# Computer Graphics Turbo Demos

Created by John Park

https://learn.adafruit.com/computer-graphics-turbo-demos

Last updated on 2026-10-30 12:29:54 AM UTC

# Table of Contents


Overview 3

- Parts

Install CircuitPython 7

- CircuitPython Quickstart
- Safe Mode
- Flash Resetting UF2

Code 11

- Installation
- Kiosk Mode
- How It Works
- Voxel Space Demo Code
-
- Cornell Box Demo Code
-
- Metaballs Demo Code
- 3D Models Demo Code
- Menu Code

Setup and Run Demos 20

- Menu Select
- VoxelSpace
- Cornell Box Path Tracer
- Metaballs
- 3D Models

# Overview

It's time to run some classic computer graphics demos in CircuitPython on the Fruit Jam. You'll feel like you're right back at SIGGRAPH in the golden olden days of CGI!

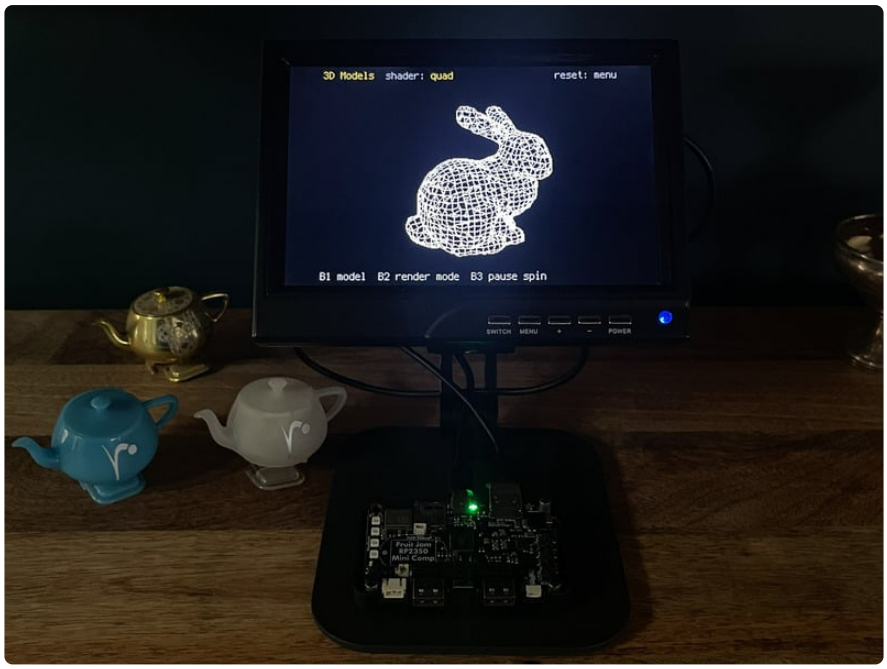
By using CircuitPython Turbo () you can pre-compile some intensely math-y bits of code for maximum speed.

**VoxelSpace**: a Novalogic Comanche-style voxel terrain fly-over on height-map generated terrain.

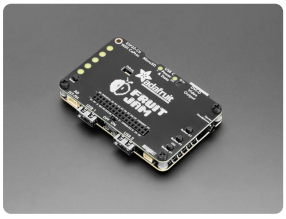
**Cornell box**: a progressive path tracer, 240x240, one sample per pixel per pass, the render refines on screen from noise to a clean image in a minute or two.

**Metaballs**: six 2D metaballs gliding on Lissajous paths, shaded as glossy 3D blobs that merge, over a glowing contoured field.

**3D models**: the Utah teapot and the Stanford bunny, spinning at 30 fps. Quad patch, triangle mesh, Lambert, Phong, and Blinn shading, plus a bonus UV mapped texture demo.




## Parts



Featured

### Adafruit Fruit Jam - Mini RP2350 Computer

We were catching up on a recent hackaday hackchat with eben upton and learned some fun facts:...

https://www.adafruit.com/product/6200

Featured

### HDMI Cable - 1 meter

Connect two HDMI devices together with this basic HDMI cable. It has nice molded grips for easy installation, and is 1 meter long (about 3 feet). This is a HDMI 1.3...

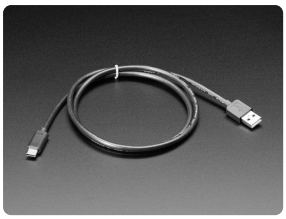
https://www.adafruit.com/product/608

Featured

### 7" Display 1280x800 (720p) IPS + Speakers - HDMI/VGA/NTSC/PAL

Yes, this is an adorable small HDMI television with incredibly high resolution and built in 3W stereo speakers! We tried to get the smallest possible HDMI/VGA display with...

https://www.adafruit.com/product/1667

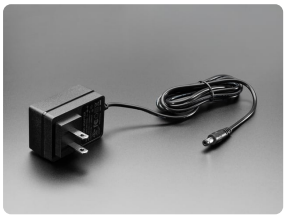


Featured

## USB Type A to Type C Cable - approx 1 meter / 3 ft long

As technology changes and adapts, so does Adafruit. This  USB Type A to Type C cable will help you with the transition to USB C, even if you're still...

