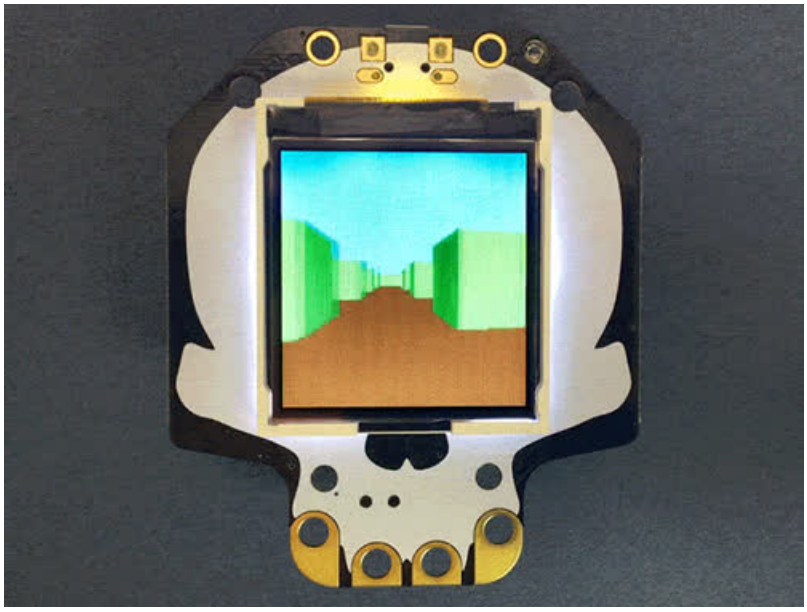
adafruit learning system

Hallowing Minotaur Maze

Created by Phillip Burgess



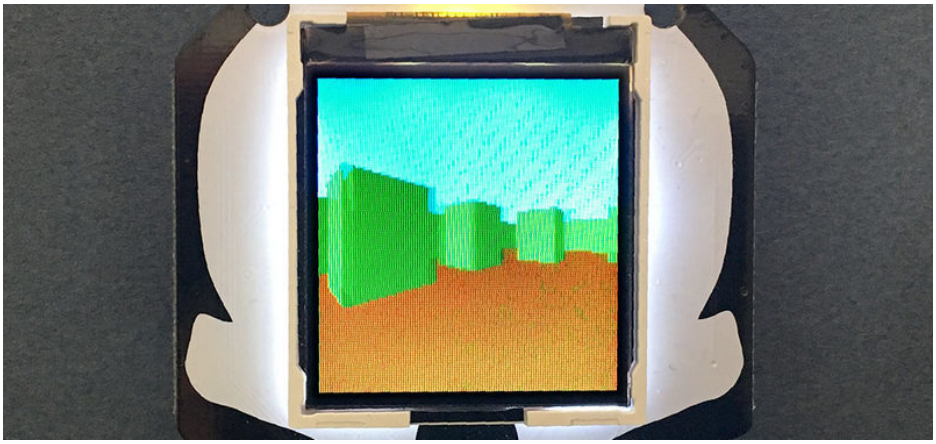
Last updated on 2020-06-29 06:23:58 PM EDT

Overview

In Greek mythology, the ***Labyrinth*** was a cunning maze built by **Daedalus** and **Icarus** to contain the savage half-bull ***Minotaur.*** So complex was the Labyrinth, its own builders could only escape on wings of Daedalus' making...which didn't end well for Icarus. The Minotaur, surviving on human sacrifices, roamed the Labyrinth for years until slain by **Theseus**.

Image: Edward Burne-Jones — Tile Design - Theseus and the Minotaur in the Labyrinth — 1861, public domain.

This project has you losing yourself in the Labyrinth, **smoothly animated in 3D** (30+ frames per second) on the Hallowing display, using tricks that might have impressed Daedalus had he stuck around for microcontrollers and bitmapped graphics.



It's really just a fancy **graphics demo**, something to fidget with and amaze your friends*. Despite the name, there's no actual Minotaur after you. No flag to capture. No exit to flee. The code and techniques used here may prove insightful to programmers learning to maximize graphics performance on microcontrollers.

* If they're not amazed, they're not really your friends.

The Minotaur Maze code uses ***ray casting*** (https://adafru.it/CwW) — a computer graphics technique that's sort of a dollar-store version of ray tracing. Most famously seen in id Software's *Wolfenstein 3D (https://adafru.it/CwX),* ray casting is really a *two-dimensional* algorithm...the most difficult math is required only across a *single scan line,* the result then extended vertically to create the *illusion* of a *three-dimensional environment.*

The ray casting part of the code is adapted from a tutorial by Lode Vandevenne (https://adafru.it/Cx2).

