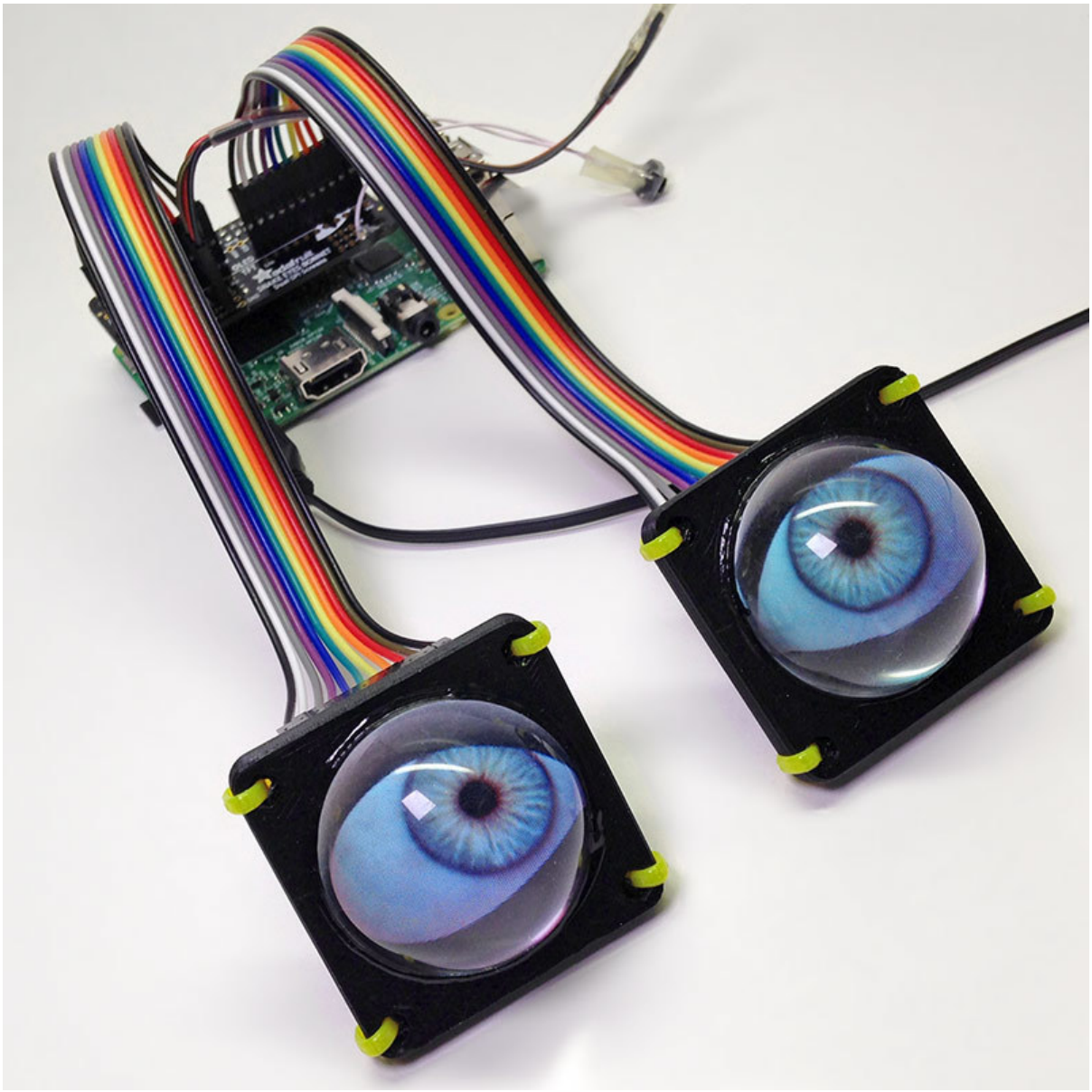
Animated Snake Eyes Bonnet for Raspberry Pi

Created by Phillip Burgess

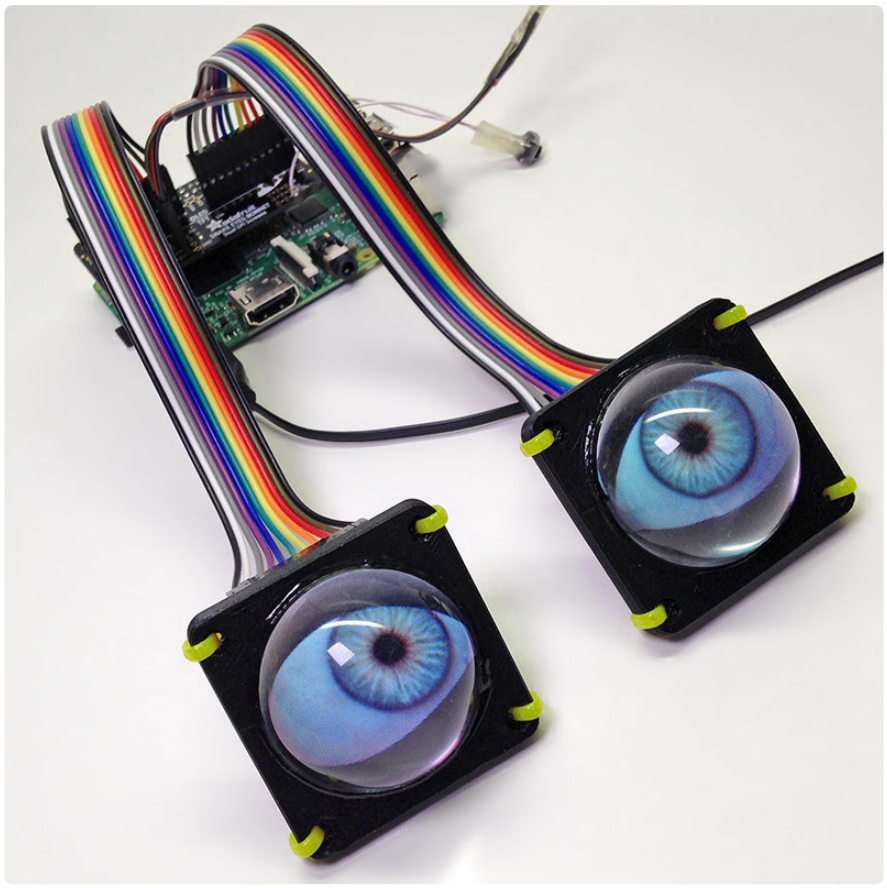


https://learn.adafruit.com/animated-snake-eyes-bonnet-for-raspberry-pi

Last updated on 2024-06-03 02:02:57 PM EDT

Table of Contents

Overview



The **Snake Eyes Bonnet** is a **Raspberry Pi accessory** for driving **two** small OLED or TFT LCD displays, and also provides four **analog inputs** for sensors.

It's fantastic for making cosplay masks, props, spooky sculptures for Halloween, animatronics, robots...anything where you want to add a pair of animated eyes!

It's a follow-on of sorts to another project: Electronic Animated Eyes Using Teensy 3.2 **(https://adafru.it/j6D).** The Teensy 3.2 is a very capable microcontroller, and the code for that project squeezed every bit of space and performance from it. I had been experimenting with the Raspberry Pi as an alternative...though still somewhat experimental, why not make that work available to others?

The Raspberry Pi offers some potential **benefits:**

- Hardware-accelerated **3D graphics** (OpenGL), including **antialiasing**.
- A **faster CPU**, ample **RAM** and dual SPI buses could yield faster frame rates.
- Standard graphics formats like **JPEG**, **PNG** and **SVG** can be decoded on the fly; **no preprocessing step**.
- The eye rendering code is written in a **high-level language** — Python — making it easier to customize.

