation:

This code is set to work on a 64x64 pixel LED matrix only. Do not connect other sizes of displays like 32x64, etc. as it likely will not work. Different density 64x64 displays, with different pitches, in millimeters, should be fine.

Parts



Adafruit Matrix Portal S3 CircuitPython Powered Internet Display

Folks love our wide selection of RGB matrices and accessories for making custom colorful LED displays... and our RGB Matrix...

https://www.adafruit.com/product/5778

64x64 RGB LED Matrix Panel - 2mm Pitch

Wintertime can be rough in the city. The sky is gray. The weather is unpredictable. So slough off those seasonal blues with some Times Square razzle dazzle from this...

https://www.adafruit.com/product/5362

Install CircuitPython

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** drive to iterate.

Set up CircuitPython Quick Start!

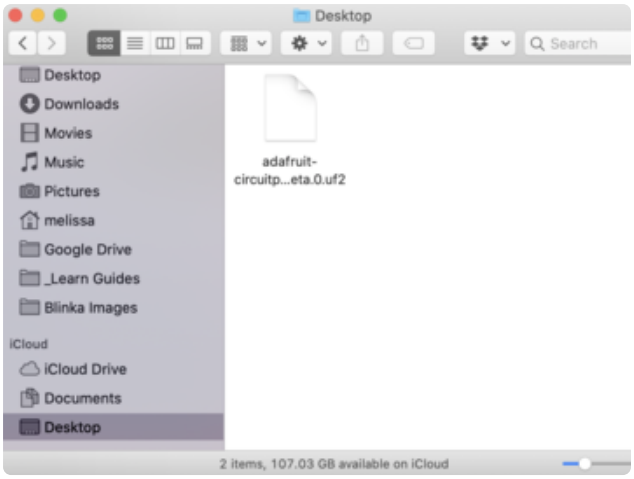
Follow this quick step-by-step for super-fast Python power :)

Text emphasized with a blue exclamation:

This project requires CircuitPython version **10.4.0-alpha.2** or newer, because that is when Viper support was added for the Espressif port. As of 9/29/26 version **11.0.0-alpha.1** is the latest development release. Scroll down to the latest development release on the downloads page.

Further Information

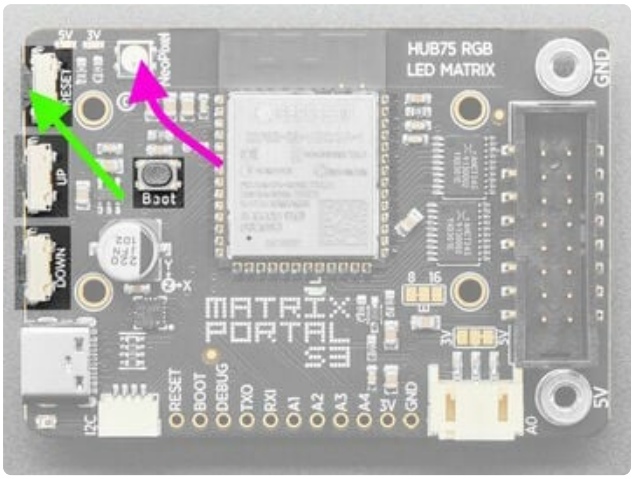
For more detailed info on installing CircuitPython, check out Installing CircuitPython ().



Click the link above and download the latest UF2 file.

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

Plug your MatrixPortal S3 into your computer using a known-good USB cable.



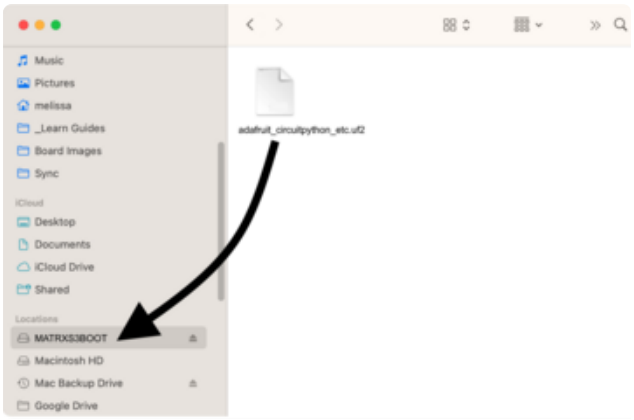
A lot of people end up using charge-only USB cables and it is very frustrating! So make sure you have a USB cable you know is good for data sync.