https://www.adafruit.com/product/4474



Featured

## 9 VDC 1000mA regulated switching power adapter - UL listed

This is a really nice power supply. It's a switching DC supply so it's small and light and efficient. It is thin so it fits in power strips without blocking other outlets. The...

https://www.adafruit.com/product/63

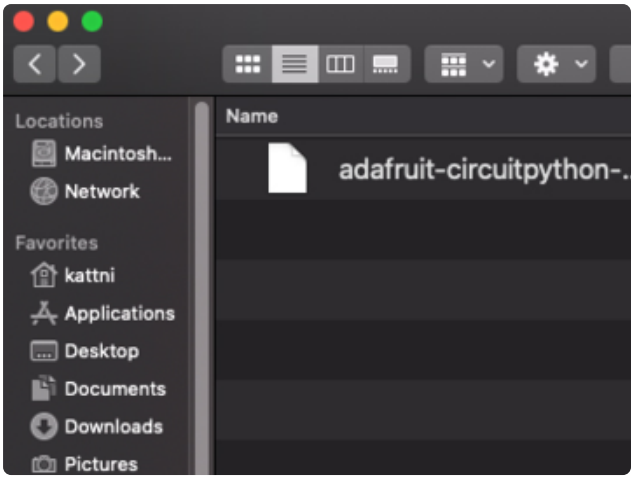
# Install CircuitPython

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

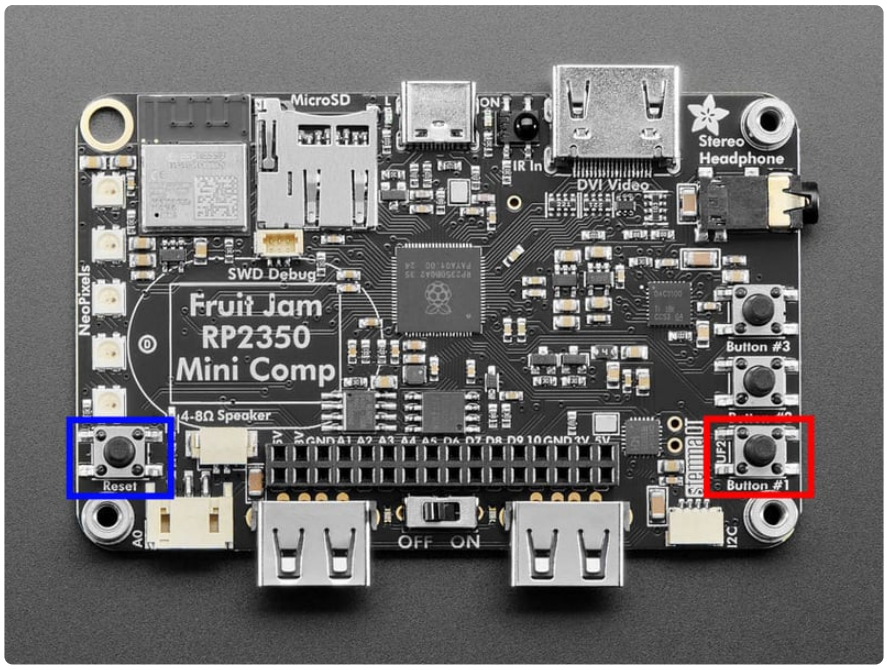
Follow this step-by-step to quickly get CircuitPython running on your board.

| | | |
|---|---|---|
| ! | Text emphasized with a yellow exclamation: | Please use the latest release of 10.x or higher for the Fruit Jam. Also use the latest libraries for the best functionality. |




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

Save it wherever is convenient for you.

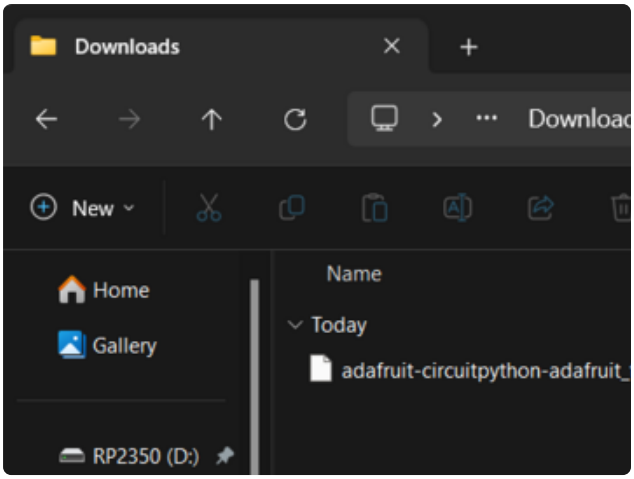



To enter the bootloader, hold down the **BOOT/BOOTSEL button** (highlighted in red above), and while continuing to hold it (don't let go!), press and release the **reset button** (highlighted in red or blue above). **Continue to hold the BOOT/BOOTSEL button until the RP2350 drive appears!**