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/CwY

https://adafru.it/CwY

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).

Navigating the Maze

Tilt Hallowing left or right to turn. Tilt forward and back to move in the corresponding direction (the code will do its best to handle different orientations of the board...either sitting flat like on a desk, or held vertically as on a pendant).

Keep in mind that it's simply a **graphics demo** and there's no real gameplay...you can explore the maze but there's no exit, nor any lurking Minotaur.

Build From Source

Building the project from source gives you the opportunity to customize the maze layout and colors.

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/Cxu

https://adafru.it/Cxu

The code as written is fairly specific to the Hallowing M0 board. Advanced programmers might have some success adapting it to other M0 or M4 boards, or different displays.

The **maze layout** is specified in the file MinotaurMaze.ino starting around line 37:

```
// This is the maze map. It's fixed at 32 bits wide, can be any height but
// is 32 in this example. '1' bits indicate solid walls, '0' indicate empty
// space that can be navigated. Perimeter wall bits MUST be set! Keep the
// center area empty since the player is initially placed there.
uint32_t worldMap[] = {
 0b11111111111111111111111111111111,
 0b10000000000000010000000001000001,
 0b10000000000000010111101111011101,
 0b10000000000000000100000100000101,
 0b10000000000000011101110101011101,
 0b10000010100000100010000010000101,
 0b10000010100000011111111010101101,
 0b10000011100000100000000000100001,
 0b10000000000000011101110111011101,
 0b10000000000000010001000001000101,
 0b10000000000000011111111111101111,
 0b10000000000000000000000000000001,
 0b11111011111101110011111011111111,
 0b10000000001010000001000000000001,
 0b10100000101010000001001001001001,
 0b10101010101000000000000000000001,
 0b10101010101000000000000000000001,
 0b10100000101010000001001001001001,
 0b10000000001010000001000000000001,
 0b11111011111101110011111011111111,
 0b10000000000000000000000000000001,
 0b10000010100000000111000010101001,
 0b10001000001000000111000001010101,
 0b10000000000000000111000000000001,
 0b10010000000100000000000011111101,
 0b10000001000000000000000010000101,
 0b10010000000100000011111010100101,
 0b10000000000000000010001010000001,
 0b10001000001000000010101010000101,
 0b10000010100000000010101011111101,
 0b10000000000000000000100000000001,
 0b11111111111111111111111111111111,
};
```

Things to keep in mind:

- "**1**" bits indicate **solid walls**. "**0**" bits are **navigable space**.
- The **outer perimeter bits** must **all be set**.
- The **center area** of the maze should be **kept clear** (using "0" bits) as that's where the player is initially positioned.
- The initial **point of view** is facing **due east** (to the right).

A little further down in the code, starting around line 96, are the **colors** used for different elements:

```
const uint8_t colorSky    = 0x3E,   // Color of sky
              colorGround = 0x82,   // Color of ground
              colorNorth  = 0x04,   // Color of north-facing walls
              colorSouth  = 0x05,   // Color of south-facing walls
              colorEast   = 0x06,   // Color of east-facing walls
              colorWest   = 0x07;   // Color of west-facing walls
```

A RAM-saving trick used in the code limits the available color palette to **256 selections** (click for full-resolution image to better read the numbers):

Beyond this, the remaining graphics are ***not* easily customized**. The next page explains some of the program's internals which experienced programmers might be able to work from, or glean ideas for your own projects.

The full source code:

```
Temporarily unable to load content:
```

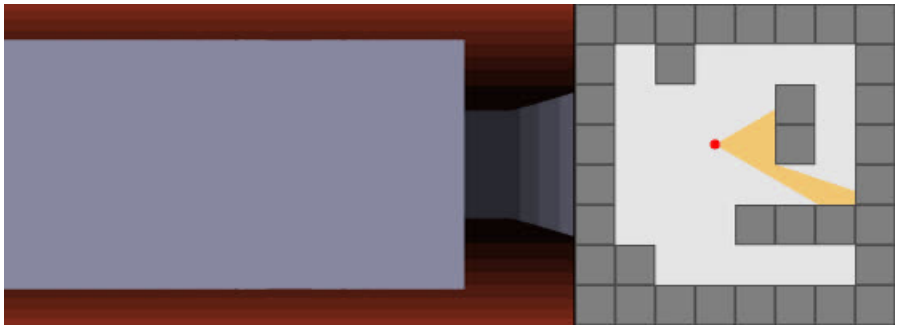
How it Works

Ray Casting

The ray casting part of this project's code is adapted from a tutorial by Lode Vandevenne (https://adafru.it/Cx2) — and they go into *much* greater detail there if you'd like to learn more about it!