And some possible **downsides** to the Pi:

- Raspberry Pi takes time to boot an operating system off an SD card, whereas Teensy is **instant-on** with all code in flash memory, . The Pi also requires an explicit **shutoff procedure** (usually (https://adafru.it/AJr)).
- The Raspberry Pi is **not as suitable** for **wearable** applications...it's larger, uses more power, and the SD card makes it less rugged.

This is a somewhat technical and not-inexpensive project. **Please read through everything first before commiting.** If it seems daunting, the original Teensy

Eyes (https://adafru.it/j6D) are more “Arduino-like” to build and customize, or other guides like Animating Multiple LED Backpacks (https://adafru.it/iwB) provide a more approachable introduction to code and electronics with less of an investment.

A Raspberry Pi 2 or greater is highly recommended. The code will run on a Pi Zero or other single-core Raspberry Pi boards, but performance lags quite a bit. Pi 4 works now, which was previously incompatible.

Hardware Assembly

Our Bonnet only comes with the PCB that lets you connect two displays. The displays are not included and must be purchased separately!

Some situations don’t require the screens or Bonnet at all! See the “Using Just the Software” page if planning a single, non-interactive HDMI eye.

Compatible Devices

- The code for this project works **only** with our 128x128 pixel OLED **(http://adafru.it/1431)** and TFT **(http://adafru.it/2088)** displays and 240x240 pixel IPS TFT **(http://adafru.it/3787)** displays. Other displays such as our 160x128 TFT or various PiTFT displays **ARE NOT SUPPORTED AT ALL**, period, not even sorta.
- Any recent Raspberry Pi board with the **40-pin GPIO header** should work. The very earliest Pi boards — Model A and B, with the 26-pin GPIO header — are not compatible.
- **A Raspberry Pi 2 or greater is highly recommended.** The code will run on a Pi Zero or other single-core Raspberry Pi boards, but performance lags quite a bit. Pi 4 works now, which was previously incompatible.

Plan a Head

Before committing to any particular hardware, think your project through. There are some decisions to be made...

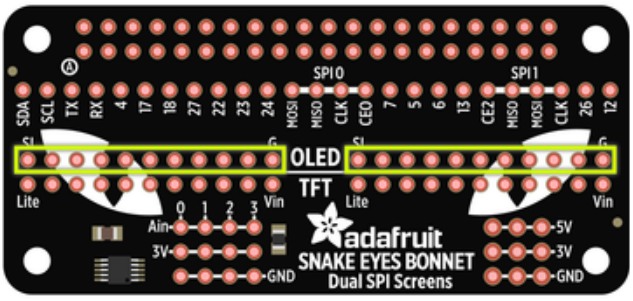
- **One** display or **two**? You don’t have to connect two displays...some of the most creative variants of the “Teensy eyes” had only a single eye (https://adafru.it/taM).
- OLED **(http://adafru.it/1431)**, TFT LCD **(http://adafru.it/2088)** or IPS TFT **(http://adafru.it/3787)** displays? OLEDs have a **wide viewing angle** and **excellent**

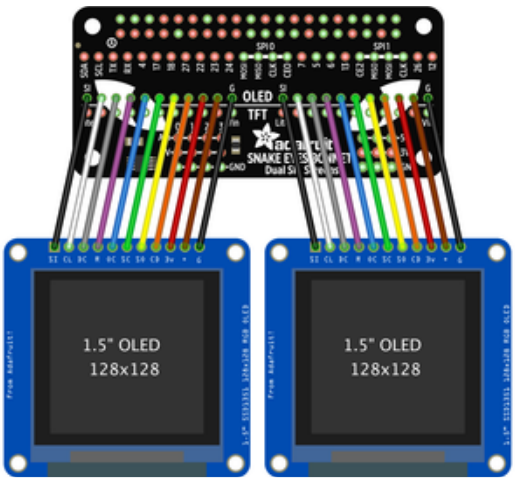
contrast and **color saturation**, but they're somewhat **pricey**, and have a **finite lifespan** (albeit many thousands of hours). **TFTs** make good **economy** displays if you're okay with the slightly **washed-out** appearance. **IPS TFT** displays are in-between in cost and super sharp!

- What model of **Raspberry Pi** to drive it? The latest **multi-core** boards have enough performance for **buttery smooth animation**...but their **size** and **power draw** might make them best for stationary displays, like maybe a Halloween window prop. Costume and portable installations may fare better with the diminutive **Pi Zero**, though the animation will be much less smooth.
- Will the animation be running **autonomously**, or do you plan to control the eyes with a **joystick** and **buttons**? Will the pupils react to **light**? These require additional components.

There's one more factor to consider: **how do you want everything connected?** Think about your intended installation. Is it temporary or permanent? Is space at a premium or do you have ample working room? These can influence your choice of wiring and connectors.

The breakout pins along the edge of each display board are wired up to matching pins on the bonnet boards. But you need the correct row for each display type...

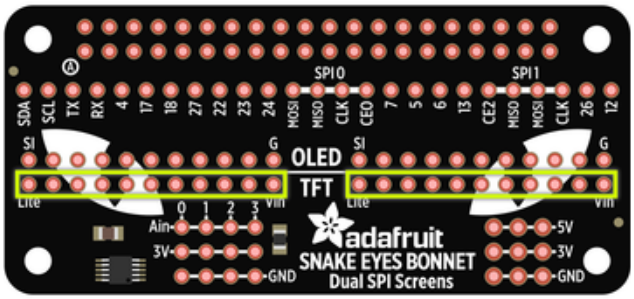


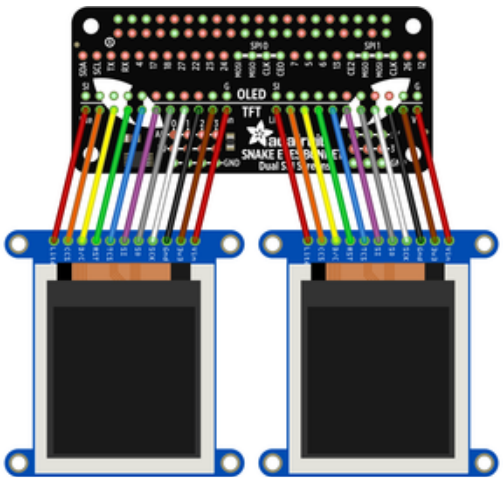


If using OLED display(s): use the "upper" rows (with the word "OLED" between them).

There are **11 pins** on the OLED breakout boards, which map directly to the 11 pins on the bonnet board.

Make absolutely certain the wires are in the same order. "SI" and "G" on the display board should go to "SI" and "G" on the bonnet, and each pin in-between...no wires should cross.





If using TFT or IPS display(s): use the “lower” rows (with the word “TFT” between them).

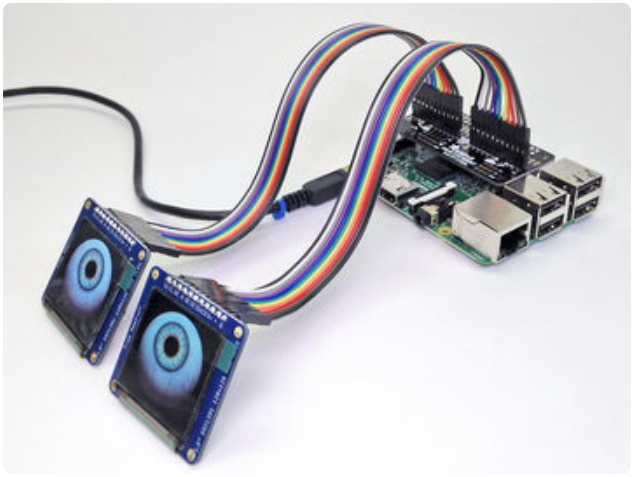
There are **11 pins** on the TFT breakout boards, which map directly to the 11 pins on the bonnet board.

Make absolutely certain the wires are in the same order. “Vin” and “Lite” on the display board should go to “Vin” and “Lite” on the bonnet, and each pin in-between... no wires should cross.

The 1.54" IPS display breakout has a 12th pin labeled "TE". DO NOT CONNECT THAT PIN TO THE BONNET. Only connect the other 11 pins from "Vin" through "Lite."

There’s a common trope in science fiction stories: that there is no “up” or “down” in space. Wiring these displays is a little like that...it doesn’t matter if the wires or headers come out the front or back of the display breakout board, or use straight or right-angle pins...as long as those wires get from the display to the bonnet in the **same positions and order**, everything’s good.

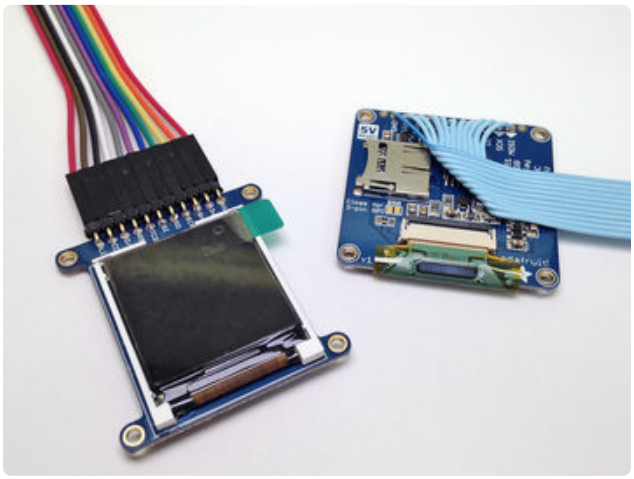
(As for up and down: right now the software assumes the displays are oriented with the **breakout pins along the top edge**; other rotations are not currently handled.)



Here's a probable layout for a non-portable installation. **Straight header pins** have been soldered to the bonnet board and the displays, projecting "up" and "back," respectively. Two 11-pin **female-to-female** ribbon cables then join everything.

This might be easiest to assemble (and disassemble), and the straight header pins on the displays make them easy to **re-use** in breadboard projects later.

...but that's not the only option.



This TFT display has **right-angle header pins** on the front of the board (front, back, doesn't matter as long as the wires connect in the same order). This makes it very slim, but also a fair bit taller.

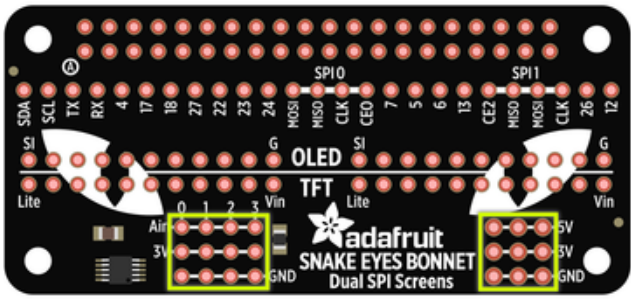
And this OLED has an 11-pin ribbon cable **directly soldered** to the board, no header pins at all. Again...front or back, ribbon could go straight out or can double back across the board...it all depends on your construction and space needs, as long as the pin order is followed. This is the most space-efficient, but requires patience and ace soldering skills, and isn't easily re-used in other projects.

If using "rainbow" ribbon cables: these have 10 colors, while the displays use 11 wires...this means the cables will have the **same color** wire along both edges. Therefore, **DO NOT rely on a visual mnemonic** like "black wire is ground" or "red wire is Vin," because your cable may have **two** black wires, or two reds, or two anything. Instead, make sure to **manually follow the first wire all the way from the bonnet to the display**, make sure they line up right, then install the remaining wires in order.

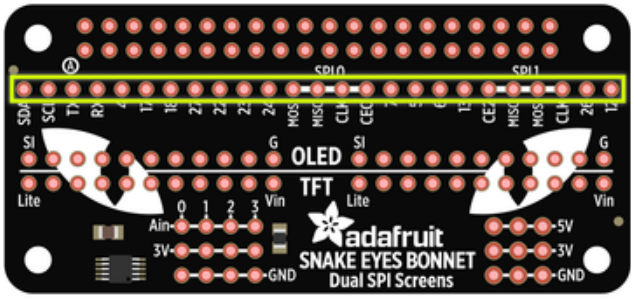
If you're really economizing for space, here's a secret: only 7 wires are really needed...it's just easier and less error-prone to solder a header at each end and plug all 11 wires straight through. If using the OLED display, the SC, SO, CD and 3V pins can

optionally be skipped. If using TFT, the 3v3 SO, CCS and Lite pins can be skipped. Make certain the exact same pins are skipped at the bonnet end, don't mix them up!

To ensure a clean signal from bonnet to displays, aim to keep your wiring **short and tidy.** Electrical interference can lead to animation glitches...we'll explain on the next page how to dial that back if needed. It's possible to use long ribbon cables (even a couple meters), but it invites problems with interference or signal reliability.



For optional features like joysticks, light sensors, blink and halt buttons, see the "Customizing the Hardware" page.



When finished soldering, you can **peel the clear plastic covering off the displays.** It's meant to protect them in shipping and during soldering. But unlike the screen protector on a phone or tablet, it's not optically pure and makes the display look cloudy.

The project thumbnail image shows lenses and **3D-printed enclosures**. These were a carry-over from an earlier project that used 1.5" acrylic cabochons (https://adafru.it/UB8), before we started carrying proper lenses (http://adafru.it/3917) which provide better optics and have a mounting flange...but we don't have enclosure designs for those, you'd need to model something yourself.

Software Installation

This software currently will not run on Raspberry Pi Bookworm.

We'll start by installing a suitable version of **Raspberry Pi OS** onto a SD card. If this is your first time or you need a refresher, we have a separate guide explaining the process (https://adafru.it/jd1) (opens in new window).

If using a Raspberry Pi 4, Pi 400, or Compute Module 4: the latest "**Bullseye**" Raspberry Pi OS **Desktop** software is required ("Lite" versions, and versions prior to "Bullseye" in late 2021, won't support this code on these boards). For brevity, we'll call all of these boards "Pi 4" going forward in this guide.

All other Raspberry Pi boards: Raspberry Pi OS Lite (Legacy) software is required. Look for both Lite and Legacy in the name!

For all boards: use the **32-bit** version of the operating system, not the 64-bit variant.

If your target system is a Raspberry Pi Zero, you may find the setup process easier with a spare full-size Raspberry Pi board with Ethernet and/or USB, then move the card over to the Pi Zero when finished. If this is not an option, see this guide (https://adafru.it/tbT) for steps to make the Pi Zero act as a "USB Ethernet gadget," and also create a file called "ssh" in the boot volume to enable ssh login.

If using the **Raspberry Pi Imager** application: from the "CHOOSE OS" menu, select a compatible operating system version as described above. Before writing the card, see "Enable SSH" below.

If using **Balena Etcher** or other application: download a compatible operating system image from the Raspberry Pi web site (https://adafru.it/XEz) (opens in new window). After writing the card, see "Enable SSH" below.

In either case, use the correct OS for the target system, as described earlier.

Enable SSH

It's usually easiest to set up and install this project remotely over a network, so you can copy-and-paste commands from this page to the command line.

If using the **Raspberry Pi Imager** application: before writing the card, press Control+Shift+X (Windows) or Command+Shift+X (Mac) to open the Advanced Options

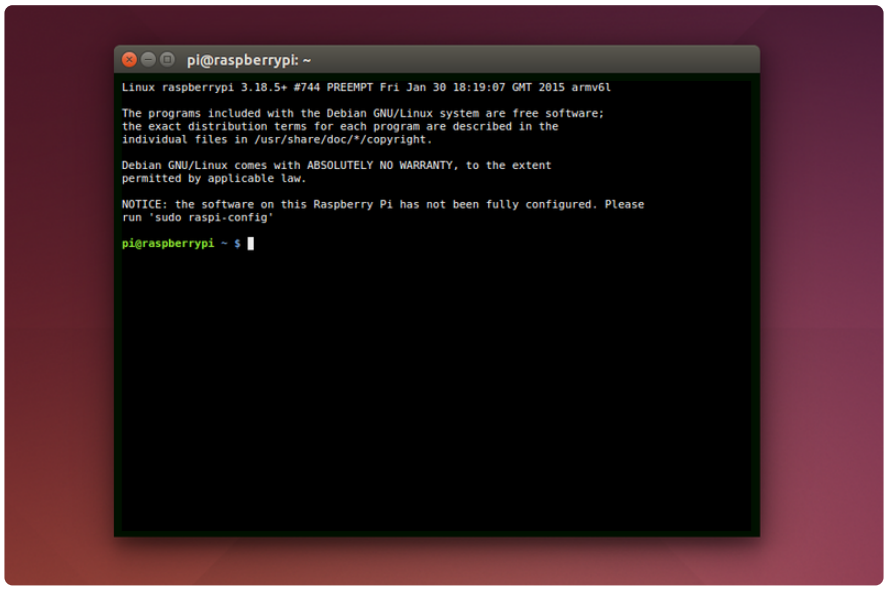
menu. Here you can **Enable SSH** for remote access. If you want to set up WiFi instead of a wired Ethernet connection, those options are also in this menu. Now you can write the card!

If using **Balena Etcher** or other application: after writing the card, don't eject! Create an empty file called **ssh** in the **/boot** partition. This will enable SSH over a wired Ethernet connection. If you want WiFi, you'll have to connect temporarily with Ethernet and configure wireless through **raspi-config**...or see below.

A third option—and required if you only have WiFi but no Ethernet—is to do this on the Raspberry Pi after first boot, with monitor and keyboard connected. The **raspi-config** utility includes options for enabling WiFi and SSH. Then the rest of this installation can be done remotely.

Once your Raspberry Pi is powered up and connected to a network you can follow the steps below to install the Pi Eyes software.

If you're familiar with connecting to the Raspberry Pi over SSH (https://adafru.it/jsE) you can use an SSH terminal application to connect and skip down to the install commands section below (https://adafru.it/uRC).



Install Commands

Run the following at the command line:

```
cd
curl https://raw.githubusercontent.com/adafruit/Raspberry-Pi-Installer-Scripts/
```

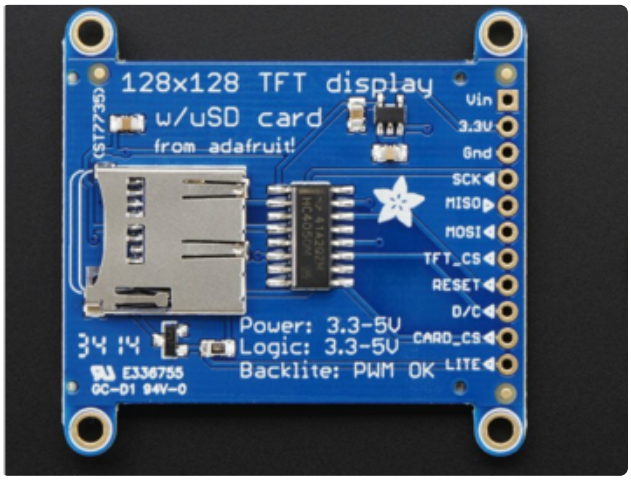
```
master/pi-eyes.sh &gt;pi-eyes.sh
sudo bash pi-eyes.sh
```

This downloads and runs a script which installs all the prerequisite software and does some system configuration. It will ask a few questions along the way...

- Is the target system a Pi 4-type board, or something else? Target system is an important distinction here — this refers to the Pi board where this SD card will ultimately be used, if it's different from the one where you're currently installing.

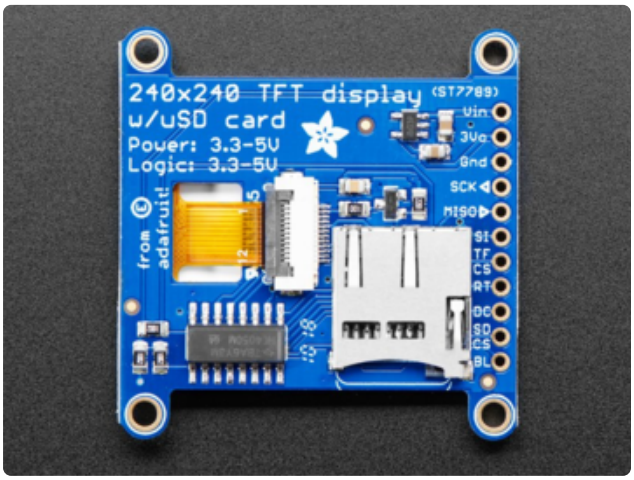
- Will you be connecting **OLED** (128x128), **TFT** (128x128) or **IPS** (240x240) displays? Can't mix and match; must be one or the other. (There's also an HDMI option — see the "Using Just the Software (https://adafru.it/zfJ)" page for guidance.)

These are the screen types:



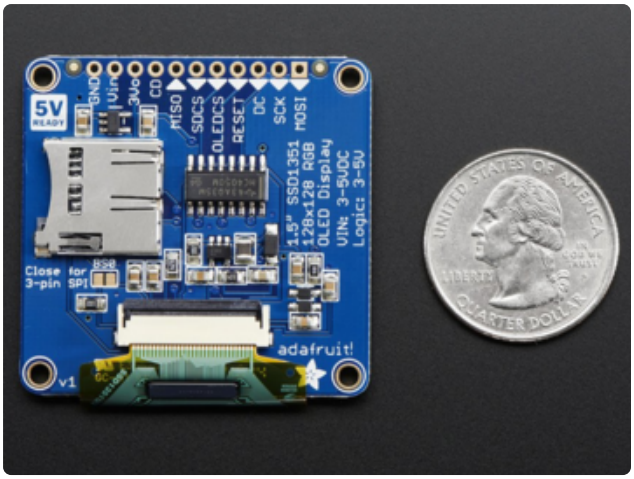
Select **TFT** if you have Product #2088

The 1.44" 128x128 plain TFT



Select **IPS** if you have Product #3787

The 1.54" 240x240 IPS TFT



Select **OLED** if you have Product #1431

The 1.5" 128x128 color OLED

- The GPIO-halt utility performs an **orderly shutdown** using a momentary pushbutton between an unused GPIO pin and ground, no login required. If you don't have a button and plan to shutdown the system the usual way, answer "**n**".
- Will you be connecting **ANALOG** input devices like a **JOYSTICK** or **LIGHT SENSOR**? If so, **answer "y" when asked "Install ADC support?"**

There's one final confirmation before installation, explaining the steps that will be taken and making it extra clear that the Pi will be dedicated to this one task, it should not be installed on a "daily driver" computer that you use for real work.

It can take a long time to download, build and install everything — ESPECIALLY NUMPY! Often 10 minutes or more. Be patient, it's working.

The installer has to collect software packages from all over the world. Very infrequently, a server or connection might be unreachable. If this happens, or the script "hangs" mid-download, set the project aside for a couple hours and try again later.

Dry Run

Reboot when prompted by the software installer. When the system restarts, after a minute or so, you should see some activity on the display(s).

The eyes run automatically on every startup; it is a dedicated system. If you don't want that, you'll need to edit /etc/rc.local and remove or comment out the line that includes "python3 eyes.py", typing the full command that appears there manually each time you want to start the code.

IT'S NORMAL THAT THE EYES MAY EXHIBIT "GLITCHES" ON THE FIRST TRY. We'll fine-tune some parameters to get them working right.

If everything seems to be working well, you can skip ahead to the next page and ignore the steps below.

If the eyes are experiencing glitches (video snow, tearing, dropped frames or weird inverted colors), here's what to do...

Log into the Pi remotely using ssh. Then type the following commands:

```
cd /boot/Pi_Eyes
sudo killall fbx2
```

The displays will stop updating. This is normal.

Then, if you have **OLED** displays, type the following:

```
sudo ./fbx2 -o -b 8000000
```

Or, for **TFT** displays, try:

```
sudo ./fbx2 -t -b 10000000
```

The first argument (**-o**, **-t** or **-i**) sets the display type in use; OLED, TFT or IPS, respectively. Second argument (**-b**) sets the **maximum bitrate** for the displays. The higher the bitrate, the smoother the animation...**but**...there's a limit to how fast this can go, and it can vary with wire lengths, connections, environment (such as interference from other nearby devices) and even slight manufacturing variances from one display to the next.

For **OLED** displays, the default bitrate is **10000000** (10 MHz). For **TFT** displays, default is **12000000** (12 MHz). **IPS** uses **96000000** (96 MHz) by default. But if there's trouble, we have to dial these back.

Try a lower value, like the 8 MHz or 10 MHz examples above. Watch the output for a minute...does it seem to have stabilized now? If only one of the two displays glitches, you'll still need to work the speed down until both run reliably.

Press **Control+C** to kill the program and test again with a different bitrate...maybe work down 4 MHz at a time, then up 1 MHz at a time until you find the "sweet spot"

between speed and reliability. Once you find it, Control+C again and let's make the change permanent...

```
sudo nano /etc/rc.local
```

A couple lines from the bottom you'll find this:

```
/boot/Pi_Eyes/fbx2 -o &amp;
```

(or "-t" if using TFT displays)

Insert the additional **-b** and **bitrate value** before the **&** character:

```
/boot/Pi_Eyes/fbx2 -o -b 8000000 &amp;
```

Save changes and **exit** the editor. Then:

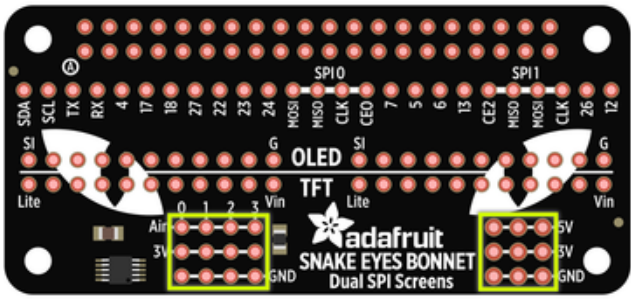
```
sudo reboot
```

After a minute or so the eyes should come up again, glitch-free this time. If not, repeat these steps again, trying a lower bitrate until you find a setting that works reliably.

Another way to reduce glitches is to **solder ribbon cables directly between the bonnet and displays, with no headers or plugs in-between;** every intermediary part is an opportunity for noise or connection problems. Consider this if you plan on permanent installation.

Customizing the Hardware

By default the eyes will animate on their own, looking around randomly. With some minor additional hardware (and enabling corresponding lines in the code), the eyes' direction, blinks and pupil dilation can be controlled manually or with sensors...

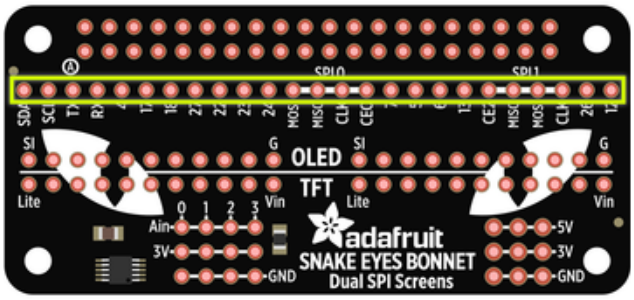


At the bottom-right of the bonnet are a few extra connection points for **5 Volts**, **3.3 Volts** and **ground**, if you have a circuit that needs them. These are OK for **small loads** like ICs or a few LEDs, but not for big things like servos, which will need their own power source.

To the left are four **analog input pins** (along with four more 3.3V and ground points). You can use these to interface analog circuits such as a **joystick** to steer the eyes, **photocell** to make the pupils react to light, or perhaps to monitor battery voltage (use a voltage divider in this case, since the analog input must be 0 to 3.3 Volts).

Woops, we swapped SDA and SCL on the silkscreen - apologies! SCL is the left-most pin, and SDA is the pin to the right of it!

Additionally, most of the GPIO pins are broken out in a single row across the bonnet. **Some of these serve special purposes and should be avoided,** but technical users still have access to them if really needed...



TX and **RX** are available as **GPIO14** and **15** if the serial console is **disabled** with raspi-config.
SPI0 (SPI bus used for the right eye) uses **GPIO8-11** (CE0, MISO, MOSI and SCLK, respectively). Steer clear!
SPI1 (second SPI bus used for left eye) uses **GPIO16** (CE2) and **19-21** (MISO, MOSI, SCLK). Avoid! Also, I2S audio devices can't be used because the pins overlap.
GPIO5 and **6** connect to the DC and RESET pins of both displays, so these too should be avoided unless you have some special situation.

Analog Controls

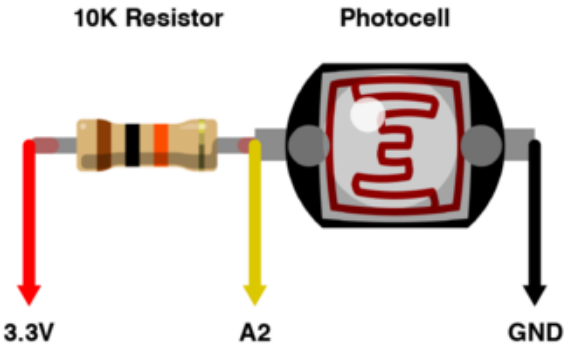
Any **analog** controls that are added should include connections to the **3.3V** and **GND** pins. Do not use the 5V pins or there will be...trouble.

XOUT and **YOUT** from a joystick can connect to analog pins **A0** and **A1**.

The eyes move autonomously by default — settings in the code enable the joystick instead.

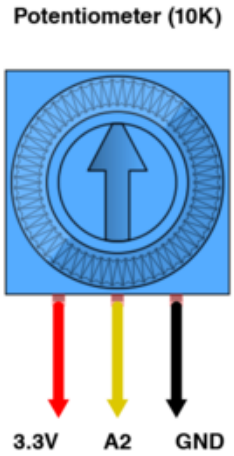
If you need to mount the joystick in a different orientation, there are also settings to invert each axis. Swap the X and Y pins in the code to use the joystick sideways.

This thumb stick has a click feature, which could be used to control eye blinks if desired. The smaller “mini” stick doesn’t have this, but is extra tiny for working into a costume or puppet.



To have the pupils contract or expand in response to light, connect a photocell and 10K resistor in series. The midpoint connects to analog pin **A2**.

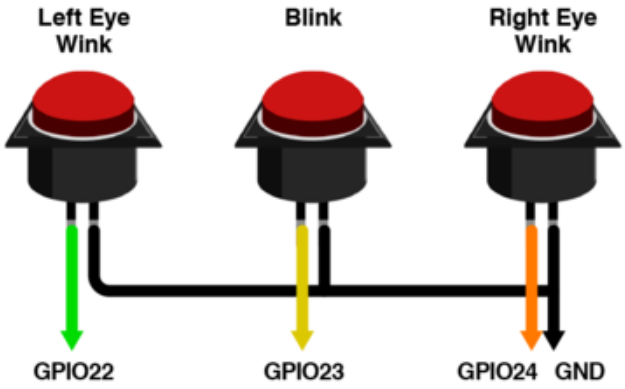
Analog input for the pupils (either photocell or the dial below) are enabled in the code by default. You can comment out IRIS_PIN in the code to have this move autonomously.



