PicoVision: Hardware

PicoVision is a microcontroller-powered, HDMI stick, using two RP2040's - one Pico W as a "CPU" and one RP2040 as a "GPU" - to produce high (for a microcontroller) resolution HDMI output by swapping two physical - PSRAM backed - frame-buffers back and forth between them.

The two PSRAMs act as a front and back buffer. While the "CPU" (the Pico W) writes to one, the "GPU" (RP2040) reads from the other, applies some "hardware" effects and generates the HDMI signals.

- GPU Features
  - Display Resolutions & Pen Types
    - RGB555 or P5 Resolutions
    - Widescreen Modes
    - Pixel-doubled Resolutions
    - Frame vs Display
  - Sprites
    - Sprite blending
    - PicoVision Sprite .pvs Format
  - Scanlines
- GPU GPIO

GPU Features

Display Resolutions & Pen Types

PicoVision supports three pen types- RGB888, RGB555 and P5 (32 colour palette mode).

These require three, two and one bytes per pixel respectively. Generally the larger a pen mode (in bytes) the lower its max supported resolution.

RGB555 or P5 Resolutions

- 720 x 576 - 5:4 - PAL
- 720 x 480 - 3:2 - NTSC
- 640 x 480 - 4:3 - VGA

Widescreen Modes Some additional (P5 and RGB555) modes are supported with the optional widescreen build of PicoVision's MicroPython firmware:

- 800 x 450 - 16:9
- 800 x 480 - 5:3 - WVGA
- 800 x 600 - 4:3 - SVGA
- 960 x 540 @ 50Hz - 16:9 - 540p / qHD
- 1280 x 720 @ 30Hz - 16:9 - Limited compatibility with displays

:warning: Note that some of these modes run the GPU extremely overclocked, and also use a clock pulse width for the PSRAM that is outside of the range specified in the datasheet. So while these do work on most hardware, we can't guarantee they will work on all PicoVisions.

Pixel-doubled Resolutions PicoVision's "GPU" can pixel double along the horizontal or vertical axis, opening up a range of weirder resolution options that use half the width or height of a supported resolution and can support larger pen types.

If you're pixel doubling to a widescreen mode, you'll still need the widescreen build.

RGB888 is only supported when using horizontal pixel doubling. Other than that restriction, it can use all the same modes as RGB555 and P5.

Some examples:

- 320 x 240 - 4:3 (640 x 480) - HVGA - Good for DOS, NES, SNES, SMS, SMD
- 360 x 240 - 3:2 (720 x 480)
- 360 x 288 - 5:4 (720 x 576) - Close to CIF
- 400 x 240 - 5:3 (800 x 480) - WQVGA
- 960 x 270 - 32:9 (960 x 540)

These can be used to accomplish non-square pixels, for example if your screen aspect ratio is 16:9 and you output 960 x 270 your pixels will appear twice as tall as they are wide.

If you output 480 x 540 your pixels will now appear twice as wide as they are tall.

:warning: The actual shape your pixels appear will be a function of your mode's aspect ratio and your display's aspect ratio. Simply displaying a 4:3 mode such as 320 x 240 on a 16:9 display will stretch your pixels slightly horizontally.

Frame vs Display Since the PSRAM frame buffers are larger than necessary for storing frame data, it's possible to store a larger frame and display only a portion of it.

Sprites

The RP2040 serving as a GPU on PicoVision supports blending a handful of additional images over your frame data. These are called sprites. Sprites are currently only available in RGB555 and P5 modes.

A sprite refers to a single slot on the GPU, which is capable of displaying an image at a specific X/Y coordinate with vertical scaling and blend options.

The number of sprites you can display at once is governed by the size of the GPU's active sprite buffer. This buffer is 56kB (or 20kB in a widescreen build) and is used to load all of the active sprites into RAM during VSync so they can be blended into the frame scanline by scanline- chasing the beam.

There is, additionally, a hard limit of 80 sprites (32 in a widescreen build).

Each sprite can reference an image up to 4kB in size, this equates to 64x32 pixels in RGB555 mode. Larger sprites will occupy more of the active sprite buffer and permit fewer similtaneous active sprites. Roughly speaking, you can get about 80 16x16 sprites or 35 32x32 sprites on screen at once before starting to exceed the available clock cycles on the GPU - if that happens red lines will be displayed. Displaying multiple copies of the same sprite is slightly cheaper than displaying different sprites, and a given sprite image is only copied to the active sprite buffer once no matter how many times it's displayed.

