How the code works

Basically, we cast a ray for every single pixel column (`update_rays()`), each of which loops through every single cell, and every single wall in that cell, and checks for intersections. Despite having no concurrency whatsoever, the vast majority of the big expensive code was designed to be easily parallelized, just pretty much out of habit.
Anyway, the rays:

The rays

Each ray does some math to figure out the intersection point between it and a wall section of a cell. It runs tons of checks to reduce unnecessary computations:

- If the wall is open and not fake, return
- If we've already hit something and the wall is fake, return
- If the lines are parallel, return
- If the distance is greater than anything we've already hit, return
- If the intersection point would be outside the extents of the wall, return

Once all of those have passed, we know we have a new closest hit. We record the t value along the wall of the intersection, the distance, a pointer to the cell itself, and the index of which wall in the cell it was. Brief aside on how the ray-wall intersections are actually calculated, *in theory*:

$$\vec{\Delta w} = \vec{W_b} - \vec{W_a}$$

$$d = \vec{\Delta w} \times \vec{R_{dir}}$$

$$\vec{\Delta p} = \vec{W_a} - \vec{R_{pos}}$$

$$u = \frac{\vec{\Delta w} \times \vec{\Delta p}}{d}$$

$$t = \frac{\vec{R_{dir}} \times \vec{\Delta p}}{d}$$

Where:

- $\vec{W_b}, \vec{W_a}$ Are the coordinates of the endpoints of the wall
- $\vec{R_{dir}}$ is the direction of the ray being cast
- $\vec{R_{pos}}$ is the starting point of the ray (aka the position of the camera)
- u is the actual distance of the starting point to the intersection
- t is the proportion along the length of the wall that the intersection happened

We reject the intersection if $u \leq 0$, $t \leq 0$, or $t \geq 1$. And of the remaining intersections, we pick the one with the lowest value of u.
Of course, all of these computations are being done with floating point numbers, ~~specifically to upset you~~ to ensure that there aren't any strange artifacts in the rendering. After we have cast all the rays, we next have to calculate the height and shade of the corresponding column of pixels.

The Columns

To update each column (`update_columns()`) we check first if the ray hit something, and that it was real. If not, we basically discard it and set the height to zero. But otherwise, we use this absolutely disgusting function:

$$d = \frac{H}{4} - 200 * \tan^{-1}\left(\frac{\sqrt{1 + \left(x - \frac{W}{2}\right)^2 * \frac{1}{a^2}}}{ray.dist}\right)$$

Where:

- d is the "extent" of the column, actually measured as the distance from the edge of the screen that is black
- H is the height of the display

- W is the width of the display
- x is the column index (basically the x position)
- a is some mystery constant from my original code made years ago, actually called `ang_height` in the code. I could not tell you what it represents, but it's calculated like this: $\frac{W}{2} * \tan\left(90 - \frac{FOV}{2}\right)$
- $ray.dist$ is simply the distance of the closest intersection that we found in the last step

The shade is then linearly scaled based on d, with a min and max of `0.05` and `1`, respectively. If the textures are off, we precompute the color of the entire column as a single color, otherwise we store the t, `shade`, and texture ID. There's less fun math in the rest of the sections, so they'll be more brief. After we've calculated the columns, next is to draw them!

The Render

This is done in the `draw_render()` portion, which is kind of a confusing name. Basically, I consider the bottom half to be the map, which makes sense, but I call the upper half the "render", since it was literally rendered. In any case, we loop row by row, then column by column. We check if the current row is within the extents of the wall, and if it is we draw either the stored color or we sample the texel[1] and scale it by the shade. If we do the texture, it uses a simpler version of the `rgb_mult` function which uses integer division instead of floating point multiplication. Turns out, 3 floating point operations, and 6 floating point conversions, *on every pixel*, wasn't very economical.
If it is outside of the extent, we check if it is inside the extent of the wall on the previous frame, and if so then we draw it black. A `memset` to completely erase the top half of the display might have worked too, but this is more robust. Going column by column likely would have been better for memory locality in terms of accessing the column structs, at the cost of worse memory locality in the pixel buffer array (unless of course I had switched to column major indexing). Anyway, I couldn't think of a good segue for this one but

The Textures

Each texture is a 16x16 image with 4 colors, each of which are defined in a palette array in the struct. Each color in the palette can be any of the 65536 colors the device supports, being stored as `uint16_t`. The textures themselves are stored as an array of 16 `uint32_t` items, each of which holds 16 individual texels[1-1], consisting of 2 bits representing the ID in the palette that it should map to. That each `uint32_t` stores exactly one row is pure serendipity. The macros are all set up in such a way that I could have allocated any number of bits for each texel as I felt like, but 2 seemed like enough. Plus, it was an interesting design constraint when it came to actually creating the textures themselves. Of course, because it's such a user unfriendly format, I had to write a quick script to turn the raw data of the `.bmp` files of the images into actual binary data that I could paste into my code.
I also created a `get_tex` function which takes in a texture ID, an x and y, and an extent, and returns the appropriate texel for rendering.