For manual control of pupil dilation (instead of responding to light) a 10K potentiometer can be used. The center leg connects to analog pin **A2** (same input as the photocell, just substituting a different analog control).

Buttons

The eyes normally blink autonomously, but you can also add one or more buttons to make them blink (or even wink individually) on command.



For all buttons, connect one leg of each to **GND**, and the opposite leg to a digital pin:

GPIO **Pin 22** is the **left** eye wink.
GPIO **Pin 23** blinks **both** eyes.
GPIO **Pin 24** winks the **right** eye.
If using our analog joystick breakout board, that stick includes a clicky button when you press down on it (on the SEL pin). This can optionally be used for manual blink control, or you can use a separate button for this (I find the joystick button a bit hamfisted).

Software Changes

Adjustments to the code must be made to use any of the above features. You'll find the code in **/boot/Pi_Eyes/eyes.py**. It's located in /boot to simplify "offline" editing on another system...if editing on the same Pi where it runs, you may need to edit as root (e.g. "sudo nano /boot/Pi_Eyes/eyes.py").

After making changes, you could hunt down the background Python process that was run at startup, kill and restart...but it's usually easiest just to **reboot**.

Hardware config settings can be found near the top of the code:

```
# INPUT CONFIG for eye motion ----------------------------------------------

JOYSTICK_X_IN   = -1    # Analog input for eye horiz pos (-1 = auto)
JOYSTICK_Y_IN   = -1    # Analog input for eye vert position (")
PUPIL_IN        = -1    # Analog input for pupil control (-1 = auto)
JOYSTICK_X_FLIP = False # If True, reverse stick X axis
JOYSTICK_Y_FLIP = False # If True, reverse stick Y axis
PUPIL_IN_FLIP   = False # If True, reverse reading from PUPIL_IN
TRACKING        = True  # If True, eyelid tracks pupil
PUPIL_SMOOTH    = 16    # If &gt; 0, filter input from PUPIL_IN
PUPIL_MIN       = 0.0   # Lower analog range from PUPIL_IN
PUPIL_MAX       = 1.0   # Upper "
WINK_L_PIN      = 22    # GPIO pin for LEFT eye wink button
```

```
BLINK_PIN       = 23    # GPIO pin for blink button (BOTH eyes)
WINK_R_PIN      = 24    # GPIO pin for RIGHT eye wink button
AUTOBLINK       = True  # If True, eyes blink autonomously
```

For example, to enable analog joystick input and a photocell, set JOYSTICK_X_IN, JOYSTICK_Y_IN and/or PUPIL_IN to analog channel numbers (0 to 3). If the response from the stick or sensor is backwards, set JOYSTICK_X_FLIP, JOYSTICK_Y_FLIP and/or PUPIL_IN_FLIP to “True” as needed.

Customizing the Look

Changing the Python Code

The installer script places the code and data in the directory **/boot/Pi_Eyes**. If you’re not familiar with Linux text editors and so forth, you can move the SD card over to a regular Windows or Mac system, where the “boot” partition appears as a drive and you can edit these files with your text editor of choice.

If using TFT or OLED screens on a Snake Eyes Bonnet board, the Python script of interest is **eyes.py**. If using HDMI output (with or without the bonnet), look for **cyclops.py** (so named because it only renders one eye).

After making changes to the code, rather than tracking down and killing processes, it’s often easier just to reboot the Pi. After a minute or so, the revised code will run on startup.

This section near the top of the code (previously mentioned on the Hardware page) includes a couple of items that are relevant to the appearance of the eyes:

```
# INPUT CONFIG for eye motion ----------------------------------------------

JOYSTICK_X_IN   = -1    # Analog input for eye horiz pos (-1 = auto)
JOYSTICK_Y_IN   = -1    # Analog input for eye vert position (")
PUPIL_IN        = -1    # Analog input for pupil control (-1 = auto)
JOYSTICK_X_FLIP = False # If True, reverse stick X axis
JOYSTICK_Y_FLIP = False # If True, reverse stick Y axis
PUPIL_IN_FLIP   = False # If True, reverse reading from PUPIL_IN
TRACKING        = True  # If True, eyelid tracks pupil
PUPIL_SMOOTH    = 16    # If &gt; 0, filter input from PUPIL_IN
PUPIL_MIN       = 0.0   # Lower analog range from PUPIL_IN
PUPIL_MAX       = 1.0   # Upper "
WINK_L_PIN      = 22    # GPIO pin for LEFT eye wink button
BLINK_PIN       = 23    # GPIO pin for blink button (BOTH eyes)
WINK_R_PIN      = 24    # GPIO pin for RIGHT eye wink button
AUTOBLINK       = True  # If True, eyes blink autonomously
```

Most relevant here are AUTOBLINK and TRACKING. If **AUTOBLINK** is changed to **False**, the eyes will stop their automatic blinking (responding only to button presses, if

you have something like that wired up). If **TRACKING** is changed to **False**, the eyelids will no longer follow the pupils as they move around (this is an interesting thing that real eyes actually do, but sometimes you want a continuous wide-eyed stare).

Sometimes you may want no eyelids at all...just a full unblinking hemisphere. You can achieve this by simply commenting out the two lines that render the eyelids (put a "#" character at the start of each line). This is much later in the code, near line 430:

```
        upperEyelid.draw()
        lowerEyelid.draw()
```

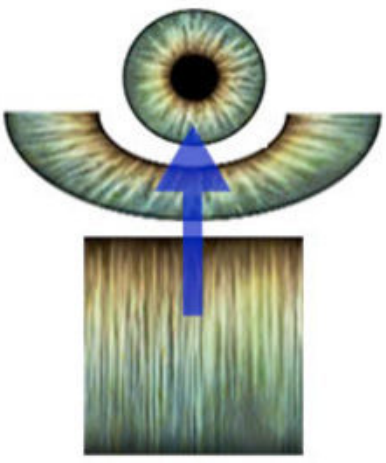
The other settings above are mostly related to hardware stuff.

Changing Graphics

Certain aspects (such as iris color) can be changed by substituting different graphics files. These are in the directory **/boot/Pi_Eyes/graphics**. One could just overwrite the files that are there, but I prefer to keep the originals around for reference and assign new names to the changed files. To make the code load these changed files, look for this section starting around line 122:

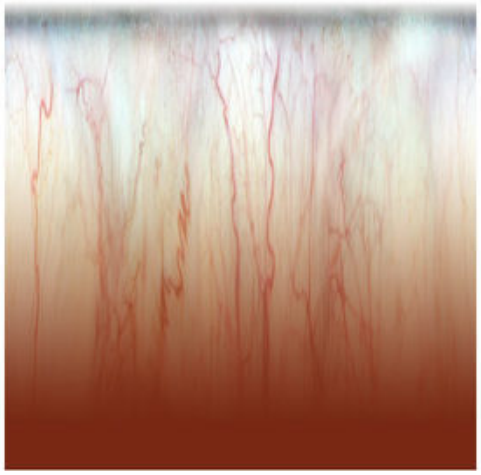
```
# Load texture maps --------------------------------------------------------

irisMap   = pi3d.Texture("graphics/iris.jpg"  , mipmap=False,
              filter=pi3d.GL_LINEAR)
scleraMap = pi3d.Texture("graphics/sclera.png", mipmap=False,
              filter=pi3d.GL_LINEAR, blend=True)
```

Most common graphics formats (JPG, GIF, PNG — the latter with or without transparency) are supported. The images are square to make the OpenGL library happy.



The graphics are stored flat and unrolled, like a map projection. The horizontal (X) axis works like the longitude, or angle around the eye, while the vertical (Y) axis is the latitude.

The **iris** (file iris.jpg) is what we think of as the “color” of the eye and is most often what you’ll want to edit. Sometimes you just need to edit the hue & saturation in a program like Photoshop, or you can make something totally custom if you’re after a particular look.



The **sclera** (file sclera.png) is the “white” of the eye...which really isn’t that white at all. There’s veins and blotches and gross stuff!

This file is a **transparent** PNG to simulate the transition where the sclera meets the lens...it’s not an abrupt transition, there’s a slight “fuzziness” to it. When editing, try to preserve that transparency (and note the couple of transparent rows at the bottom, necessary because of the way OpenGL interpolates these images).

(The third file, lidMap, probably shouldn’t be changed. It’s complicated.)

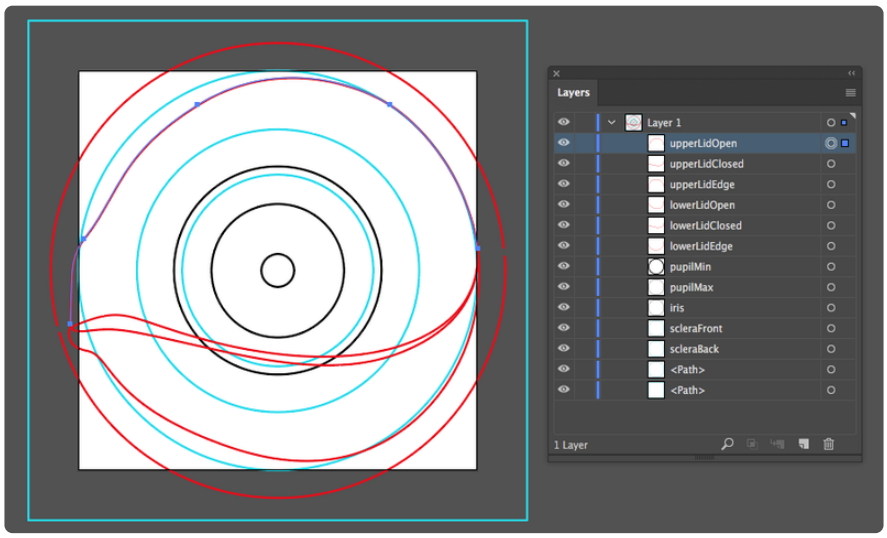
Around line 79 in the code is this:

```
# Load SVG file, extract paths &amp;amp; convert to point lists --------------------
dom               = parse("graphics/eye.svg")
```

eye.svg (or **cyclops-eye.svg** for the single-eye code) is a Scalable Vector Graphics (SVG) file that determines the size and shape of various elements. Suppose you want Krampus eyes, with that freaky horizontal goat pupil? Or dragon eyes with a vertical slit pupil? (In fact there’s an example file there for that — dragon-eye.svg.)

InkScape and Adobe Illustrator (among others) can both load and save SVG files. **But**...editing / substituting this file is fairly tricky, as the **names** of individual paths are used in the Python code. If your graphics editor of choice does not maintain these element names exactly when changing or saving the file, the Python code will fail to run.

In Illustrator, you can see the path names by toggling “Layer 1” open:



The largest blue circle there, which extends to the edges of the document, can mostly be ignored...it’s there for reference and represents the outer bounds of the eye ball (which can’t be changed).

The **iris** path (which needs to remain a circle, though you can change its size) determines the size of the outer edge of the iris relative to the whole eye.

pupilMin and **pupilMax** are the size and shape of the pupil in its most contracted and most dilated positions, as it responds to light. This doesn’t need to be a circle! Have a look at dragon-eye.svg for example.

Two additional blue circles — **scleraFront** and **scleraBack** — are used in determining the “sweep” of the white of the eye. Notice it overlaps the outer edge of the iris slightly, and is open at the back (we have no need for the back of the eye, so it’s not modeled or rendered).

Red paths are used for animating the eyelids. **upperLidOpen** and **upperLidClosed** are the shape of the upper eyelid in its fully-open and fully-closed positions. **lowerLidOpen** and **lowerLidClosed** are the same for the lower eyelid. **upperLidEdge** and **lowerLidEdge** are used by the software to generate the other edge of the eyelid mesh geometry, which is really a 2D polygon that occludes the eye behind it.

You’ll notice the default eye is **not symmetrical**. Eyes are interesting things and have a unique shape left and right! Looking at the SVG graphics, the other eye (and nose, if we had one) would be to the left. cyclops-eye.svg is left/right symmetrical...it’s designed for the Pi’s single HDMI output, which might be split to two identical

displays...the asymmetrical eye would look weird and lopsided in that case, so we go for the naïve “football shape.”

(At some point I hope to add a translucent nictitating membrane to the dragon eye, but this hasn’t happened yet.)

Replacing Everything

Well, almost everything.

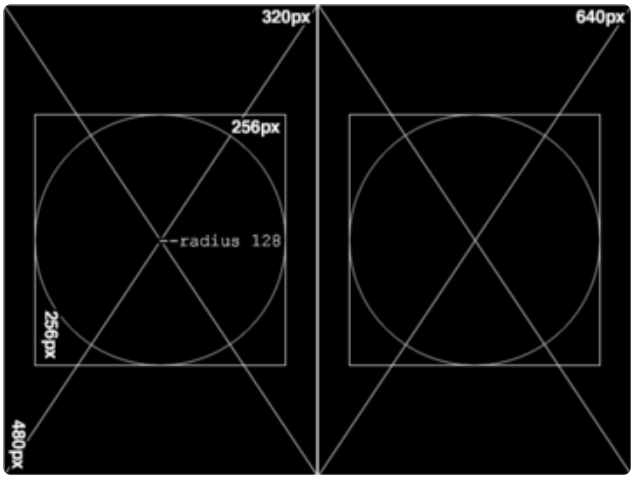
Our Python code generates OpenGL animation that goes to the Pi’s normal HDMI video framebuffer, whether or not there’s actually an HDMI display attached. A separate program — **fbx2** — continuously copies two sections of the framebuffer to the TFT or OLED displays attached to the Snake Eyes Bonnet.

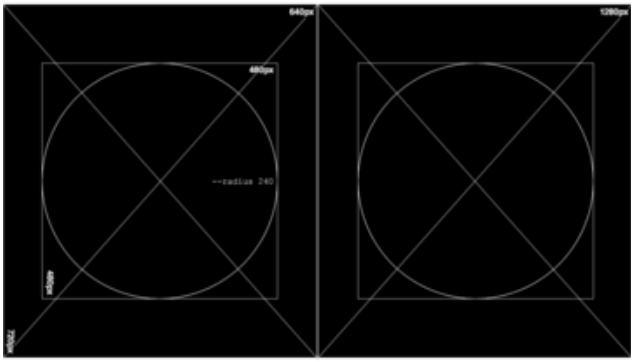
This means, if you really want, the entire eye-animating program could be replaced with code of your own design in whatever language you like. As long as your code positions graphics in the right places, fbx2 will present that on the TFT/OLED screens. (I’ve even tried installing fbx2 over RetroPie — yes, you can play Doom on these eyes!)

The pi-eyes.sh installer script configures the Pi’s video output resolution to either 640x480 pixels (for OLED or TFT) or 1280x720 pixels (for IPS). It could be nearly any resolution, but these sizes were chosen for a reason, explained in a moment...

