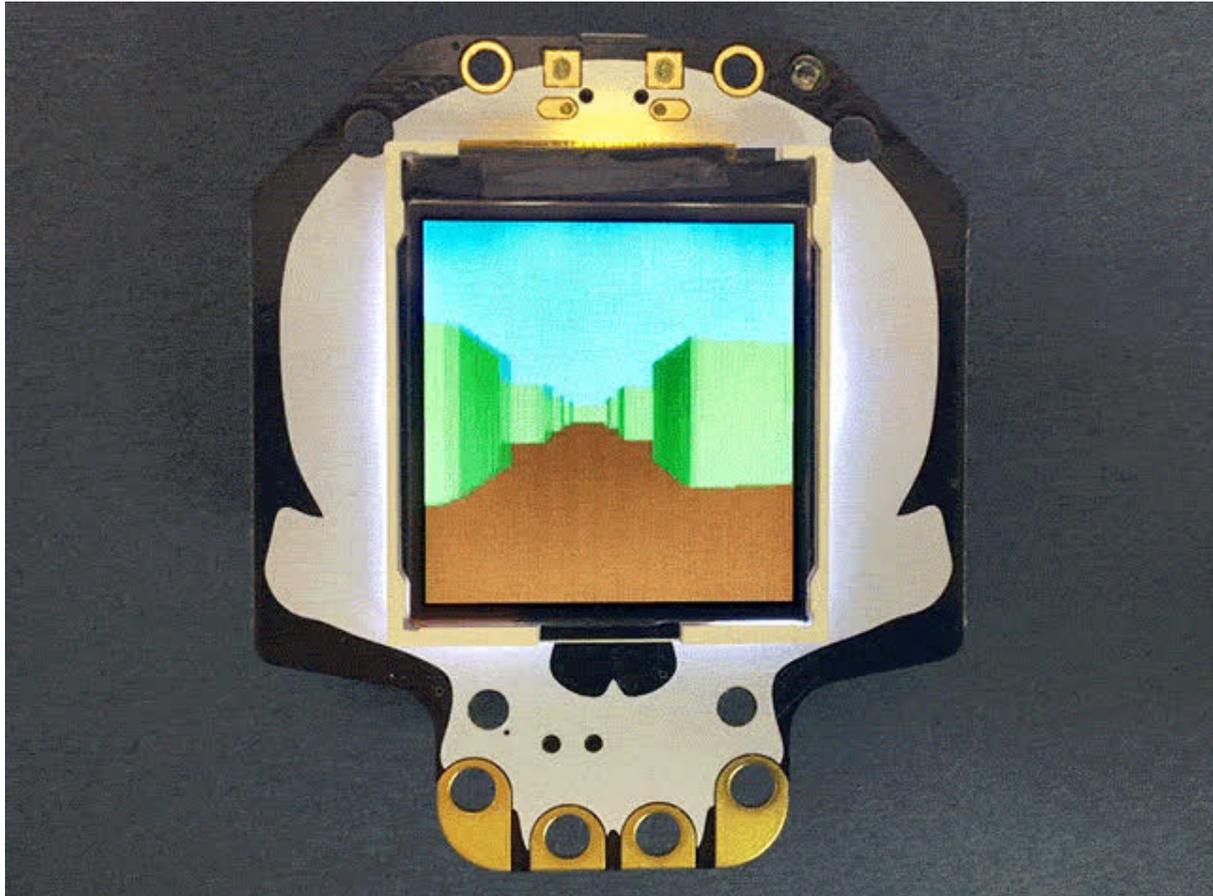




Hallowing Minotaur Maze

Created by Phillip Burgess



<https://learn.adafruit.com/hallowing-minotaur-maze>

Last updated on 2024-06-03 02:29:36 PM EDT

Table of Contents

Overview	3
Software	4
• Easy Way	
• Navigating the Maze	
• Build From Source	
How it Works	13
• Ray Casting	
• Reconfiguring the Display	
• DMA Tricks	

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.