The Cells

Again, no segue for this one. There are a few interesting things to talk about with the cells, but the one I'm most proud of is probably the way the walls are stored. It's simply a 16 bit uint that functions more like the following:

```
struct _WallSection {
        uint2_t tex_id;
        bool is_fake;
        bool is_there;
};
struct _WallSection walls[4];
```

About halfway through development I decided it was high time to add some inline functions to access these parts in a more streamlined manner, instead of manually bitmasking over and over. This made all my code cleaner and clearer, which was a massive help. Next thing to discuss is what it means for a wall to be "real."

Fake walls

Whenever a cell is generated, it tries to make sure at least one wall is open (we'll get back to this later). However, if it then also generated the neighboring cell, that one would have to also have an open wall, which would mean another cell needs to be generated, and so on and so on. Instead, the open wall is marked "fake". What this means is that if we see it (ie if a ray detects it as the closest intersection), only then do we actually generate the cell behind it. This means that the raycasting is far more efficient as only the cells that have been recently seen are actually checked against for collisions, and it means that the maze can regenerate differently each time. This is because when a cell hasn't been hit by any rays, we just delete it. The walls of neighboring cells that were open are now set to be fake, so when we look at where it was, it gets generated anew!

Cell Generation

This is one of the most convoluted parts of the entire program. Essentially, when a ray detects it has hit a fake wall, it records the X and Y coordinates of the cell to be generated, as well as the direction it came from (ie the index of the wall that was fake). These are used in the `generate_cell()` function, which goes through a myriad of double checks to make sure we don't end up at an invalid state of the maze. Checks such as

- Does a cell somehow already exist at the place we are supposed to make a new one?
- Do we have an open slot available to actually create a new active cell?

Are performed before we even do anything. After that, we try to find which walls are candidates to make open. We check the neighboring cells[2] to see if they have walls that are there and real on the line between them and the cell we are trying to create. We manually set the wall corresponding to the direction that the ray came from to be open and not fake. Then, using an algorithm I probably shouldn't be as proud of as I am, we pick a random wall from the remaining candidates. We set it to be gone and fake, both for this cell and the neighboring one.

The Map

On the lower half of the screen, we have the "mini"-map. For each currently active cell, we loop through all the walls, and then draw a line in the pixel buffer where it would be on the map, if it's real and there. We also use the solid colors of the walls, while also still using the wall's texture ID. This means that if you have the textures off, the colors you see in the render will line up with the colors you see on the minimap.
Critically, unactive cells are not cleared from the pixel buffer. The information you collect about the maze stays around. This is very much intentional. It draws attention to the fact that the maze is in fact changing. You can see the info on the map get overwritten in real time as you look away and look back.

The Joystick

We sample both the x and y coordinates of the joystick, at a rate independent of frame rate. This allows us to tune the movement to a point that feels good without having to redo it if we change the update rate of the rest of the tasks.
To get the movement, we sample the stick, turn it into floats, and rescale it to be in the range $(-1, 1)$. We also have a deadzone in the middle of $(-0.125, 0.125)$, to avoid any inaccuracies in the resting point of the joystick.
We simply add the x component of the vector to the facing angle of the camera, allowing you to turn.
We then create a vector based on the current facing direction of the camera, scale it by the y axis of the stick reading, and move the camera by that amount.
This type of control scheme is usually referred to as "tank controls" in the gaming community, due to the similarities to having to turn a tank in place by running the treads in opposite directions, as well as only being able to move in a straight line forwards or backwards.

The Display

The display driver I made uses a 128x128 buffer of 16 bit uints. I created a function to set a pixel, which simply sets the appropriate value in the pixel buffer array. I also created a `display_update_section` function, which takes a custom struct called a `ScreenRect`, which has an upper and lower corner, stored as `ax,ay,bx,by`. This then sets the update section to those coordinates using display commands, and reads out the appropriate parts of the pixel buffer to fill it in. This allows me to update the screen for the render and the minimap at different times.