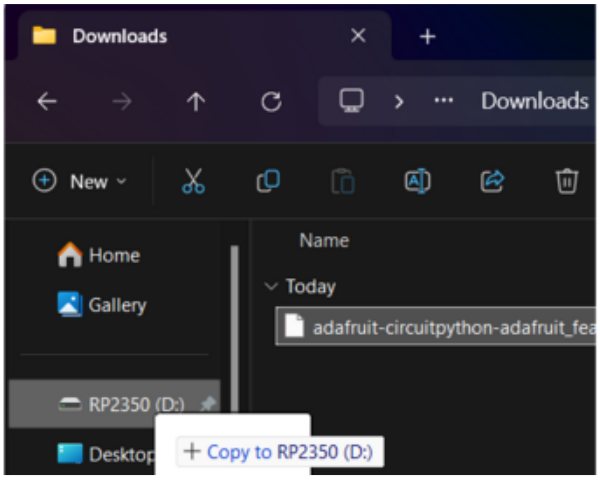
If the drive does not appear, release all the buttons, and then repeat the process above.

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.

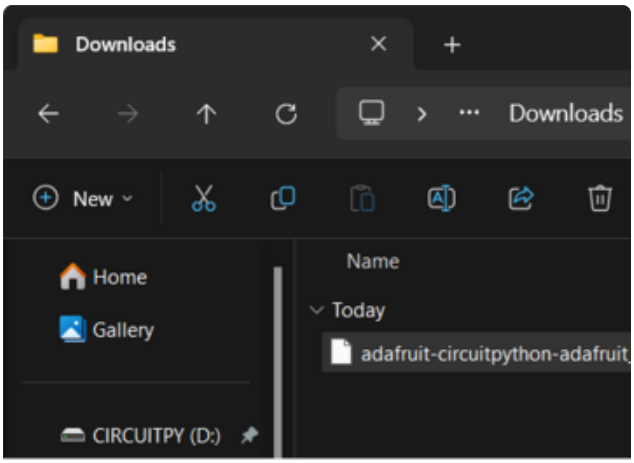
A lot of people end up using charge-only USB cables and it is very frustrating! **Make sure you have a USB cable you know is good for data sync.**




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

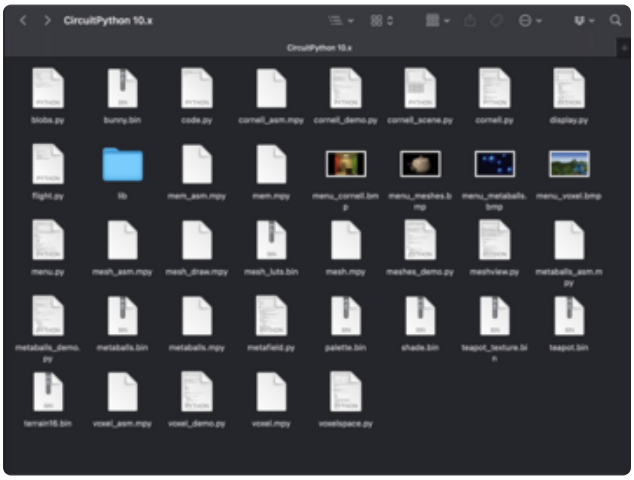
If your board ever gets into a really weird state and CIRCUITPY doesn't show up as a disk drive after installing CircuitPython, try loading this 'nuke' UF2 to RP2350. which

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.

---

# Code

The source code for Computer Graphics Turbo Demos lives here ().



## Installation

The project uses a specific set of CircuitPython libraries, compiled .mpy files, the main **code.py** file, and a number of supporting .py and asset files.

We can do this in one go. In the example code block below, click the **Download Project Bundle** button to download the zip file. Extract the contents of the zip file, open the **CG_TurboDemos_Code/** directory, and then open the directory that matches the version of CircuitPython you're using. Copy the contents of that directory to your **CIRCUITPY** drive.

```
# SPDX-FileCopyrightText: 2026 John Park with Claude Opus 5.5 for Adafruit Industries
#
# SPDX-License-Identifier: MIT
"""Boot into the demo menu (menu.py).

To run one demo directly while developing, set DEMO to its module instead:
voxel_demo, cornell_demo, metaballs_demo or meshes_demo. Keep this
file tiny: demos allocate their SRAM framebuffers first thing, and anything
imported before them uses up SRAM.
"""

DEMO = "menu"

__import__(DEMO)
```

https://github.com/adafruit/Adafruit_Learning_System_Guides/blob/main/CircuitPython_Turbo/CG_Turbo_Demos/CG_Turbo_Demos_Code/code.py

## Kiosk Mode

If you'd like the Fruit Jam to boot into a single demo instead of the menu as soon as you turn it on, you can simply change the `DEMO` variable in the code.py file and save it.