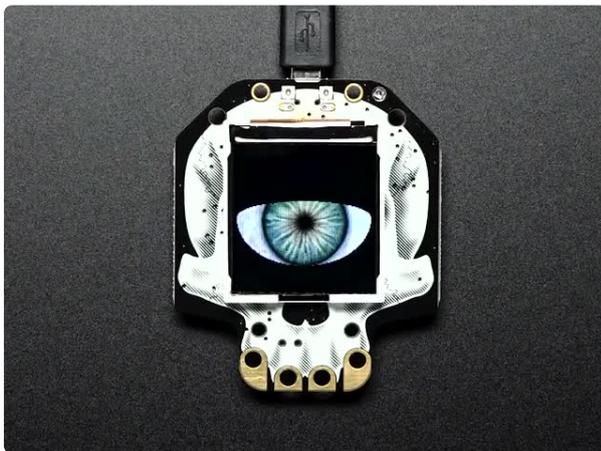
The Minotaur Maze code uses **ray casting** (<https://adafru.it/CwW>) — a computer graphics technique that's sort of a dollar-store version of ray tracing. Most famously seen in id Software's **Wolfenstein 3D** (<https://adafru.it/CwX>), ray casting is really a two-

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 \(https://adafru.it/Cx2\)](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.)

This project only works with the original HalloWing M0. It will not work on the HalloWing M4.



[Adafruit HalloWing M0 Express](https://www.adafruit.com/product/3900)

This is HalloWing..this is HalloWing...
HalloWing! HalloWing! Are you the kind of
person who doesn't...

<https://www.adafruit.com/product/3900>

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.

MINOTAUR_MAZE.UF2

<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 \(https://adafru.it/CmJ\)](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 \(https://adafru.it/Cpl\)](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

Minotaur Maze source code on Github

<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,
  0b10000000000000001000001000000101,
  0b100000000000000111011101010111101,
  0b10000010100000100010000010000101,
  0b10000010100000111111111010101101,
  0b10000011100000100000000000100001,
  0b100000000000000111011101110111101,
  0b100000000000000100010000010001001,
  0b10000000000000011111111111101111,
  0b10000000000000000000000000000001,
  0b11111011111011100111111011111111,
  0b10000000001010000001000000000001,
  0b10100000101010000001001001001001,
  0b10101010101000000000000000000001,
  0b10101010100000000000000000000001,
  0b10100000101010000001001001001001,
  0b10000000001010000001000000000001,
  0b11111011111011100111111011111111,
  0b10000000000000000000000000000001,
  0b10000010100000000111000010101001,
  0b10001000001000000111000001010101,
  0b10000000000000000111000000000001,
  0b1001000000010000000000011111101,
  0b10000001000000000000000010000101,
  0b10010000000100000011111010100101,
  0b10000000000000000001000101000001,
  0b10001000001000000010101010000101,
  0b1000001010000000001010101111101,
  0b10000000000000000000100000000001,
  0b11111111111111111111111111111111,
};
```

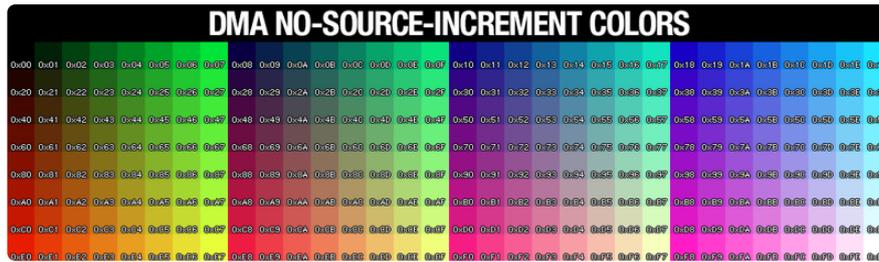
Things to keep in mind:

- “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:

```
// SPDX-FileCopyrightText: 2018 Phillip Burgess for Adafruit Industries
//
// SPDX-License-Identifier: MIT

// "Minotaur Maze" plaything for Adafruit Hallowing. Uses ray casting,
// DMA and related shenanigans to smoothly move about a 3D maze.
// Tilt Hallowing to turn right/left and move forward/back.

