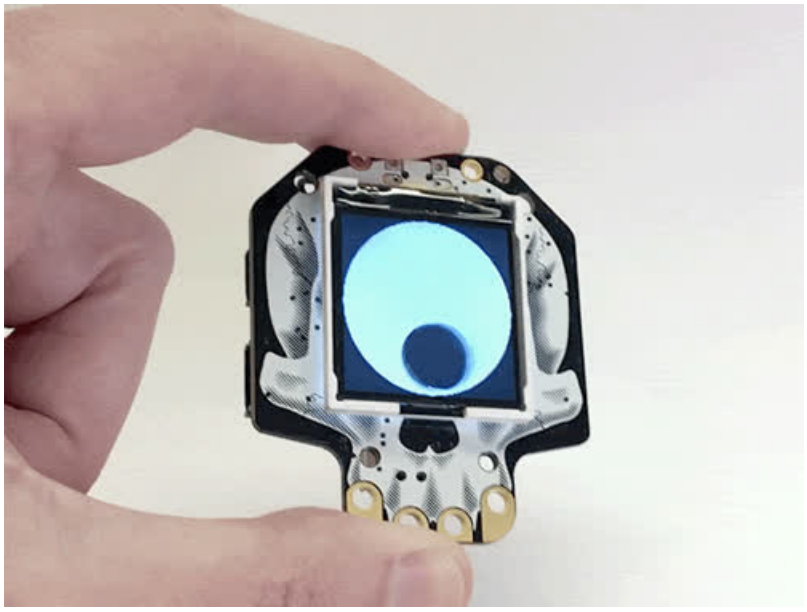
adafruit learning system

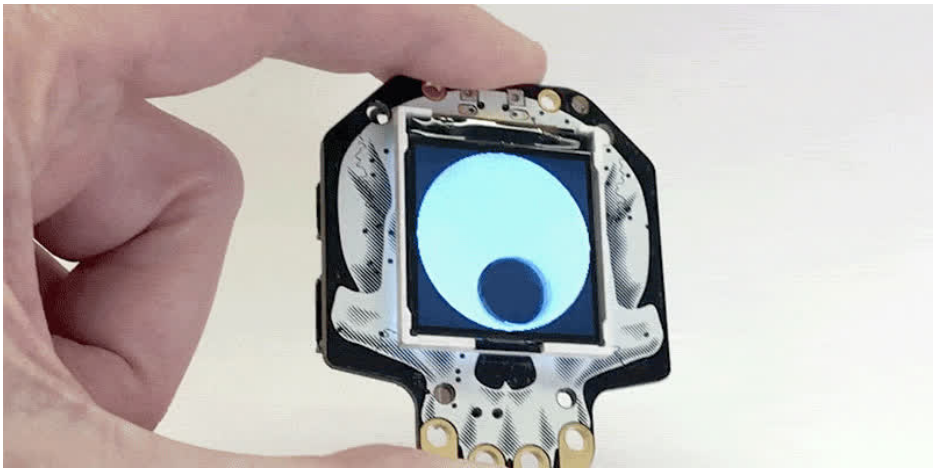
HalloWing Googly Eye

Created by Phillip Burgess



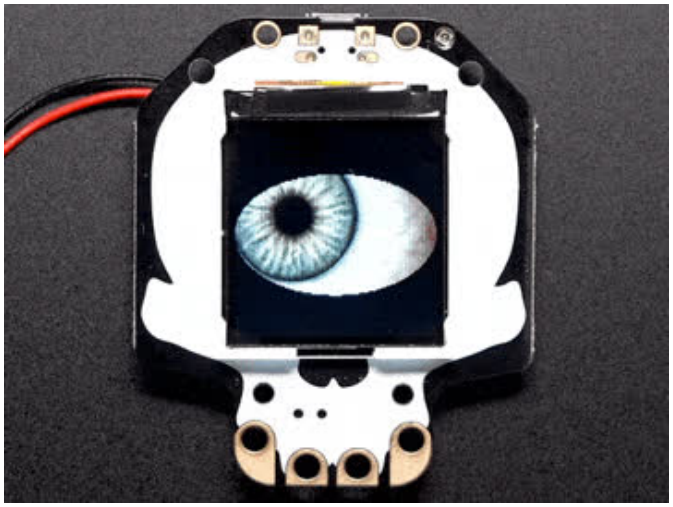
Last updated on 2020-05-19 12:30:41 AM EDT

Overview



In the pantheon of corny gags — rubber chickens, whoopee cushions and so forth — there's a special place for ***googly eyes,*** that staple of childrens crafts and tacky "You're HOW old?" birthday cards.

Photo credit: Googly Punch by Lenore Edman on Flickr, CC-BY-2.0 license. (https://adafru.it/CXM)



We've done *realistic* eyes before...it's the default program that ships on the HalloWing. (https://adafru.it/CEs) Suppose we desire something more...*zany?*

HalloWing has an accelerometer and can respond to motion...let's make it into a googly eye!