The Tasks

Each task struct is initialized with a deadline, a period, a priority, and a function pointer. The tasks are stored using a min-heap data structure, sorted by deadline, and then by priority. The deadline that is sorted by is not the same one used in initialization. Rather, when a task is inserted into the heap, two more fields are set: the `real_deadline`, which is when in real time this task actually needs to complete by, rather than the simple offset used during initialization; and the `next_add_time` (if the task is periodic) which says when the task should next be added into the heap. Every cycle, we snapshot the current number of cycles, a sort of timestamp, just so it doesn't change while we are processing. We then loop through all our periodic tasks, and add them into the heap if it's past the next add time and they aren't already in the heap[3]. Then, once we have looped through all of the tasks, we pop the top one (if there is one), and run it. Since there is no preemption and every function is blocking, this ensures that the task will run to completion and we won't cycle again until the next time we could do anything. The program only has one interrupt, that being of course the SysTick handler.

Additional Notes

Before we get into the scheduling, let's round off this section with a few miscellaneous thoughts:

- The first room is preset so that you don't start the game looking at a wall
- If you squint at the binary data for the textures, you can actually see the shapes pretty clearly
- I never want to type `uintXX_t` ever again (I really should have used some typedefs)
- Somehow it never really came up in this writeup but I actually store all of the cells in a grid. Like all the cells that will or could exist already exist, but they only store the walls. The active cells also have to store the x and y coordinates, and if they were hit, as well as a pointer to the cell in the grid they represent
    - Due to this, I also have to make sure to set all walls of a cell to solid whenever it becomes inactive. Otherwise, every time it regenerated it would lose another wall, never filling in the previously open walls

Scheduling

sigh So the reason this is going to be such a nightmare is because two tasks specifically, `update_rays()` and `draw_map()`, scale with the number of active cells. This means that their runtime can vary by theoretically up to the max number of active cells, or in this case 30 TIMES!! So I'm just going to do one analysis assuming that the times I got benchmarking are set in stone and cannot change, and then I'll do another analyzing worst possible case performance.
With that in mind, here is a list of the tasks, their periods, their priorities, and their runtimes

- Raycasting
    - Period: Every update
    - Priority: 100
    - Runtime: 19.72ms
- Updating Columns
    - Period: Every update
    - Priority: 90
    - Runtime: 5.125ms
- Drawing the Render
    - Period: Every update
    - Priority: 80
    - Runtime: 13.515ms
- Drawing the Map
    - Period: Every update
    - Priority: 80
    - Runtime: 11.245ms
- Updating the cells
    - Period: Every update
    - Priority: 90
    - Runtime: <0.05ms
- Reading the joystick and moving the camera
    - Period: 10ms
    - Priority: 120
    - Runtime: 0.6ms
- Generating a new cell
    - Period: Every update
    - Priority: 150
    - Runtime: <0.05ms

Generating a cell is technically a periodic task, it just exits immediately when there is no cell to generate. And yes I did make sure that was a cell to generate before running it to benchmark it.
Adding up the total times for everything but the joystick, we get about 49.6ms. So theoretically we could set the update time to 50ms, leaving almost no slack? Well no, because then we would have 5 joystick reads per update, giving us 52.6. If we instead increase that to an update every 60ms, we have one more joystick read, giving us 53.2, which is still well within our update time.
Again, since there are no aperiodic tasks, no interrupts, and **definitely no variability in the runtime of these functions**, we can guarantee that they will all complete without issue before their next period, meaning that it will perfectly cycle every 60ms.

If we were to instead assume worst case performance (ie perfectly linear function scaling and 30 active cells, so 30x), we would have the "Raycasting" and "Drawing the Map" tasks change as follows:

- Raycasting
  - Period: Every update
  - Priority: 100
  - Runtime: 591.6ms
- Drawing the Map
  - Period: Every update
  - Priority: 80
  - Runtime: 337.35ms

Giving us a new total update time of about 947.6ms. Factoring in the joystick, we'd need an update time of about 1020ms to meet our deadline[4]. In other words, *absolutely miserable*.
Although <1fps is not ideal, we can be essentially positive that every task will always meet its deadline. The only other task that scales at all is drawing the render, which scales with the number of "wall" pixels in view. However, with textures off it is literally just a data read and an array write, so basically nothing. With textures on it is simply a few cycles of integer arithmetic.

---

1. A texel is a **Tex**ture **El**ement, similar to a pixel. Instead of being part of a picture, it represents a single element of the base texture, before it's been scaled and rotated and shaded and projected etc. ↩ ↩
2. We can do this efficiently because all the cells actually exist in a big 2D array, so we can just check the coordinates directly. More on this in the closing notes ↩
3. We know if they are already in the heap because a flag gets set when they are added to the heap and unset when they are removed ↩
4. We can plot $947.6 + \frac{0.7x}{10}$ (the actual time needed) and just x (the update time) and see that they intersect at ~1018.9 ↩