// Ray casting code adapted from tutorial by Lode Vandevenne:
// https://lodev.org/cgtutor/raycasting.html

#include <Adafruit_LIS3DH.h> // Accelerometer library
#include <Adafruit_GFX.h>   // Core graphics library
#include <Adafruit_ST7735.h> // Display-specific graphics library
#include <Adafruit_ZeroDMA.h> // Direct memory access library

#ifdef ARDUINO_SAMD_CIRCUITPLAYGROUND_EXPRESS
  #define TFT_RST      -1
  #define TFT_DC       A6
  #define TFT_CS       A7
  #define TFT_BACKLIGHT A3 // Display backlight pin
  #define TFT_SPI      SPI1
  #define TFT_PERIPH   PERIPH_SPI1
  Adafruit_LIS3DH accel(&Wire1);
#else
  #define TFT_RST      37 // TFT reset pin
  #define TFT_DC       38 // TFT display/command mode pin
  #define TFT_CS       39 // TFT chip select pin
  #define TFT_BACKLIGHT 7 // TFT backlight LED pin
  #define TFT_SPI      SPI
  #define TFT_PERIPH   PERIPH_SPI
  Adafruit_LIS3DH accel;
#endif

// Declarations for some Hallowing hardware -- display, accelerometer, SPI
Adafruit_ST7735 tft(&TFT_SPI, TFT_CS, TFT_DC, TFT_RST);
SPISettings settings(12000000, MSBFIRST, SPI_MODE0);

// Declarations related to DMA (direct memory access), which lets us walk
// and chew gum at the same time. This is VERY specific to SAMD chips and
// means this is not trivially ported to other devices.
Adafruit_ZeroDMA dma;
DmacDescriptor *dptr; // Initial allocated DMA descriptor
```

```

DmacDescriptor    desc[2][3]  __attribute__((aligned(16)));
uint8_t           dList = 0; // Active DMA descriptor list index (0-1)

// DMA transfer-in-progress indicator and callback
static volatile bool dma_busy = false;
static void dma_callback(Adafruit_ZeroDMA *dma) {
    dma_busy = false;
}

// 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,
    0b100000000000000101111011111011101,
    0b10000000000000001000001000000101,
    0b100000000000000111011101010111101,
    0b10000010100000100010000010000101,
    0b10000010100000111111111010101101,
    0b10000011100000100000000000100001,
    0b100000000000000111011101110111101,
    0b100000000000000100010000010001001,
    0b10000000000000011111111111101111,
    0b10000000000000000000000000000001,
    0b11111011111011100111111011111111,
    0b10000000001010000001000000000001,
    0b10100000101010000001001001001001,
    0b10101010101000000000000000000001,
    0b10101010101000000000000000000001,
    0b10100000101010000001001001001001,
    0b10000000001010000001000000000001,
    0b11111011111011100111111011111111,
    0b10000000000000000000000000000001,
    0b10000010100000000111000010101001,
    0b1000100000100000011100001010101,
    0b10000000000000000011100000000001,
    0b10010000000100000000000011111101,
    0b10000001000000000000000010000101,
    0b10010000000100000011111010100101,
    0b10000000000000000001000101000001,
    0b10001000001000000010101010000101,
    0b1000001010000000001010101111101,
    0b1000000000000000000010000000001,
    0b11111111111111111111111111111111,
};
#define MAPHEIGHT (sizeof worldMap / sizeof worldMap[0])

// This macro tests whether bit at (X,Y) in the map is set.
#define isBitSet(X,Y) (worldMap[MAPHEIGHT-1-(Y)] & (0x80000000>>(X)))
// (X,Y) are in Cartesian coordinates with (0,0) at bottom-left (hence the
// MAPHEIGHT-1-Y inversion above) -- all the navigation and ray-casting math
// is done in Cartesian space, consistent with the trigonometric functions,
// whereas bitmap is represented top-to-bottom.