Everything needed for this project is built into the HalloWing board, no extra components are required. (A battery can optionally be added to make it self-contained and portable.)

Your browser does not support the video tag. Adafruit HalloWing M0 Express

$34.95
IN STOCK

Add To Cart

Software

Easy Way

If you want to get started **quickly**, download the UF2 file linked below. Turn on HalloWing and connect a USB cable to your computer. Double-click HalloWing's reset button, wait for the **HALLOWBOOT** drive to appear, then drag the UF2 file to this drive. After a few seconds, the code should be finished transferring and will run.

https://adafru.it/CYI

https://adafru.it/CYI

Here's a color variant if you'd prefer, modeled after a popular sports mascot:

https://adafru.it/D4A

https://adafru.it/D4A

This will overwrite CircuitPython if it's currently installed on your board (but your CircuitPython code and any libraries are safe).

You can **restore CircuitPython** easily by **following the directions here** (https://adafru.it/CmJ).

Build From Source

Building the project from source gives you the opportunity to **customize** the physics a bit.

This requires the **Arduino IDE** software for your computer and **Adafruit SAMD board support**, as **explained in this guide** (https://adafru.it/Cpl).

Several libraries are also required, which can be installed through the **Arduino Library Manager**(Sketch→Include Library→Manage Libraries...):

- Adafruit_LIS3DH
- Adafruit_GFX
- Adafruit_BusIO
- Adafruit_ST7735
- Adafruit_ZeroDMA

https://adafru.it/CXF

https://adafru.it/CXF

A few definitions near the top of the code determine how things behave...

```
#define G_SCALE       40.0   // Accel scale; no science, just looks good
#define ELASTICITY     0.80  // Edge-bounce coefficient (MUST be <1.0!)
#define DRAG           0.996 // Dampens motion slightly
```

`G_SCALE` influences the pull of gravity; higher numbers = stronger gravity.

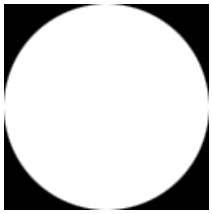
`ELASTICITY` determines the “bounce” when the pupil hits an edge. 1.0 = no energy lost in the bounce, 0.0 = complete stop. This should be set somewhere *between* these two values, not-inclusive.

`DRAG` slows the pupil movement from frame to frame, helping it come to a stop at the bottom of the eye. This too should be between 0.0 and 1.0.

The eye graphics are not easily customized, but the next page explains some of the program’s internals which experienced programmers might be able to work from.

How it Works

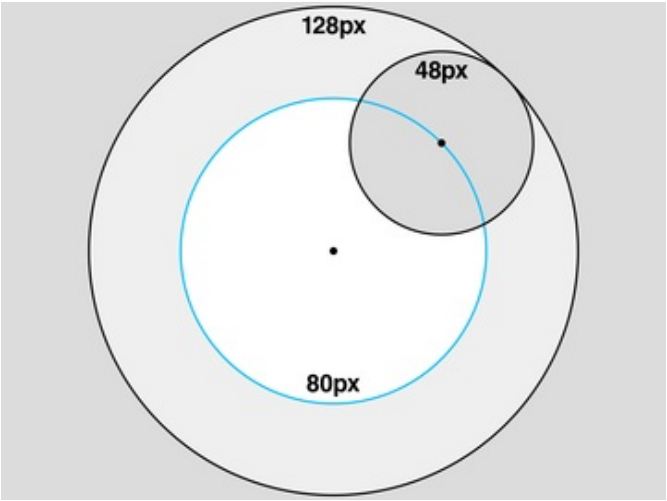
The code uses two grayscale images, nicely **antialiased** so they're not visibly "jaggy." The first, 128 by 128 pixels (matching the HalloWing screen size) represents the outside boundary of the googy eye:



The second, 48 by 48 pixels, represents the pupil. It's been drawn with a bit of a gleam because *specular highlights* make everything at least three times funnier:

The two images are embedded in the code (in the file `graphics.h`) as *uint8_t* (unsigned 8-bit) arrays. The conversion was done with some throwaway Python code. Each byte represents a brightness from 0 (black) to 255 (white).

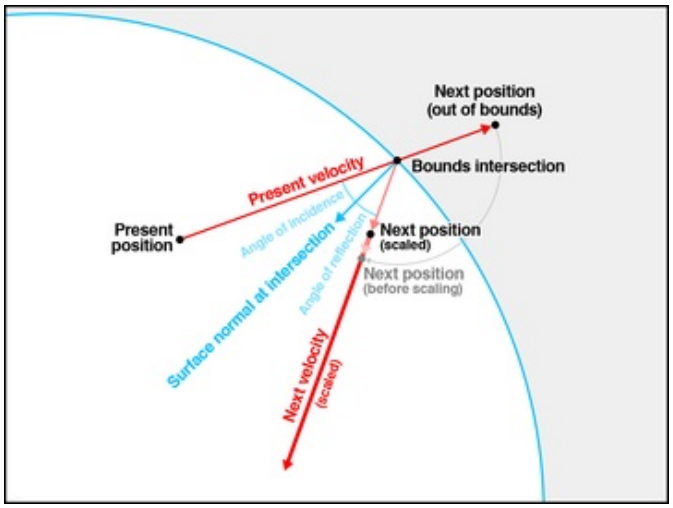
Getting the pupil to "google" properly took some doing. Making something to bounce in a rectangular space is easy — just reverse horizontal or vertical motion where an edge collision occurs — but in this googly setting, a collision may occur at *any* angle.