Click the **Reset** button (indicated by the green arrow) on your board. When you see the NeoPixel RGB LED (indicated by the magenta arrow) turn purple, press it again. At that point, the NeoPixel should turn green. If it turns red, check the USB cable, try another USB port, etc.

If double-clicking doesn't work the first time, try again. Sometimes it can take a few tries to get the rhythm right!

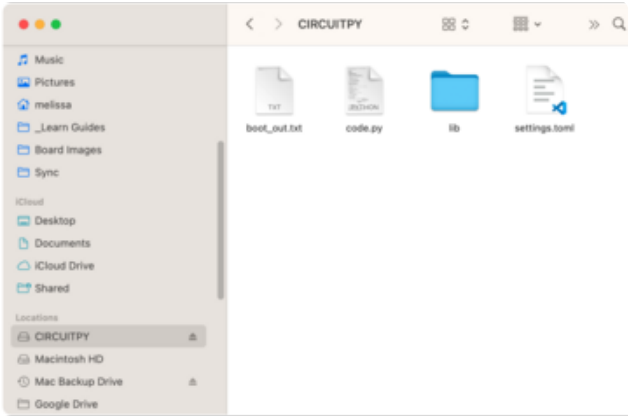
Text emphasized with a blue exclamation:

Early shipments of the MatrixPortal S3 do not have the UF2 bootloader installed. Double-clicking will not produce a BOOT drive. Follow these instructions to install the UF2 bootloader: https://learn.adafruit.com/adafruit-matrixportal-s3/factory-reset#factory-reset-and-bootloader-repair-3107941



You will see a new disk drive appear called **MATRXS3BOOT**.

Drag the **adafruit_circuitpython_etc.uf2** file over to **MATRXS3BOOT.**



The LED will flash. Then, the **MATRXS3BOOT** drive will disappear and a new disk drive called **CIRCUITPY** will appear.

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

Code

Getting the Program's Files

Viper and native compiled .mpy files are on the cutting edge of support in CircuitPython. The Adafruit Learning System project bundler has not yet been adapted to work with them. Use the green button below to download the project files instead of the usual "Download Project Bundle" button embedded with the code contents.

Connect your board to your computer via a known good data+power USB cable. The board should show up in your File Explorer/Finder (depending on your operating system) as a flash drive named **CIRCUITPY**.

Extract the contents of the zip file, copy the **lib** directory files to **CIRCUITPY/lib**. Copy the **code.py** and **logo.py** files to your **CIRCUITPY** drive. The program should self start.

After loading the project on your device the **CIRCUITPY** drive should contain these files. The **src/** folder is optional. When the code runs it will use the **.mpy** version in the **lib/** directory. Keeping a copy of the plain source code on the device is nice if you've got the flash storage space to spare, it makes it easy if you ever want to modify the code and re-compile it.