// DMA shenanigans are used for the solid color fills (sky, walls and
// floor). Typically one would use the DMA "source address increment" to
// copy graphics data from RAM or flash to SPI (to the screen). But a trick
// we can use for certain fills requires only a single byte of storage for
// each color. DMA source increment is turned OFF -- the same byte is issued
// over and over to fill a given span. Downside is a limited palette
// consisting of 256 colors with the high and low bytes of a 16-bit pixel
// value being the same. With the TFT's 5-6-5 bit color packing, the
// resulting selections are a bit weird (there's no 100% pure red, green or
// blue, only combinations) but usable. e.g. an 8-bit value 0x82 expands to
// a 16-bit pixel value of 0x8282 = 0b10000 010100 00010 = 16/31 (~52%) red,
// 20/63 (~32%) green, 2/31 (6%) blue.
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

#define FOV (90.0 * (M_PI / 180.0)) // Field of view

float    posX      = 16.0,      // Observer position,
        posY      = MAPHEIGHT / 2.0, // begin at center of map
        heading    = 0.0;      // Initial heading = east

uint32_t startTime, frames = 0; // For frames-per-second calculation

// SETUP -- RUNS ONCE AT PROGRAM START -----

void setup(void) {
    Serial.begin(115200);

    Serial.println("Init accelerometer");
    // Initialize accelerometer, set to 2G range
    if(accel.begin(0x18) || accel.begin(0x19)) {
        accel.setRange(LIS3DH_RANGE_2_G);
    }

    Serial.println("Init display");
    // Initialize and clear screen
    tft.initR(INITR_144GREENTAB);
    tft.setRotation(1);
    tft.fillScreen(0);

    // More shenanigans: the display mapping is reconfigured so pixels are
    // issued in COLUMN-MAJOR sequence (i.e. vertical lines), left-to-right,
    // with pixel (0,0) at top left. The ray casting algorithm determines the
    // wall height at each column...drawing is then just a matter of blasting
    // a column's worth of pixels.
    digitalWrite(TFT_CS, LOW);
    digitalWrite(TFT_DC, LOW);
#ifdef ST77XX_MADCTL
    TFT_SPI.transfer(ST77XX_MADCTL); // Current TFT lib
#else
    TFT_SPI.transfer(ST7735_MADCTL); // Older TFT lib
#endif
    digitalWrite(TFT_DC, HIGH);
    TFT_SPI.transfer(0x28);
    digitalWrite(TFT_CS, HIGH);

    pinMode(TFT_BACKLIGHT, OUTPUT);
    digitalWrite(TFT_BACKLIGHT, HIGH); // Main screen turn on
    Serial.println("Init backlight");

    // Set up SPI DMA. While the Hallowing has a known SPI peripheral and this
    // could be much simpler, the extra code here will help if adapting this
    // sketch to other SAMD boards (Feather M0, M4, etc.)
    int          dmac_id;
    volatile uint32_t *data_reg;
    dma.allocate();
    if(&TFT_PERIPH == &sercom0) {
        dma.setTrigger(SERCOM0_DMAC_ID_TX);
        data_reg = &SERCOM0->SPI.DATA.reg;
    }
#ifdef SERCOM1
    } else if(&TFT_PERIPH == &sercom1) {
        dma.setTrigger(SERCOM1_DMAC_ID_TX);
        data_reg = &SERCOM1->SPI.DATA.reg;
    }
#endif
#ifdef SERCOM2
    } else if(&TFT_PERIPH == &sercom2) {
        dma.setTrigger(SERCOM2_DMAC_ID_TX);
        data_reg = &SERCOM2->SPI.DATA.reg;
    }
#endif

```

```

#endif
#if defined SERCOM3
  } else if(&TFT_PERIPH == &sercom3) {
    dma.setTrigger(SERCOM3_DMACH_ID_TX);
    data_reg = &SERCOM3->SPI.DATA.reg;
  }
#endif
#if defined SERCOM4
  } else if(&TFT_PERIPH == &sercom4) {
    dma.setTrigger(SERCOM4_DMACH_ID_TX);
    data_reg = &SERCOM4->SPI.DATA.reg;
  }
#endif
#if defined SERCOM5
  } else if(&TFT_PERIPH == &sercom5) {
    dma.setTrigger(SERCOM5_DMACH_ID_TX);
    data_reg = &SERCOM5->SPI.DATA.reg;
  }
#endif
}
dma.setAction(DMA_TRIGGER_ACTON_BEAT);
dma.setCallback(dma_callback);