Despite appearances — and as explained in the introduction — ray casting is really a *2D* algorithm that just happens to look 3D. From the observer's position in a 2D map, a set of vectors or rays, one per column of the screen, is projected outward into the map. Where those rays first intersect a "solid" square of the map, and the **distance to that intersection**, determines the **height** of the wall for that one column. And which **side** of the square determine's the wall's **color**. Each column is then just drawn as three vertical lines: a section above the wall is the "sky," then the wall itself, then a second below as "ground." If very close to a wall, there might not be any visible sky or ground for that column. This operation is quickly repeated 128 times — once per column of the Hallowing's display.

The combined result looks like a 3D labyrinth. And by changing the observer's position and direction between frames, we've got "3D" animation. *Sorcery!*



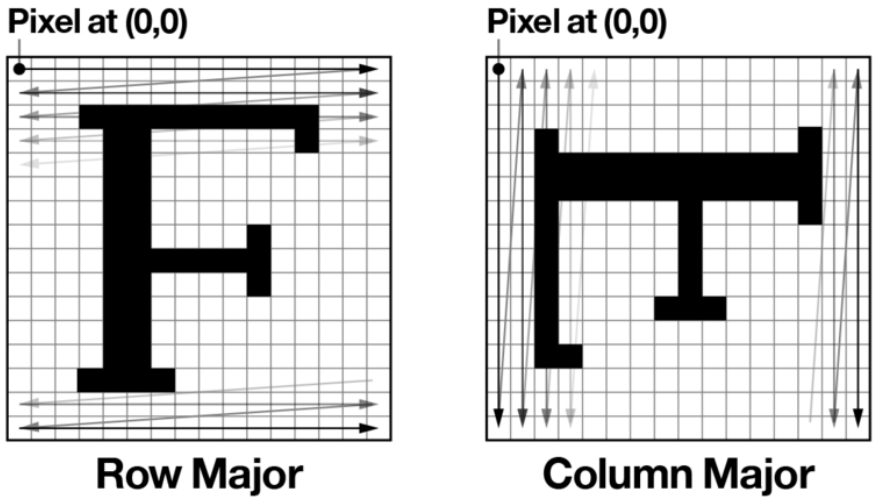
Reconfiguring the Display

Ray casting alone is a *software* technique. Now we pair this up with some *hardware tricks* to make the whole thing work...

Ray casting works column-by-column, but most bitmapped displays store data row-by-row, and compensating for this difference would normally require buffering *two entire screens* worth of data (one is being transferred from RAM to screen while the next frame in RAM is being calculated) — that's 64 kilobytes...but we only have **32 kilobytes** available in the Hallowing's SAMD21 microcontroller.

An interesting property of many TFT displays (including the one used on Hallowing) is that the "mapping" of pixels can be changed. In most situations when redrawing the full screen, as each pixel's color is sent over the SPI bus, the pixel positions increment from left to right across each row, and then rows proceed from top to bottom...it's said to be ***"row major,"*** and most graphic displays work this way...it's a throwback to how CRT monitors worked.

With a small change, we can configure the display to work in ***"column major"*** order — successive pixels increment top to bottom along each column, with columns proceeding left to right. It's a peculiar layout, combining rotation and mirroring operations, but it's *perfect* for this project's needs. Each column of graphics can be issued to the display immediately after being processed. No need to buffer a whole screen's worth, let alone two!



Here's the code that reconfigures the display. This only needs to be done once, after the display is initialized in the setup() function.

While the ST7735 library does use a similar technique to handle screen rotation in hardware (via the setRotation() function), this combination of rotation and mirroring is peculiar enough that we must work around the library and talk directly to the TFT driver's device registers...

```
  digitalWrite(TFT_CS, LOW);
  digitalWrite(TFT_DC, LOW);
#ifdef ST77XX_MADCTL
  SPI.transfer(ST77XX_MADCTL); // Current TFT lib
#else
  SPI.transfer(ST7735_MADCTL); // Older TFT lib
#endif
  digitalWrite(TFT_DC, HIGH);
  SPI.transfer(0x28);
  digitalWrite(TFT_CS, HIGH);
```

DMA Tricks

While the ray casting explanation above talks about "drawing lines," if you dig through the code you won't find even a *single call* to drawLine(). Aside from initializing the display hardware, we have to go around the Adafruit_ST7735 library and do things a *lot* faster...