Change `DEMO = "menu"` to any of these:

- `DEMO = "voxel_demo"`
- `DEMO = "cornell_demo"`
- `DEMO = "metaballs_demo"`
- `DEMO = "meshes_demo"`

## How It Works

Each demo has three layers:

- CircuitPython for setup and the frame loop
- Compiled Turbo/Viper (**.mpy**) to pass data to the code
- Inline assembler ( `asm_thumb` ) for the per-pixel work

Text emphasized with a green exclamation:

**Turbo** a.k.a. **Viper** (@micropython.viper) is Python syntaxed with typed integers and pointers, and it's much faster than plain Python but slower than hand-written **asm** (see below). In the doc's VoxelSpace numbers, that's about 120 ms per frame for Viper versus 24–31 ms for asm.

Text emphasized with a blue exclamation:

**asm_thumb** is MicroPython's inline assembler. You put the decorator @micropython.asm_thumb on a function, and its body is written as ARM Thumb instructions in Python-style syntax instead of normal Python code: `@micropython.asm_thumb def add(r0, r1): add(r0, r0, r1) # r0 = r0 + r1; r0 is the return value`

The **.py** files in **CG_Turbo_Demos_Code/** run from source on the board. They handle setup, buttons, the camera/animation, and the frame loop.

The Turbo modules live in **turbo_source/**. **tools/build.py** compiles them with `mpy-cross -march=armv7emsp`. For modules that contain `asm_thumb` functions, it then runs Liz Clark's **tools/asm_to_viper.py** (). Stock CircuitPython loads Viper machine code but has no inline assembler, so that tool relabels each asm function as Viper.

A relabelled asm function can't take normal arguments, so every kernel is called the same way:

1. Python fills an int32 "args block" in SRAM with sizes, addresses and per-frame values.
2. A short Viper function turns offsets into absolute addresses and passes the block's address to the asm.
3. The asm finds that address at `[r3]`, untags it with `asr(r7, r7, 1)`, and reads everything else at fixed offsets from `r7`.

Every kernel comes in a pair. A readable Viper `render()` is the reference and runs on the board and on the desktop (with stand-in decorators in **tools/host/**). The asm version is written by a generator script, **tools/gen_*_asm.py**, and must produce identical pixels: edit the generator, never its output.

**display.py** is shared by all four demos. It creates the `picodvi` framebuffer (320x240 RGB565, pixel-doubled to 640x480) before anything else, so the back buffer gets fast internal SRAM. It also sets aside a `scratch` area of SRAM for each demo's hot data and warns if anything lands in slower PSRAM. `flip()` waits for vblank, then copies the back buffer to the screen faster than the display scans it out, so nothing tears.

## Voxel Space Demo Code

VoxelSpace is a Comanche-style terrain fly-over built from a grayscale height map and color texture map.

- **How it draws** For each screen column it marches out from the camera and draws a vertical span for each terrain sample it hits.
- **Terrain data** 16-bit samples (height plus a color index) with two smaller copies for distant terrain, stored in PSRAM.
- **Code** The whole frame is one asm call, written by gen_voxel_asm.py.
- **Buttons** The view modes, height scaling and fog all work by changing the input data, so the asm never changes.

The whole 320x200 frame is one asm call, which takes 24-31 ms in flight and holds a locked 30 fps; the Viper reference takes 117-130 ms.

## CircuitPython

**voxel_demo.py** creates the Display, with 8 KB of scratch, before importing anything else, then builds a `VoxelSpace`. Each frame, `flight.step()` moves the camera along a winding path and eases its height toward the terrain ahead. Then `space.render(screen.back)` draws the frame and `screen.flip()` shows it.

**voxelspace.py** does all the setup:

- **Terrain:** loads 672 KB into PSRAM. Each texel is a u16, height in the high byte and palette index in the low byte, with two smaller mip levels for distant terrain.
- **Distance table:** each step holds `z`, `scale / z`, a fog amount and a mip level, built in integer 16.16 math so the board and the desktop make the same table.
- **Each frame:** `view()` writes the camera position, heading vectors and horizon into the args block, and redraws the sky gradient only when the horizon moves.
- **Buttons:** none of them touch the kernel. The view mode swaps in another copy of the terrain plus a gray palette, height scaling changes a shift value, and fog-off writes zeros into the table's fog column.

## Turbo .mpy and asm

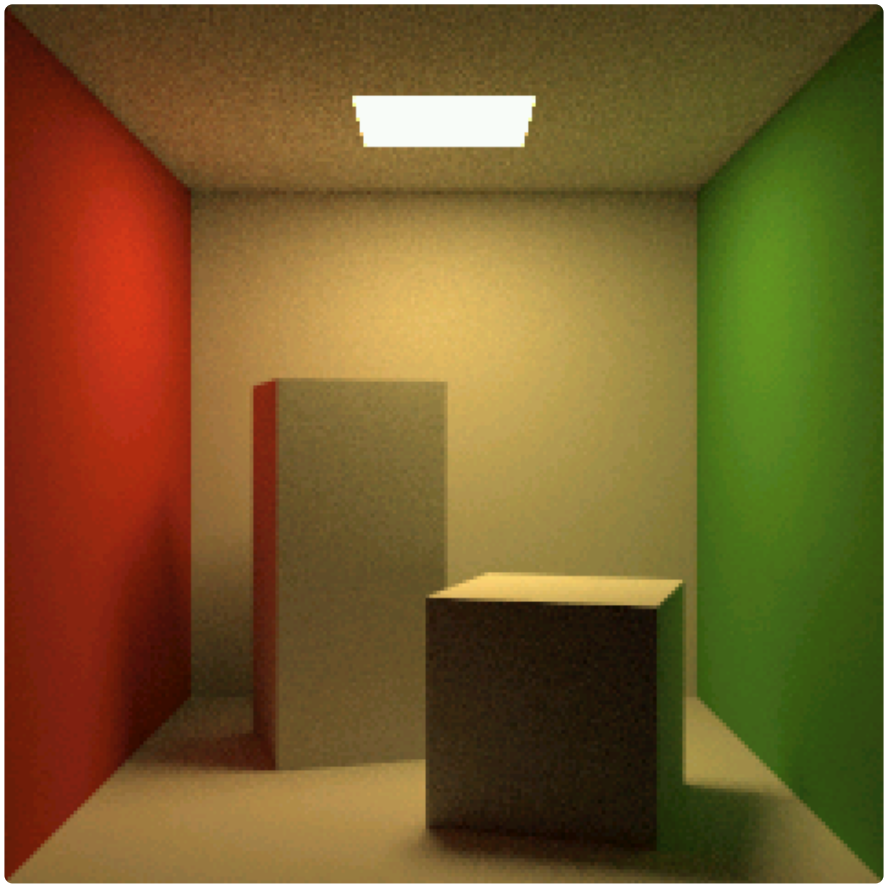
**voxel.py** holds `render()`, the Viper reference. For each distance step, near to far, it walks a row of 320 sample points across the map. It projects each sample's height to a screen row and, if that is above anything already drawn in that column, fills the span down to the previous top. A per-column "y buffer" tracks how far up each column has been drawn, and whatever is left at the top gets the sky.

`render_fast()` is a short Viper shim. It converts the block's offsets into absolute addresses in slots 0-31, because asm `ldr` offsets only reach 124 bytes, then calls `voxel_asm.frame_asm()` once. **tools/gen_voxel_asm.py** writes the inner loop out once per mip level, so its shifts are constants, each with and without the fog blend. Constants sit in `r8` - `r12`, leaving about 25 instructions per sample.

## Cornell Box Demo Code

Cornell box is a progressive path tracer at 240x240 that takes 0.73 s per pass.

- **Paths** Each path has up to 3 bounces. At every matte surface it also samples the ceiling light directly.
- **Accumulation** Results add up in a float32 buffer in PSRAM.
- **Accuracy** The asm uses the FPU and matches the numpy reference exactly. Scene constants are stored as float32 bit patterns so the board can't round them differently.
- **Boxes mode** It swaps the spheres for rotated boxes, which nearly doubles the pass time.



One asm call traces one sample for each pixel of a 240-pixel row on the RP2350's FPU, and a full pass over all 240 rows takes 0.73 s.

### CircuitPython

**cornell_demo.py** uses `Display(back=False)`: the tracer draws straight into the framebuffer, one row at a time. The outer loop runs one pass, adding one sample per pixel. It checks the buttons between rows, so a press restarts within one row instead of after a whole pass.

**cornell.py** keeps a 113-word parameter block in SRAM scratch and a 690 KB float32 accumulation buffer in PSRAM:

- **Each row:** `trace_row()` writes the row's accumulation address, output address and pixel index into the block, then calls the asm.
- **Scene: cornell_scene.py** is generated by **tools/cornell_scene.py**, with every float stored as its bit pattern, so the board never parses a decimal and maybe rounds it differently from the desktop.
- **Buttons:** wall color and spheres/boxes overwrite a few words in the block, then clear the accumulation buffer with `mem.zero()`. That Viper loop takes 98 ms; Python slice assignment took 10 s.

### Turbo .mpy and asm

This demo has no Viper kernel. The reference is **tools/cornell_ref.py**, a NumPy float32 tracer that does the same operations in the same order, so the board's pixels and sums match it bit for bit. `cornell_asm.trace_row()` comes from **tools/gen_cornell_asm.py** and is about 1,300 lines, nearly all FPU instructions (`vadd`, `vmul`, `vdiv`, `vsqrt`, `vcmp`).

For each pixel, the asm:

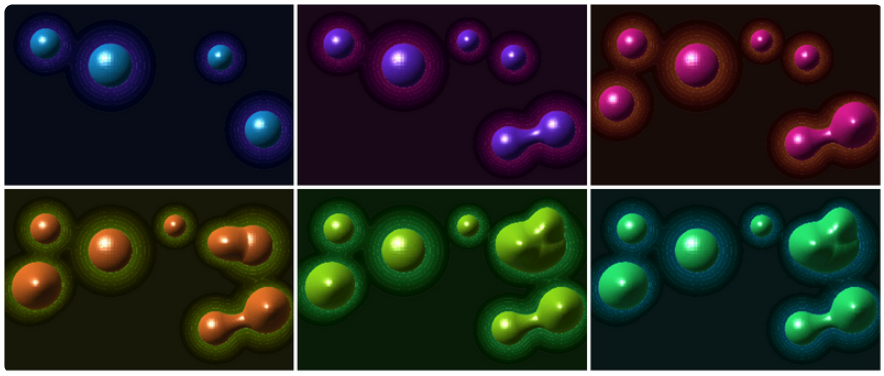
1. Seeds an xorshift32 random generator from the pixel index and pass number, and shoots a ray through a jittered point in the box's open front.
2. Finds where the ray leaves the box, then tests the two spheres. In boxes mode it does a slab test against each rotated box instead, at 6 divides each.
3. Reflects off the mirror sphere. On matte surfaces it samples a random point on the ceiling light, shadowed by the objects, then bounces in a cosine-weighted direction, for up to 3 interactions.
4. Adds the color into the accumulation buffer, scales by exposure / samples, clamps, takes a square root as a cheap gamma, and packs RGB565 to the screen.

The random generator and other shared steps are `bl` subroutines. Long jumps use `bl` too, because Thumb's short branches reach only about 2 KB. The kernel saves `s16`-`s31` into the block, because the caller expects them preserved.

## Metaballs Demo Code

Metaballs demo shows six to ten 2D balls at 320x200 and 30 fps, all in integer math.

- **The field** Each ball adds Wyvill's falloff and its gradient over its own bounding area.
- **Inside the blobs** The surface height is `sqrt(field − threshold)`, and its slope looks up a 64x64 shading table, which makes the blobs look round.
- **Outside** The field value picks a color from a glow palette with contour bands.



An all-integer field kernel draws six balls in 20 ms (the Viper reference takes 117 ms), and the demo holds 30 fps with up to 10 balls.

### CircuitPython

**metaballs_demo.py** follows the VoxelSpace pattern, with 14 KB of scratch. Each frame, `blobs.positions(t)` gives each ball's (x, y, radius) on its Lissajous path. `metafield.set_balls()` converts them to 1/16-pixel fixed point and precomputes `r²` and `(1 << 28) / r²` per ball, so the kernel never divides. Then it renders and flips.

**metafield.py** loads **metaballs.bin** into PSRAM: six hue sets of 9 KB, each a 256-color glow palette, a 64x64 lit-surface table and a slope-scale table. The current set is copied into SRAM next to the args, so changing the hue is one 9 KB copy. Turning the glow off overwrites the glow palette with its first color. The double-click on button 1 is plain Python with a 0.35 s timer.

### Turbo .mpy and asm

`metaballs.render()`, the Viper reference, works one row at a time:

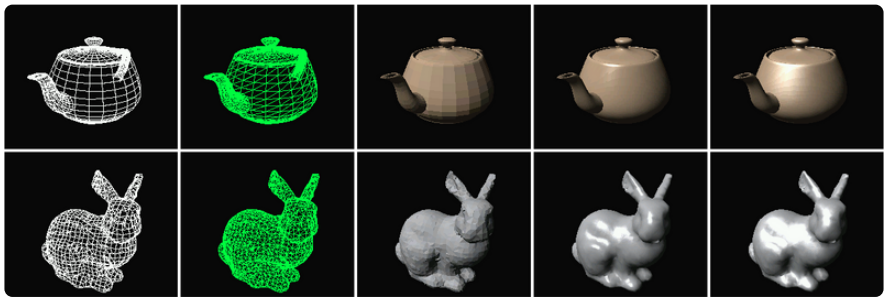
1. Clears three per-row sums: the field and its x and y gradients.
2. For each ball whose circle crosses the row, adds Wyvill's falloff `(1 - d²/R²)²` and its gradient across the ball's horizontal extent.
3. Inside a blob (field above the threshold), scales the gradient by a table value and uses it to index the 64x64 shading table, which gives the glossy 3D look.
4. Outside, uses the field value to index the glow palette, contour bands included.

`render_fast()` fills the address slots and calls `metaballs_asm.frame_asm()` once per frame. The generated asm from **tools/gen_metaballs_asm.py** stores the three row sums interleaved per pixel, so one pointer reaches all three, and clears them while shading so the next row starts clean.

## 3D Models Demo Code

**3D models** demo animates the Utah teapot and the Stanford bunny meshes at 30 fps, with multiple shader options.

- **Transform** Vertices are rotated and projected on the FPU.
- **Sorting** Back faces are culled, then the rest are bucket-sorted far to near and painted in that order (the painter's algorithm).
- **Filling** Triangles are filled using edge functions.
- **Lighting** Lambert, Phong and Blinn are 64x64 tables looked up by the interpolated normal. This works because the light is directional and the view is nearly orthographic.
- **Texture** Texture mode interpolates two values per 32-bit register so the inner loop fits in the available registers.