// Initialize DMA descriptor lists. There are TWO lists, used for
// alternating even/odd scanlines (columns in this case)...one list is
// calculated and filled while the other is being transferred out SPI.
// Each list contains three elements (though not all three are used every
// time), corresponding to the sky, wall and ground pixels for a column.
for(uint8_t s=0; s<2; s++) { // Even/odd scanlines
  for(uint8_t d=0; d<3; d++) { // 3 descriptors per line
    // No need to set SRCADDR, BTCNT or DESCADDR -- done later
    desc[s][d].BTCTRL.bit.VALID = true;
    desc[s][d].BTCTRL.bit.EVOSEL = 0x3;
    desc[s][d].BTCTRL.bit.BLOCKACT = DMA_BLOCK_ACTION_NOACT;
    desc[s][d].BTCTRL.bit.BEATSIZE = DMA_BEAT_SIZE_BYTE;
    desc[s][d].BTCTRL.bit.SRCINC = 0;
    desc[s][d].BTCTRL.bit.DSTINC = 0;
    desc[s][d].BTCTRL.bit.STEPSEL = DMA_STEPSEL_SRC;
    desc[s][d].BTCTRL.bit.STEPSIZE = DMA_ADDRESS_INCREMENT_STEP_SIZE_1;
    desc[s][d].DSTADDR.reg = (uint32_t)data_reg;
  }
}

// The DMA library MUST allocate at least one valid descriptor, so that's
// done here. It's not used in the conventional sense though, just before
// a transfer we copy the first scanline descriptor to this spot.
dptr = dma.addDescriptor(NULL, NULL, 42, DMA_BEAT_SIZE_BYTE, false, false);

startTime = millis(); // Starting time for frame-per-second calculation
}

// LOOP -- REPEATS INDEFINITELY -----
void loop() {

  // Update heading and position from accelerometer...
  uint8_t mapX = (uint8_t)posX, // Current square of map
  mapY = (uint8_t)posY; // (before changing pos.)
  accel.read(); // Read accelerometer
#ifdef ARDUINO_SAMD_CIRCUITPLAYGROUND_EXPRESS
  heading += (float)accel.x / -20000.0; // Update direction
  float v = (abs(accel.y) < abs(accel.z)) ? // If board held flat(ish)
  (float)accel.y / 20000.0 : // Use accel Y for velocity
  (float)accel.z / -20000.0; // else accel Z is velocity
#else
  heading += (float)accel.y / -20000.0; // Update direction
  float v = (abs(accel.x) < abs(accel.z)) ? // If board held flat(ish)
  (float)accel.x / 20000.0 : // Use accel X for velocity
  (float)accel.z / -20000.0; // else accel Z is velocity
#endif
  if(v > 0.19) v = 0.19; // Keep speed under 0.2
  else if(v < -0.19) v = -0.19;
}

```

```

float   vx   = cos(heading) * v,           // Direction vector X, Y
        vy   = sin(heading) * v,
        newX = posX + vx,                 // New position
        newY = posY + vy;

// Prevent going through solid walls (or getting too close to them)
if(vx > 0) {
    if(isBitSet((int)(newX + 0.2), (int)newY)) newX = mapX + 0.8;
} else {
    if(isBitSet((int)(newX - 0.2), (int)newY)) newX = mapX + 0.2;
}
if(vy > 0) {
    if(isBitSet((int)newX, (int)(newY + 0.2))) newY = mapY + 0.8;
} else {
    if(isBitSet((int)newX, (int)(newY - 0.2))) newY = mapY + 0.2;
}

posX = newX;
posY = newY;

TFT_SPI.beginTransaction(settings); // SPI init
digitalWrite(TFT_CS, LOW); // Chip select
tft.setAddrWindow(0, 0, 128, 128); // Set address window to full screen
digitalWrite(TFT_CS, LOW); // Re-select after addr function
digitalWrite(TFT_DC, HIGH); // Data mode...