code.py

This is the main code entry point. It initializes the matrix and accelerometer. It runs the main loop, deferring to **sand.mpy** to carry out the heavy math for each step of the simulation. This version uses `FramebufferDisplay` to drive the LED matrix panel.

```
# SPDX-FileCopyrightText: 2026 Mikey Sklar for Adafruit Industries
# SPDX-FileCopyrightText: 2020 Phil Burgess for Adafruit Industries
# SPDX-License-Identifier: MIT
#
# "Pixel dust" on a 64x64 HUB75 panel, Matrix Portal S3 edition. Tilt the
# board and the sand runs with gravity, read from the onboard LIS3DH, around
# the Adafruit logo as in PixelDust's demo3-logo. The
# physics lives in src/sand.py so turbo can compile it; this file only wires
# up the display, the accelerometer and the frame loop. Prints frames/second
# and the time spent in the kernel every 100 frames.
import time
from array import array

import adafruit_lis3dh
import board
import displayio
import framebufferio
import rgbmatrix

import logo
import sand
```

```
22
23  displayio.release_displays()
24
25  # 64x64 panel, 1/32 scan, five address lines (MTX_ADDRA..E).
26  WIDTH, HEIGHT = 64, 64
27  MAX_FPS = 1000  # the C demo uses 45; 1000 is effectively unthrottled
28  ELASTICITY = 128  # bounce keeps this /256 of its speed
29  # Grains start as N_COLORS blocks of BOX_WIDTH x BOX_HEIGHT along the bottom.
30  N_COLORS = 8
31  BOX_WIDTH = WIDTH // N_COLORS
32  BOX_HEIGHT = 8
33  N_GRAINS = N_COLORS * BOX_WIDTH * BOX_HEIGHT
34  OBSTACLE = N_COLORS + 1  # cell value for the logo in the collision map
35  BG = (0, 0, 0)  # background behind the logo (demo3-logo.cpp: 0, 20, 80)
36  # Which accelerometer axis feeds which simulation axis, and its sign. The
37  # panel's orientation relative to the LIS3DH is not something the code can
38  # know; flip these if the sand runs the wrong way.
39  AX, AY, AZ = (0, 1), (1, 1), (2, 1)
40
41  matrix = rgbmatrix.RGBMatrix(
42      width=WIDTH,
43      height=HEIGHT,
44      bit_depth=4,
45      # This panel has red and blue swapped, so the B pins drive red. For a
46      # panel wired the usual way, list them R, G, B.
47      rgb_pins=[
48          board.MTX_B1,
49          board.MTX_G1,
50          board.MTX_R1,
51          board.MTX_B2,
52          board.MTX_G2,
53          board.MTX_R2,
54      ],
55      addr_pins=[
56          board.MTX_ADDRA,
57          board.MTX_ADDRB,
58          board.MTX_ADDRC,
59          board.MTX_ADDRD,
60          board.MTX_ADDRE,
61      ],
62      clock_pin=board.MTX_CLK,
63      latch_pin=board.MTX_LAT,
64      output_enable_pin=board.MTX_OE,
65  )
66  display = framebufferio.FramebufferDisplay(matrix, auto_refresh=False)
67
68  # Bottom layer: the white logo alpha-blended onto the background, once. The
69  # palette index is the logo's gray level, so the blend lives in the palette.
70  LOGO_X = (WIDTH - logo.WIDTH) // 2
71  LOGO_Y = (HEIGHT - logo.HEIGHT) // 2
72  backdrop = displayio.Bitmap(WIDTH, HEIGHT, 256)
73  backdrop_palette = displayio.Palette(256)
74  for g in range(256):
75      a1 = g + 1
76      a2 = 257 - a1
77      backdrop_palette[g] = (
```

```
        ((255 * a1 + BG[0] * a2) >> 8) << 16
        | ((255 * a1 + BG[1] * a2) >> 8) << 8
        | ((255 * a1 + BG[2] * a2) >> 8)
    )
for y in range(logo.HEIGHT):
    for x in range(logo.WIDTH):
        backdrop[LOGO_X + x, LOGO_Y + y] = logo.GRAY[y * logo.WIDTH + x]

# Top layer: the sand. One byte per cell so the Bitmap's buffer maps
# row-major with no padding (width is a multiple of 4). It doubles as the
# kernel's collision map; any nonzero cell is solid, so the logo's mask goes
# in as OBSTACLE cells, drawn transparent so the backdrop shows through.
bitmap = displayio.Bitmap(WIDTH, HEIGHT, 256)
palette = displayio.Palette(OBSTACLE + 1)
palette[0] = 0x000000
palette.make_transparent(0)
palette[OBSTACLE] = 0x000000
palette.make_transparent(OBSTACLE)
for y in range(logo.HEIGHT):
    for x in range(logo.WIDTH):
        if logo.MASK[y * 5 + (x >> 3)] & (0x80 >> (x & 7)):
            bitmap[LOGO_X + x, LOGO_Y + y] = OBSTACLE
for i, rgb in enumerate(
    (
        (64, 64, 64),  # dark gray
        (120, 79, 23),  # brown
        (228, 3, 3),  # red
        (255, 140, 0),  # orange
        (255, 237, 0),  # yellow
        (0, 128, 38),  # green
        (0, 77, 255),  # blue
        (117, 7, 135),  # purple
    )
):
    palette[i + 1] = (rgb[0] << 16) | (rgb[1] << 8) | rgb[2]
group = displayio.Group()
group.append(displayio.TileGrid(backdrop, pixel_shader=backdrop_palette))
group.append(displayio.TileGrid(bitmap, pixel_shader=palette))
display.root_group = group

accel = adafruit_lis3dh.LIS3DH_I2C(board.I2C(), address=0x19)
accel.range = adafruit_lis3dh.RANGE_4_G

grains = array("i", [0] * (5 * N_GRAINS))
rng = array("I", [time.monotonic_ns() & 0x7FFFFFFF])
n = 0
for c in range(N_COLORS):
    for y in range(HEIGHT - BOX_HEIGHT, HEIGHT):
        for x in range(c * BOX_WIDTH, (c + 1) * BOX_WIDTH):
            sand.place(grains, bitmap, n, x, y, WIDTH, c + 1)
            n += 1
bitmap.dirty()
display.refresh()

print(N_GRAINS, "grains")