Images larger than 2kB also occupy two consecutive image indices, for example a 64x32 RGB555 image (4kB) loaded at index 1 would occupy indices 1 and 2. Sprites 32x32 or smaller only use one image index.

Displaying this 64x32 image will have roughly the same cost as displaying eight 16x16 pixel sprites, limiting you to fewer consecutive onscreen sprites than the normal 80 (or 32) limit. You can balance this with smaller sprites to get back within the GPUs budget.

In summary:

1. There's a hard limit of 80 sprites on screen and a total of 56kB active sprite data
2. There's a hard limit of 10 simultaneous sprites per scanline
3. Each sprite can be up to 4kB in size, with a maximum width of 64 pixels and height of 32 pixels
4. Sprite images are stored in PSRAM in "indexed" locations
5. A single sprite, referenced by its "slot", specifies an index for the image data, X and Y coordinates, vertical scale and blend mode
6. Larger images can be used, but will occupy more of the 56kB active sprite data and so potentially limit you to fewer sprites

Sprite blending The blend mode on each sprite slot controls how the sprite image data is overlayed on to the canvas. For all modes except overwrite, the alpha bit in the sprite controls whether or not the pixel from the sprite is used.

The available modes are:

Mode	Meaning
OVERWRITE	The sprite overwrites the canvas with no checking of alpha bits

Mode	Meaning
UNDER	Each pixel from the sprite overwrites the corresponding pixel in the canvas if the alpha bit on the sprite is 1 and the alpha bit on the canvas is 0.
OVER	Each pixel from the sprite overwrites the corresponding pixel in the canvas if the alpha bit on the sprite is 1, regardless of the canvas alpha.
BLEND_UNDER	Each pixel from the sprite is blended with the corresponding pixel in the canvas if the alpha bit on the sprite is 1 and the alpha bit on the canvas is 0.
BLEND_OVER	Each pixel from the sprite is blended with the corresponding pixel in the canvas if the alpha bit on the sprite is 1, regardless of the canvas alpha.

The "under" modes allow sprites to go behind a part of the image that is on the canvas. The "over" modes allow sprites to go in front of the canvas regardless of the alpha setting. This allows you to create 4 depth levels in your scene, going from "back" to "front" they are:

1. Parts of the canvas image, drawn from PicoGraphics, with canvas alpha set to 0.
2. Sprites drawn in under mode
3. Parts of the canvas image, drawn from PicoGraphics, with canvas alpha set to 1.
4. Sprites drawn in over mode

The "blend" modes allow sprites to be drawn "semi-transparent". These are only valid in RGB555 mode. For each component of the colour (red, green and blue), the blend function takes the average of the sprite and canvas colours.

PicoVision Sprite .pvs Format If you want to streamline loading sprite image data, you can precompose your images into the following format:

```
Sprite entry:
  1 byte: Width
  1 byte: Height

  For each line:
    1 byte: Offset of first pixel on the line
    1 byte: Line width

  2 bytes: Padding if height is even
```

```
Height times:
  Line width times:
    Sprite pixel data (uint16_t ARGB1555, or uint8_t P5)
```

For P5 the internal pixel format for the pvs file has the palette entry in bits 6-2 and the alpha value in bit 0.

Scanlines

The GPU works with scanlines, "chasing the beam" to render sprites and apply scrolling effects on the fly.

During VSync, data is loaded into the active sprite buffer in GPU RAM (that's the internal memory of the RP2040 serving as the GPU) since it's not possible to stream it from PSRAM for each scanline at the same time as reading in the scanline data.

Additionally a series of sprite patch buffers - there are ten per scanline - are configured, pointing to the relevant sprite data for each scanline.

When the scanline is being written out to the screen, the patches from the sprite patch buffers are blended into the line immediately before it is encoded into HDMI symbols for the display.

GPU GPIO

As well as display driving functionality, the GPU RP2040 acts as an I2C GPIO expander, providing access to 9 GPIO pins on the header and the A and X buttons.

The buttons and pin 29 are on regular RP2040 GPIOs. The remaining GPIOs use pins on the RP2040 that are normally assigned to the flash and USB, but are also usable as GPIOs when not performing those duties, as is the case for the GPU.

The "high" GPIOs are accessed using pin numbers 0-7 through the `get/set_gpio_hi` family of functions on the C++ DVDisplay class. For MicroPython these pins and pin 29 are all accessed through the common `get/set_gpu_io` family of functions, using pin numbers 0-7 and 29.

Separate `is_button_x_pressed` and `is_button_a_pressed` functions allow you to check the button state.