// Ray casting code is much abbreviated here.
// See Lode Vandevonne's original tutorial for an in-depth explanation:
// https://lodev.org/cgtutor/raycasting.html

int8_t  stepX, stepY; // X/Y direction steps (+1 or -1)
uint8_t skyPixels, floorPixels, // # of pixels in sky, floor
        side, // North/south or east/west wall hit?
        i; // Index in DMA descriptor list
uint16_t wallPixels; // # of wall pixels
float   frac, rayDirX, rayDirY,
        sideDistX, sideDistY, // Ray length, current to next X/Y side
        deltaDistX, deltaDistY, // X-to-X, Y-to-Y ray lengths
        perpWallDist, // Distance to wall
        x1 = cos(heading + FOV / 2.0), // Image plane left edge
        y1 = sin(heading + FOV / 2.0),
        x2 = cos(heading - FOV / 2.0), // Image plane right edge
        y2 = sin(heading - FOV / 2.0),
        dx = x2 - x1, dy = y2 - y1;

for(uint8_t col = 0; col < 128; col++) { // For each column...
    frac = ((float)col + 0.5) / 128.0; // 0 to 1 left to right
    rayDirX = x1 + dx * frac;
    rayDirY = y1 + dy * frac;
    mapX = (uint8_t)posX;
    mapY = (uint8_t)posY;
    deltaDistX = (rayDirX != 0.0) ? fabs(1 / rayDirX) : 0.0;
    deltaDistY = (rayDirY != 0.0) ? fabs(1 / rayDirY) : 0.0;

    // Calculate X/Y steps and initial sideDist
    if(rayDirX < 0) {
        stepX = -1;
        sideDistX = (posX - mapX) * deltaDistX;
    } else {
        stepX = 1;
        sideDistX = (mapX + 1.0 - posX) * deltaDistX;
    } if (rayDirY < 0) {
        stepY = -1;
        sideDistY = (posY - mapY) * deltaDistY;
    } else {
        stepY = 1;
        sideDistY = (mapY + 1.0 - posY) * deltaDistY;
    }
}

```

```

do { // Bresenham DDA line algorithm...walk map squares...
  if(sideDistX < sideDistY) {
    sideDistX += deltaDistX;
    mapX      += stepX;
    side      = 0; // East/west
  } else {
    sideDistY += deltaDistY;
    mapY      += stepY;
    side      = 1; // North/south
  }
} while(!isBitSet(mapX, mapY)); // Continue until wall hit

// Calc distance projected on camera direction
perpWallDist = side ? ((mapY - posY + (1 - stepY) / 2) / rayDirY) :
                  ((mapX - posX + (1 - stepX) / 2) / rayDirX);

wallPixels = (int)(128.0 / perpWallDist); // Colum height in pixels
if(wallPixels >= 128) { // >= screen height?
  wallPixels = 128; // Clip to screen height
  skyPixels = floorPixels = 0; // No sky or ground
} else {
  skyPixels = (128 - wallPixels) / 2; // 1/2 of non-wall is sky
  floorPixels = 128 - wallPixels - skyPixels; // Any remainder is floor
}

// Build DMA descriptor list with up to 3 elements...
i = 0;
if(skyPixels) { // Any sky pixels in this column?
  desc[dList][i].SRCADDR.reg = (uint32_t)&colorSky;
  desc[dList][i].BTCNT.reg = skyPixels * 2;
  desc[dList][i].DESCADDR.reg = (uint32_t)&desc[dList][i + 1];
  i++;
}
if(wallPixels) { // Any wall pixels?
  // North/south or east/west facing?
  desc[dList][i].SRCADDR.reg = (uint32_t)(side ?
    ((stepY > 0) ? &colorSouth : &colorNorth) :
    ((stepX > 0) ? &colorWest : &colorEast));
  desc[dList][i].BTCNT.reg = wallPixels * 2;
  desc[dList][i].DESCADDR.reg = (uint32_t)&desc[dList][i + 1];
  i++;
}
if(floorPixels) { // Any floor pixels?
  desc[dList][i].SRCADDR.reg = (uint32_t)&colorGround;
  desc[dList][i].BTCNT.reg = floorPixels * 2;
  desc[dList][i].DESCADDR.reg = (uint32_t)&desc[dList][i + 1];
  i++;
}
desc[dList][i - 1].DESCADDR.reg = 0; // End descriptor list