Direct memory access (**DMA**) is a feature of the SAMD21 microcontroller that facilitates an explicit form of multitasking...for example, the chip can transfer a buffer full of data from RAM to the SPI bus (to the TFT display) without requiring the CPU's intervention for every byte...it can go along calculating other things while the transfer proceeds in the background.

The process is explained a bit in the **Hallowing Spirit Board tutorial (https://adafru.it/Cxq)**, which also achieves smooth full-screen animation. Instead of working one drawing operation at a time, we push data out the SPI bus *as fast as possible* in *one long stream.* That code uses two buffers in RAM, each enough for **one row's worth of pixels**...one line is being calculated while the prior line is concurrently issued over SPI using DMA, and we switch between them at the start of each new line.

The Minotaur Maze code takes this idea to the extreme. It doesn't buffer even a single scanline in RAM. In fact, all of

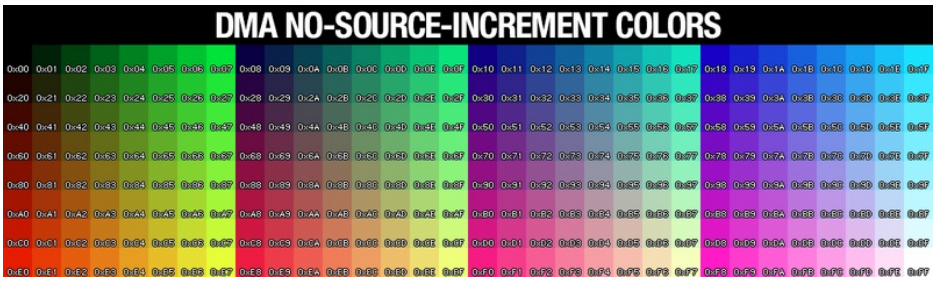
the “graphics data” as it were — the colors of the walls, sky and floor — occupy **six bytes of flash memory**!

DMA usually operates on contiguous blocks of memory. Copy this block of data from here to here, transfer this block from RAM to SPI, and so forth. But there’s a configuration bit in the *DMA descriptor* (the structure that defines a DMA transfer operation) indicating whether the source data pointer should be incremented after each byte. For example, if reading data *from* SPI, you want the source pointer (aimed at the SPI data register) to stay put. By using this when writing *to* SPI, the same byte will be transferred repeatedly…and we can use this to fill spans of pixels…our vertical columns of wall and so forth.

```
const uint8_t color = 0x42; // One color byte is stored in RAM or flash

descriptor.SRCADDR.reg       = (uint32_t)&color;
descriptor.DSTADDR.reg       = (uint32_t)SPI.DATA.reg;
descriptor.BTCTRL.bit.SRCINC = 0;
descriptor.BTCTRL.bit.DSTINC = 0;
descriptor.BTCNT.reg         = num_pixels * 2;
...
```

The downside to this approach is that we don’t have full use of the 16-bit color space that the TFT display can provide. Each pixel expects 16 bits of color data, but because we’re **issuing the same byte twice for each pixel**…the high and low bytes must be the same…that limits our options to a fixed palette of **256 entries**:



It’s a decent spread, but if you’re looking for *just the right nuanced color* you may not find it. Also, interactions between the color byte pairs mean you can’t get a 100% pure red, green or blue…green will always have a little blue mixed in, red will always have a little green mixed in, and so forth. Black (0x00 hexadecimal) and white (0xFF) are the only predictable colors.

So are textured walls possible?

My hunch is that the SAMD21 is fast enough to do this, at a slightly reduced (but still interactive) frame rate. Maybe I’ll revisit this code at some point. It would have to use some RAM buffering and we wouldn’t get use of the no-source-increment fill trick described above, which is really the thing I wanted to cover here.

Hallowing’s SAMD21 processor runs at 48 MHz, the SPI bus at 12 MHz, and each pixel requires 16 bits of data. Between the latter two, we can estimate a theoretical peak transfer rate of 750,000 pixels per second. And dividing the former, that allows about 64 processor clock cycles per pixel. Can we stretch a bitmap that quickly? Probably, or at least something in the ballpark. Some overhead is still required for the ray casting calculations and setting up DMA transfers…but enough left over that I suspect there’s still a lot of graphics surprises remaining to be squeezed out of this little board!