Three asm kernels run each frame (transform, sort, draw) and render the teapot or bunny in 14-31 ms; the Viper sort and draw alone took 49-83 ms.

## CircuitPython

**meshes_demo.py** uses a narrower 240x200 back buffer to free SRAM, plus 46 KB of scratch. Each frame it computes a rotation matrix (spin around y, a fixed tilt about x), renders and flips. The flip uses `mem_asm.copy_rows_asm`, because the narrow buffer goes in the middle of each 320-pixel screen row.

**meshview.py** loads the models (vertices, normals, face normals, triangle indices, quad edge masks and teapot UVs) and the 512x512 texture into PSRAM. It divides the SRAM scratch into four parts, each rounded up to 4 bytes so the next starts word-aligned:

- **geo:** the transformed vertices, the drawing order and the face normals.
- **args:** the parameter block for the sort and draw kernels.
- **Lighting table:** the current 64x64 table, copied in when the model or shader changes.
- **Transform block:** the floats for the transform.

## Turbo .mpy and asm

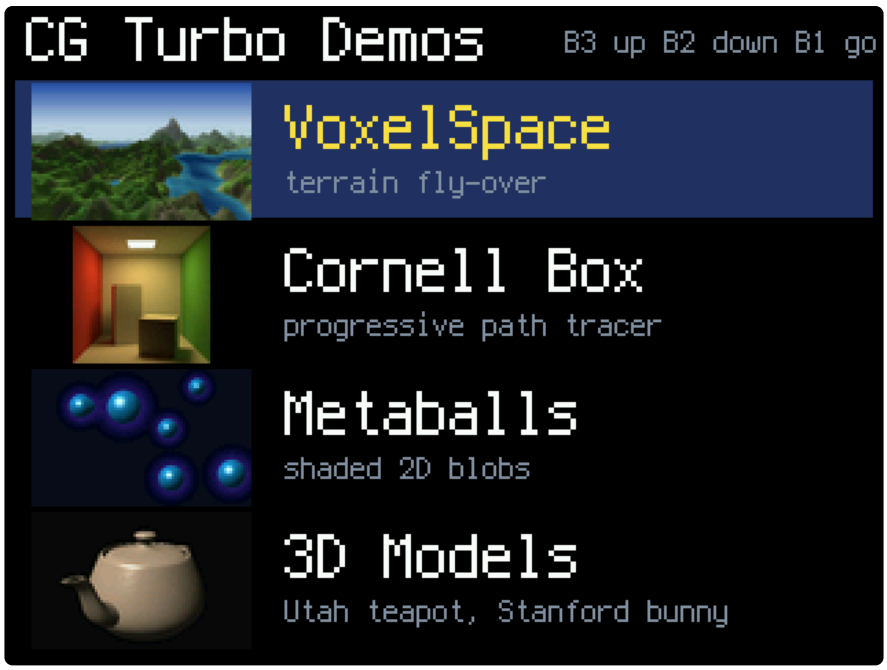
1. **Transform:** `mesh_asm.transform_asm`, FPU asm called straight from Python, rotates and projects every vertex. It writes 1/16-pixel screen x and y, an integer depth and the normal's x and y as u16s into geo, plus packed face normals for Lambert mode.
2. **Sort:** `mesh.sort` drops back faces (negative signed area on screen) and bucket-sorts the rest into 256 depth buckets, far to near.
3. **Draw:** `mesh.draw` clears the buffer and paints each triangle by mode:
    - **Wireframes:** Bresenham lines, using the edge mask in quad mode.
    - **Filled modes:** edge-function fills with normals interpolated per pixel. The normal's x and y index the lighting table, which works because the light is directional and the view nearly orthographic.
    - **Texture:** s, t and the normal are interpolated in wrapping 32-bit math, two values packed per register.

`mesh.sort` and `mesh.draw` are the Viper references. `render_fast()` fills a 32-word block and calls `sort_asm` and `draw_asm`, both generated by **tools/gen_mesh_draw.py**. The asm steps the edge functions instead of multiplying per pixel. In texture mode it computes each row's run of pixels directly with three `sdiv`s, and the paint loop uses all 8 low registers plus `r8`-`r12` and `lr`.

Every mode holds 30 fps except the 512x512 texture, which drops a frame now and then. All model/mode frames match **tools/mesh_ref.py** bit for bit.

## Menu Code

**Menu:** Each demo starts as a fresh run of its own file, using `supervisor.set_next_code_file` and then a reload. That way every demo can put its buffers in fast internal memory first.




# Setup and Run Demos

Plug the Fruit Jam into your display with the HDMI cable. Power the Fruit Jam from a 5V 1A USB C power supply, then turn it on. You can even power it from the USB port on some monitors, including the one linked in this guide.

The controls are driven by the three push buttons on the Fruit Jam, **BUTTON1**, **BUTTON2**, & **BUTTON3**. The board boots into a menu of the demos: button 3 moves up, button 2 down, button 1 starts the highlighted demo.

Each demo shows its buttons in the border around its image. When you're done with a demo, press the Fruit Jam reset button to restart the board and return to the menu.