while(dma_busy); // Wait for prior DMA transfer to finish
// Copy scanline's first descriptor to the DMA lib's descriptor table
memcpy(dpPtr, &desc[dList][0], sizeof(DmacDescriptor));
dma_busy = true; // Mark as busy (DMA callback clears this)
dma.startJob(); // Start new DMA transfer
dList = 1 - dList; // Swap active DMA descriptor list index
}
while(dma_busy); // Wait for last DMA transfer to complete
digitalWrite(TFT_CS, HIGH); // Deselect
TFT_SPI.endTransaction(); // SPI done

if(!(++frames & 255)) { // Every 256th frame, show frame rate
  uint32_t elapsed = (millis() - startTime) / 1000;
  if(elapsed) Serial.println(frames / elapsed);
}
}

```

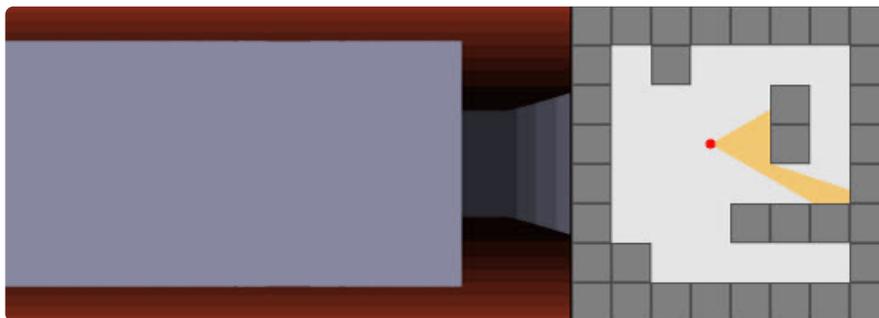
How it Works

Ray Casting

The ray casting part of this project’s code is adapted from [a tutorial by Lode Vandevenne \(https://adafru.it/Cx2\)](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 Halloween’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

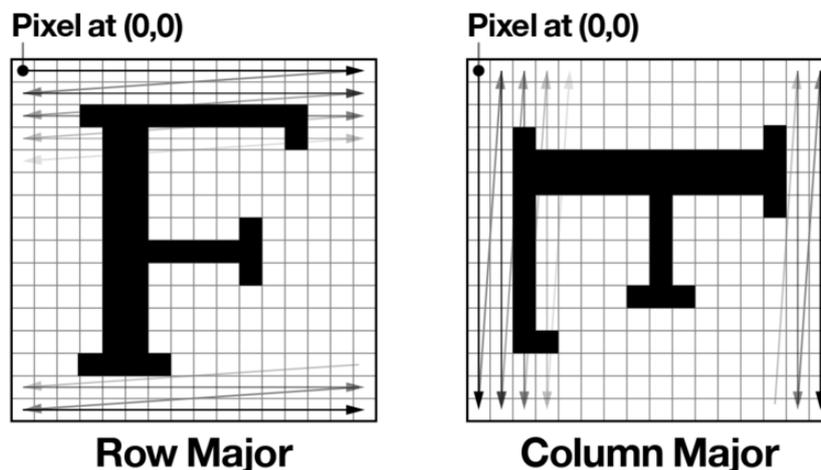
Ray casting alone is a software technique. Now we pair this up with some hardware tricks to make the whole thing work...

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 \(https://adafru.it/Cxq\)](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!**

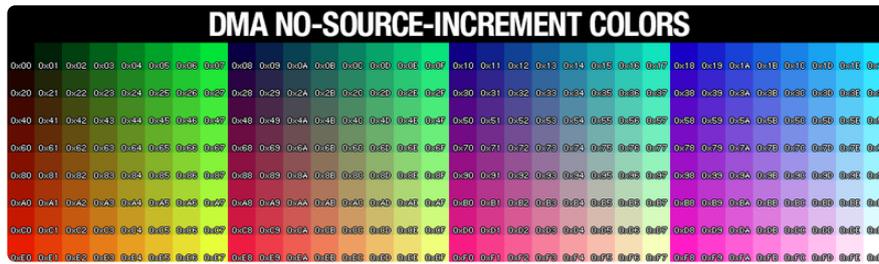
DMA usually operates on contiguous blocks of memory. Copy this block of data from here to here, transfer this block from RAM to SPI, and so forth. But there’s a configuration bit in the DMA descriptor (the structure that defines a DMA transfer operation) indicating whether the source data pointer should 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!