Picture the screen split evenly; two rectangular regions side-by-side. The eyes will be centered within these regions, regardless their size or aspect ratio. The size of the rendered eyes is specified on the command line to the Python code using the --radius option, for example:

```
python eyes.py --radius 128
```





The fbx2 program performs repeated screen captures, scaling it by 50% (providing a smooth 2:1 downsampling in the process) and then copying two squares centered on these regions to the connected SPI screens. For OLED or TFT, the screens are 128x128 pixels, so the sections of the screen copied are 256x256 pixels, and the eyes have a radius of 128 pixels. IPS screens are 240x240 pixels, so the copied sections are 480x480 pixels and the eyes have a radius of 240 pixels.

There's a lot of unused blank space around the eyes; this is normal and by design. If developing your own code, this allows you to put debugging or status information in the margins of the HDMI display, but not have it copied to the SPI screens. Also, the eyelids are rendered by dynamically generating 2D meshes, and these need to be a little larger than the eyes to fully block things out. If the eyes were right up against each other, these would overlap.

Using Just the Software

The pair of TFT displays or OLEDs are great for life-size props or costumes. If you're looking for something larger than life, you can use the Raspberry Pi's video output to feed an **HDMI** monitor, video projector...or we've been having fun with the Gakken WorldEye, a 10 inch hemispherical display from Japan...

Using HDMI video out **doesn't require the Bonnet board**, unless you want **analog inputs** for a joystick control or a light sensor.

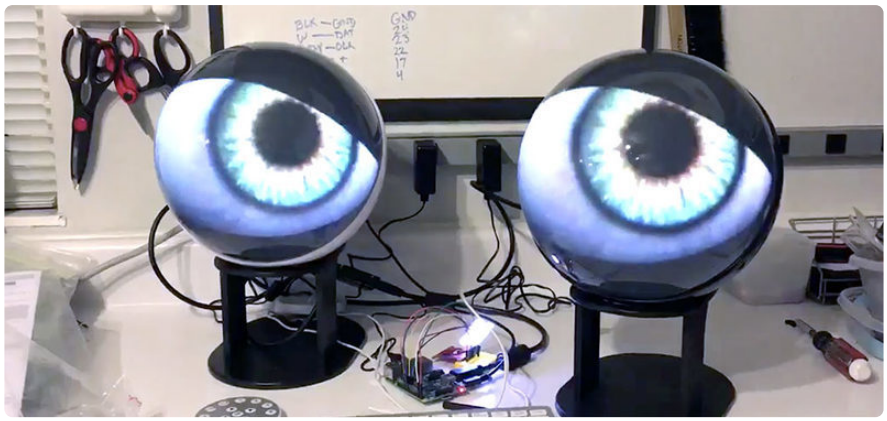
Follow the steps on the "**Software Installation** (https://adafru.it/zfK)" page — start with Legacy Raspbian Lite for a Pi 2 or 3, do some first-time configuration and get the system on your network. For Pi 4, start with 32-bit Legacy Raspberry Pi Desktop.

Download and run the **pi-eyes.sh** installer script as explained there.

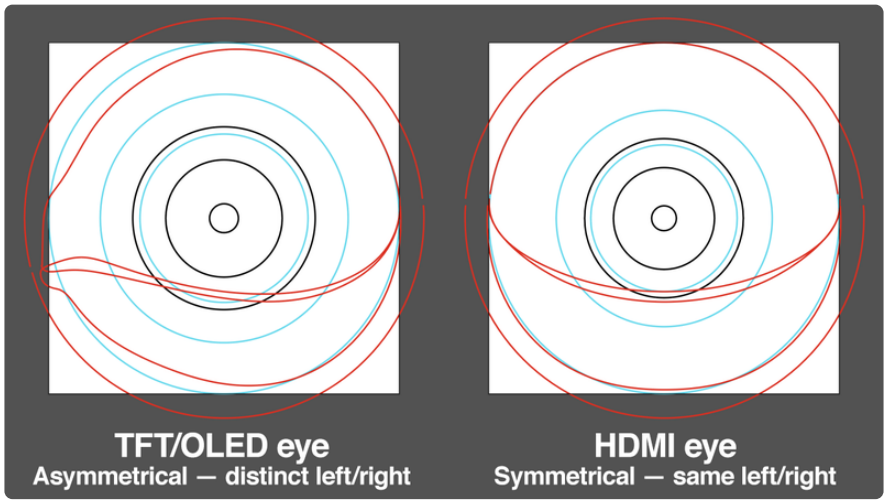
- When prompted for a screen type, select **HDMI**.
- Install GPIO-halt if you want a single-button shutdown option. This requires wiring up a button between one of the GPIO header pins and ground.
- Install ADC support only if you have the Snake Eyes Bonnet and want analog inputs for a joystick or light sensor.
- Install USB Ethernet gadget support only if using a Raspberry Pi Zero (or Zero W) and want a direct USB connection from your main computer. It's explained a bit in this guide (https://adafru.it/sUc).

After rebooting (about 30 seconds to one minute), you'll be greeted with a **single large eye** centered on the screen.

Use an **HDMI splitter** to feed the same image to multiple screens...



In order to work with the WorldEye display (or other HDMI devices), the code and graphics for this version are slightly different. The TFT/OLED code (eyes.py), designed for two separate physically-wired screens, provides a **distinct shape** for the left and right eyes to approximate the caruncle (the inner corner of the eyes). The HDMI code (cyclops.py) generates a single **symmetrical** eye (no carnucle), so it can be fed to an HDMI splitter and not look lopsided on two screens.



If using something other than a WorldEye display, and if the eye image looks oddly stretched and not round, this probably has to do with the screen **aspect ratio**. We can adjust the Pi's video output resolution to match your display's native HDMI resolution:

```
sudo nano /boot/config.txt
```

(Or substitute your editor of choice for "nano")

Look for this line near the end of the file:

```
hdmi_cvt=640 480 60 1 0 0 0
```

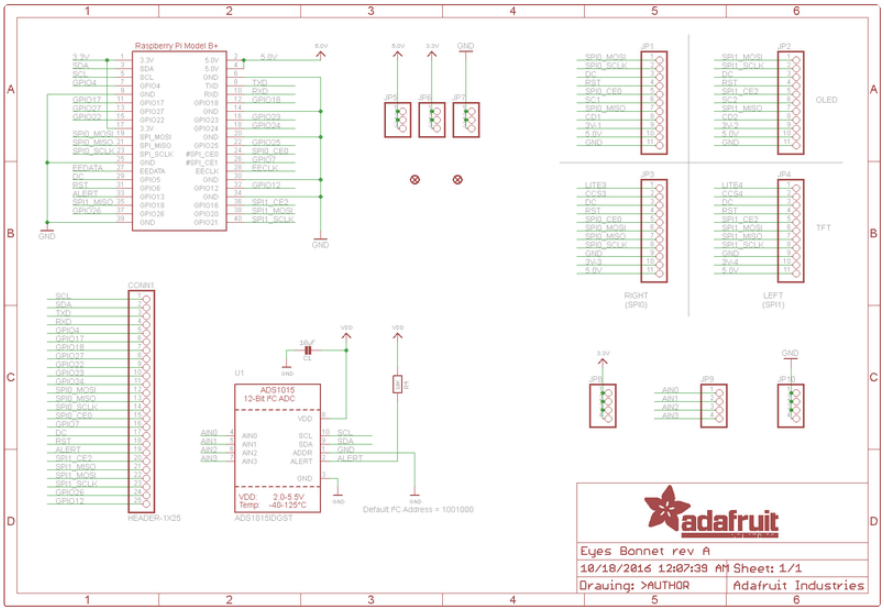
Change the first two numbers — 640 and 480 — to the desired resolution. Save the changes and then **reboot**.

Downloads

Files

- EagleCAD PCB files on GitHub (https://adafru.it/tDt)
- Fritzing object in Adafruit Fritzing library (https://adafru.it/aP3)

Schematic



Fab Print

Dims. in mm