## Menu Select

You can use the Fruit Jam **Button 3** and **Button 2** to navigate the four demos, the current selection is highlighted.

Press the Fruit Jam **Button 1** to start the selected demo.

Press the Fruit Jam **reset** button when you are done and you'll be returned to the main menu.

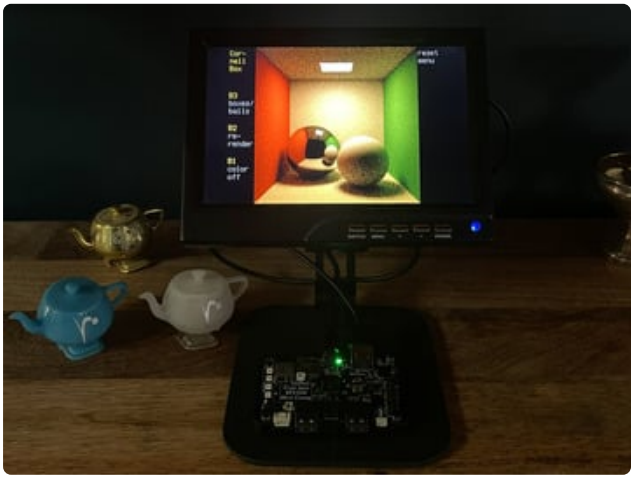
## VoxelSpace

**VoxelSpace**: a Novalogic Comanche-style voxel terrain fly-over on height-map generated terrain.

**Button 1: color map -> height map -> shaded height map**
**Button 2: terrain height: full -> 1/2 -> 1/4 -> 1/8 -> flat**
**Button 3: fog on / off**

s-macke/VoxelSpace () by Sebastian Macke (): the voxel terrain algorithm of NovaLogic's Comanche () (1992), explained.

Note, the learn guide author worked with this voxel engine on the Delta Force games, which were totally rad.



## Cornell Box Path Tracer

**Cornell box**: a progressive path tracer, 240x240, one sample per pixel per pass, refining on screen from noise to a clean image in a minute or two.

**Button 1: side walls white / back to red and green (starts over)**
**Button 2: start over from zero samples**
**Button 3: the original's two boxes / back to the two spheres (starts over)**

The Cornell box (), from Cornell University's Program of Computer Graphics: the scene and reference image the path tracer was compared against. Goral, Torrance, Greenberg and Battaile, "Modeling the interaction of light between diffuse surfaces" () (SIGGRAPH 1984).

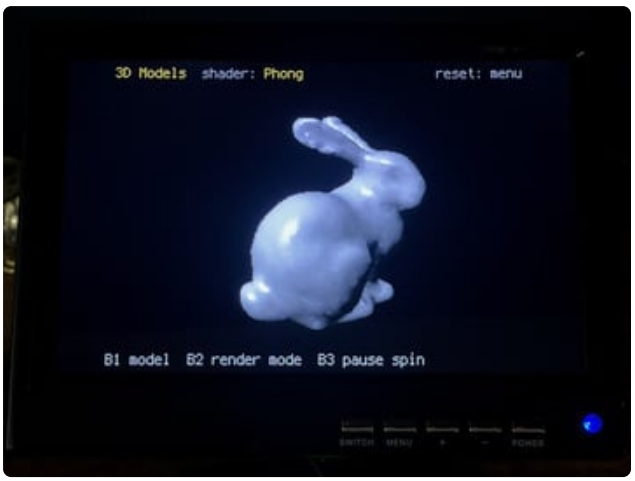
Jim Kajiya (), "The rendering equation" () (SIGGRAPH 1986): path tracing.

## Metaballs

**Metaballs**: six 2D metaballs gliding on Lissajous paths, shaded as glossy 3D blobs that merge, over a glowing contoured field.

**Button 1: next hue (six); double-click: glow off / on**
**Button 2: one ball fewer (down to 1)**
**Button 3: one ball more (up to 10)**

Jim Blinn (), "A generalization of algebraic surface drawing" () (ACM Transactions on Graphics, 1982): the blobby objects behind metaballs.




## 3D Models

**3D models**: the Utah teapot and the Stanford bunny, spinning at 30 fps. Quad patch, triangle mesh, Lambert, Phong, and Blinn shading, plus a bonus UV mapped texture demo.

**Button 1: teapot / bunny**
**Button 2: quad wireframe (white, the start) -> triangle wireframe (green) -> Lambert (flat) -> Phong -> Blinn -> texture (teapot only)**
**Button 3: pause / resume the spin**

The Utah teapot () is Martin Newell's 1975 Bezier patch data. The bunny is from the Stanford 3D Scanning Repository ().

**tools/make_meshes.py** downloads both and rebuilds **CG_Turbo_Demos_Code/teapot.bin** and **CG_Turbo_Demos_Code/bunny.bin**.

The texture is the UV test grid from the University of Utah's teapot page (). **tools/make_teapot_texture.py grid.png** rebuilds **CG_Turbo_Demos_Code/teapot_texture.bin** from it.