```

```
FRAME_NS = 1_000_000_000 // MAX_FPS
next_frame = time.monotonic_ns()
frames = 0
kernel_ns = 0
t_report = next_frame

while True:
    # Throttle to MAX_FPS. The kernel's time varies with how much is
    # moving, so without this gravity would look stronger in a settled pile.
    now = time.monotonic_ns()
    if now < next_frame:
        continue
    next_frame = now + FRAME_NS

    a = accel.acceleration  # m/s^2
    ax = int(a[AX[0]] * 1000) * AX[1]
    ay = int(a[AY[0]] * 1000) * AY[1]
    az = int(a[AZ[0]] * 1000) * AZ[1]

    t0 = time.monotonic_ns()
    sand.iterate(grains, bitmap, N_GRAINS, WIDTH, HEIGHT, ax, ay, az, ELASTICITY, rng)
    kernel_ns += time.monotonic_ns() - t0
    bitmap.dirty()  # the kernel wrote through the buffer, tell displayio
    display.refresh()

    frames += 1
    if frames == 100:
        elapsed = now - t_report
        print(
            "%.1f fps, kernel %.2f ms/frame, accel (%d, %d, %d)"
            % (frames * 1e9 / elapsed, kernel_ns / frames / 1e6, ax, ay, az)
        )
        frames = 0
        kernel_ns = 0
        t_report = now
```

https://github.com/adafruit/Adafruit_Learning_System_Guides/blob/main/CircuitPython_Turbo/sand_simulation/matrix_portal_s3/code.py

sand.py

This is the code for the viper module that handles the physics and collision math for the simulation. Viper functions are written as Python, but have an extra set of rules that must be followed.

Local variables and arguments must be `int`, `uint` (32-bit only), `bool`, `ptr`, `ptr8`, `ptr16`, `ptr32`, or `object`. They cannot use `floats` like normal Python numbers do. Parameters get passed to the Viper function as a list of values, and are accessed via their index within that list. There is no overflow protection. Integers will silently wrap, out-of-range indexes can corrupt memory or crash instead of raising a clean `IndexError`. Variables cannot change types. Mixing of typed values with `object` values will fall back to slower runtime calls.

This code gets compiled by `mpy-cross` for the specific architecture of the chip that it will run on. The **sand_viper.mpy** file included in the bundle is compiled for the ESP32-S3 architecture `xtensawin`.

```
# SPDX-FileCopyrightText: 2026 Mikey Sklar for Adafruit Industries
# SPDX-FileCopyrightText: 2020 Phil Burgess for Adafruit Industries
#
# SPDX-License-Identifier: MIT
# "Pixel dust" sand physics from Adafruit_PixelDust (Phil Burgess, BSD),
# ported to a turbo viper kernel. Integer only: positions are 8.8 fixed
# point (pixel * 256), velocities are in the same units per frame, terminal
# velocity is 256 (one pixel) so grains never tunnel through each other.
#
# `g` is an array('i') of 5 ints per grain: x, y, vx, vy, colour. `fb` is any
# writable byte buffer of height rows, width bytes apart, one byte per cell:
# 0 free, else the colour of the grain sitting there. A displayio.Bitmap with
# value_count=256 and a width that is a multiple of 4 works directly and is
# both the collision map and the frame. `rng` is a one-element array('I')
# holding the random state, advanced in place.
#
# Where the C truncates toward zero on a division this does too, so a grain
# bouncing left behaves like one bouncing right.
import micropython

# pylint: disable=too-many-locals, undefined-variable, too-many-nested-blocks, too-many-branches, too-many-statements
# pylint: disable=import-outside-toplevel


@micropython.viper
def iterate(
    g: ptr32,
    fb: ptr8,
    n: int,
    width: int,
    height: int,
    ax: int,
    ay: int,
    az: int,
    elasticity: int,
    rng: ptr32,
) -> int:
    xmax = width * 256 - 1
    ymax = height * 256 - 1

    # Scale raw accelerometer input (milli-m/s^2) down to grain units. The C
    # has a `scale` parameter that the demo leaves at 1, so it is folded in.
    if ax >= 0:
        ax = ax // 256
    else:
        ax = -((-ax) // 256)
    if ay >= 0:
        ay = ay // 256
    else:
        ay = -((-ay) // 256)
```

```
    if az < 0:
        az = -az
    az = az // 2048
    # A bit of random jitter per grain so tall stacks topple instead of
    # sliding as a block. Stronger the further from level the panel is.
    if az >= 4:
        az = 1
    else:
        az = 5 - az
    ax -= az
    ay -= az
    az2 = az * 2 + 1

    r = rng[0]
    moved = 0
    for i in range(n):
        b = i * 5
        x = g[b]
        y = g[b + 1]
        vx = g[b + 2]
        vy = g[b + 3]

        # Apply the 2D acceleration vector plus jitter.
        r = (r * 214013 + 2531011) & 0x3FFFFFFF
        vx += ax + ((r >> 16) % az2)
        r = (r * 214013 + 2531011) & 0x3FFFFFFF
        vy += ay + ((r >> 16) % az2)
        # Clip velocity as a 2D vector at 256 so diagonal motion is no
        # faster than axial. Integer Newton sqrt stands in for sqrtf().
        v2 = vx * vx + vy * vy
        if v2 > 65536:
            v = 512
            for _ in range(8):
                v = (v + v2 // v) >> 1
            t = vx * 256
            if t >= 0:
                vx = t // v
            else:
                vx = -((-t) // v)
            t = vy * 256
            if t >= 0:
                vy = t // v
            else:
                vy = -((-t) // v)

        # Tentative new position, bouncing off the walls.
        nx = x + vx
        ny = y + vy
        if nx < 0:
            nx = 0
            if vx >= 0:
                vx = -((vx * elasticity) // 256)
            else:
                vx = ((-vx) * elasticity) // 256
        elif nx > xmax:
            nx = xmax
```

```
            if vx >= 0:
                vx = -((vx * elasticity) // 256)
            else:
                vx = ((-vx) * elasticity) // 256
        if ny < 0:
            ny = 0
            if vy >= 0:
                vy = -((vy * elasticity) // 256)
            else:
                vy = ((-vy) * elasticity) // 256
        elif ny > ymax:
            ny = ymax
            if vy >= 0:
                vy = -((vy * elasticity) // 256)
            else:
                vy = ((-vy) * elasticity) // 256

        # Cell indices before and after. Comparing indices rather than x and
        # y separately makes the blocked-direction test cheap.
        ox = x >> 8
        oy = y >> 8
        cx = nx >> 8
        cy = ny >> 8
        oldidx = oy * width + ox
        newidx = cy * width + cx

        if oldidx != newidx and fb[newidx] != 0:
            delta = newidx - oldidx
            if delta < 0:
                delta = -delta
            if delta == 1:
                # Blocked left or right: cancel X, bounce X, Y is fine.
                nx = x
                if vx >= 0:
                    vx = -((vx * elasticity) // 256)
                else:
                    vx = ((-vx) * elasticity) // 256
            elif delta == width:
                # Blocked up or down: cancel Y, bounce Y, X is fine.
                ny = y
                if vy >= 0:
                    vy = -((vy * elasticity) // 256)
                else:
                    vy = ((-vy) * elasticity) // 256
            else:
                # Diagonal: try skidding along the faster axis first, then
                # the other, else stop dead.
                avx = vx
                if avx < 0:
                    avx = -avx
                avy = vy
                if avy < 0:
                    avy = -avy
                if avx >= avy:
                    if fb[oy * width + cx] == 0:
                        ny = y
```

```
                    if vy >= 0:
                        vy = -((vy * elasticity) // 256)
                    else:
                        vy = ((-vy) * elasticity) // 256
                elif fb[cy * width + ox] == 0:
                    nx = x
                    if vx >= 0:
                        vx = -((vx * elasticity) // 256)
                    else:
                        vx = ((-vx) * elasticity) // 256
                else:
                    nx = x
                    ny = y
                    if vx >= 0:
                        vx = -((vx * elasticity) // 256)
                    else:
                        vx = ((-vx) * elasticity) // 256
                    if vy >= 0:
                        vy = -((vy * elasticity) // 256)
                    else:
                        vy = ((-vy) * elasticity) // 256
            else:
                if fb[cy * width + ox] == 0:
                    nx = x
                    if vx >= 0:
                        vx = -((vx * elasticity) // 256)
                    else:
                        vx = ((-vx) * elasticity) // 256
                elif fb[oy * width + cx] == 0:
                    ny = y
                    if vy >= 0:
                        vy = -((vy * elasticity) // 256)
                    else:
                        vy = ((-vy) * elasticity) // 256
                else:
                    nx = x
                    ny = y
                    if vx >= 0:
                        vx = -((vx * elasticity) // 256)
                    else:
                        vx = ((-vx) * elasticity) // 256
                    if vy >= 0:
                        vy = -((vy * elasticity) // 256)
                    else:
                        vy = ((-vy) * elasticity) // 256
            newidx = (ny >> 8) * width + (nx >> 8)

        c = g[b + 4]
        fb[oldidx] = 0
        fb[newidx] = c
        if newidx != oldidx:
            moved += 1
        g[b] = nx
        g[b + 1] = ny
        g[b + 2] = vx
        g[b + 3] = vy
```

```
    rng[0] = r
    return moved


def place(g, fb, i, x, y, width, colour):
    """Put grain i at pixel (x, y) with the given colour index (1..255).
    Returns False, leaving everything unchanged, if the cell is taken."""
    idx = y * width + x
    if fb[idx]:
        return False
    fb[idx] = colour
    b = i * 5
    g[b] = x * 256
    g[b + 1] = y * 256
    g[b + 2] = 0
    g[b + 3] = 0
    g[b + 4] = colour
    return True


def _turbo_bench():
    # 64x64 world, 512 grains in the demo's eight 8x8 colour blocks along the
    # bottom, 60 frames of gravity that swings between the four edges.
    # Returns a hash of every grain field; every tier must reproduce it.
    from array import array

    W, H, N = 64, 64, 512
    fb = bytearray(W * H)
    g = array("i", [0] * (5 * N))
    rng = array("I", [12345])
    i = 0
    for c in range(8):
        for y in range(H - 8, H):
            for x in range(c * 8, c * 8 + 8):
                place(g, fb, i, x, y, W, c + 1)
                i += 1
    dirs = ((0, 9800, 300), (9800, 0, 300), (0, -9800, 300), (-9800, 0, 300))
    for k in range(60):
        ax, ay, az = dirs[(k // 15) % 4]
        iterate(g, fb, N, W, H, ax, ay, az, 128, rng)
    h = 0
    for v in g:
        h = (h * 31 + v) & 0x3FFFFFFF
    return h
```

https://github.com/adafruit/Adafruit_Learning_System_Guides/blob/main/CircuitPython_Turbo/sand_simulation/matrix_portal_s3/sand.py

Here is the command used to compile the sand module for the Matrix Portal S3.

```
mpy-cross -march=xtensawin sand.py
```