The fact that the eye and pupil are both perfect circles makes a **shortcut** possible...

Rather than calculating the intersections between two circles (the 128-pixels-across outside boundary, and the 48-pixels-across pupil), we can more easily process this as a *single point* moving within an 80 pixel wide circle (the 128 pixel boundary minus the 48 pixel pupil).

You'll see constants for these `#defined` in the code, though it's using radii (half these measurements) rather than diameters (the aforementioned pixel dimensions). Using a Cartesian coordinate system with (0,0) at the screen center also helps.



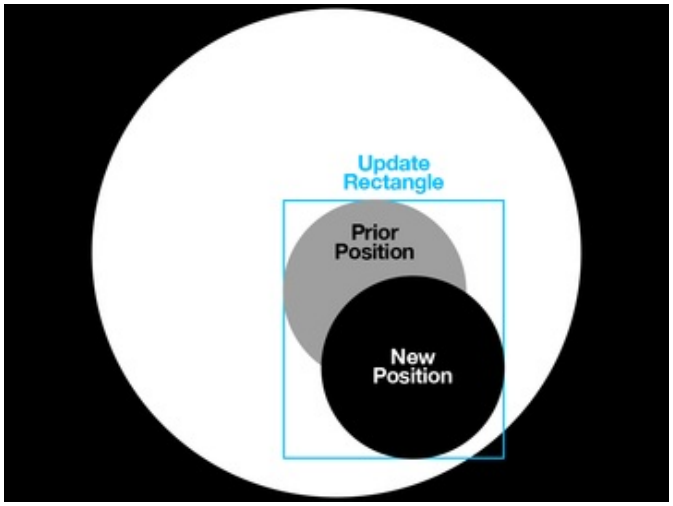
As each frame of animation is processed, the pupil's vector of motion is modified by input from the accelerometer, plus a tiny bit of drag or resistance to keep things from sliding forever.

If that vector of motion would have the pupil's center point crossing outside the 80-pixel threshold, the path of motion is reflected back into the eye at a suitable angle, and both the position and velocity are scaled back slightly to represent an inelastic collision (where some kinetic energy is lost due to internal friction), else it would never stop bouncing.

The pull of gravity and these other coefficients can be changed near the top of the code:

```
#define G_SCALE       40.0   // Accel scale; no science, just looks good
#define ELASTICITY     0.80  // Edge-bounce coefficient (MUST be <1.0!)
#define DRAG           0.996 // Dampens motion slightly
```

The `G_SCALE` value isn't scientifically determined...it just seemed a reasonably "realistic" value with some trial and error. `ELASTICITY` and `DRAG` also aren't based on any specific real-world material physical properties, again just trial and error with what felt "real." You can fiddle around with these, but the latter two values should not be equal or greater than 1.0 (else the pupil will *pick up* energy rather than settling at the bottom).



Each frame, the code updates the minimum **bounding rectangle** of the pupil's old and new positions. The whole rectangle is **drawn in one pass**, rather than erasing the old pupil and then drawing the new one, which would exhibit a lot of **flicker** and destroy the illusion.

Though the HalloWing's TFT display supports only 5 bits for red and blue, and 6 bits for green, the boundary and pupil images are stored using 8 bits per pixel. We're not pressed for space and the math for overlaying the two images is just easier that way. The conversion to "565" color is performed as each 8-bit value is processed.

Unlike the original "spooky eye" code, which allows two HalloWings to communicate and stay in sync (https://adafru.it/CWz), this project *intentionally* does not operate like that. The desynchronization of two eyes reacting to slightly different inputs adds to the hilarity — it's exactly how googly eyes should be!

Color Eye

The principle behind the color eye's operation is exactly the same, only the graphics are different. The code is a bit

more complicated in this case because it has to blend two RGB images rather than grayscale, but the code shares a lot in common.

The color eye is enabled near the top of the code, where it normally looks like this:

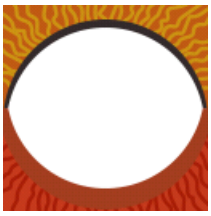
```
#include "graphics.h"
//#include "gritty.h"
```

Just switch out the comment characters ("//") so the second line is enabled rather than the first:

```
//#include "graphics.h"
#include "gritty.h"
```

This tells the compiler to use a different graphics file, where a flag is also set to enable the color drawing code.

There's the same 128x128 size background, but now in color:



And the pupil is just slightly larger than the grayscale version, now 54 by 54 pixels: