Contents

Daedalus

A game about traveling Daedalus' labyrinths, with a doom style (pseudo) 3D renderer in the terminal.

Run the game

Just run `make` on a Linux terminal.

Basic game logic

We store in memory the (x, y) coordinates of the player and the angle it's looking at as floats. The angle is in the range [-pi, pi).

We store the different maps in memory, as a "2D array" of bytes. We use `0` to represent a space and `1` to represent a wall.

Movement and collision

To do movement, we calculate the "forward" vector given a particular position, using some trigonometry. Then, we add that vector multiplied by some speed, and we store it in temporary registers.

Then, we check for collision. We truncate the resulting coordinates and we check if the value of the map at that location. If the player is out of bounds, we decide to treat that as a wall or a space according to a variable in memory called `oob_collision_mode`. If the resulting position is inbounds, we update the player position in memory[^1].

[^1]: We are aware that this technically allows the player to move through some corners, but it is not in our best interest to fix this behaviour.

A note about trigonometric functions

We use `sin` and `cos` extensievly in Daedalus. Using the C math library requiered extra linking, which we deemed unnecessary. Instead, we implemented our own trig funcitons using Taylor series. Concretely, we used the following series:

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!}$$ $$\cos(x) \approx 1 - \frac{x^2}{2!} + \frac{x^4}{4!}$$

This approximation only starts to deviate around $\pm\frac{\pi}{2}$. If the angle is greater than this, we add or subtract π to be inside the accurate range and change the sign of the result, to get an accurate value (this is why we restrict the player angle)

Rendering

We have implemented a primitive, but functional rendering system.

It has two main components: raycasting and the shader pipeline.

Raycasting

Raycasting is a necessary component to create a 3D renderer. Raycasting usually work by calculating intersections between lines and planes using math. We deemed this approach effectively impossible for us to implement in x86 assembly. So we have used an alternative approach.

To shoot a ray at a specific angle, we calculate the forward direction (same thing as with player movement) and we scale it by something small (we have chosen `0.005`). Then, we add it a bunch of times until we hit a wall. Or, rather, until the position of the ray is inside of a wall. The number of times we had to add the vector is proportional to the distance, which is all we care about.

This method would likely be unacceptably slow in an actual application where we have to shoot thousands or millions of rays at very high accuracies. But we only shoot a couple hundred rays per frame and most inaccuracies are masked by the low resolution of the game.

Shader pipeline

To do the actual graphics, we have used a shader pipeline similar to the ones used by graphics cards. The shader goes through each pixel, does some calculations which only depend on the coordinates of the pixel in the screen and the "uniforms" passed. Since we only have one shader, we don't pass any uniforms, we just have the necessary data in somewhere in memory.

To be clear, we are aware that we are completely throwing out of the window the reason graphics cards work this way. Namely, paralelization. Implementing paralelization in assembly is like going skydiving while blindfolded. And without a parachute. No, the reason we have this pipeline is because it nicely separtes the rendering part from the raycasting part, it is easier to debug, and it is more flexible. In general, we found that it is a very useful abstraction thanks to which is was feasable to add fade in and out effects.

Speed, especially with our modern hardware, isn't our primary concern.

From raycast to shader

We still need some glue to go from the raycast to the pixels on the screen. Before starting the shader process, we throw rays at angles next to the angle the player is looking at. We shoot one ray for each column in the screen. Then, we convert that ray into the height of the wall that should be rendered at that column. It has to be inversely proportional to the distance, something like `height = constant/distance`. In reality we have to divide by the cosine of the offset of the angle to make equivalent distances be planes instead of shperes. So, the actual function is `height = constant/(distance*cos(offset_angle))`.

Extras

Scenes

We also have a scene system. This is used to navigate between the pause menu, the cutscenes, and the actual map. We have 3 actual maps and a fourth special maps. There is a cutscene before each map

Terminal dimensions

The game automatically detects the size of the terminal and adjust accordingly. To do this, you have to do a `IOCTL` call. However, that messes the non-canonical mode so in the build system we run a program that gets the size of the terminal and saves it to disk, which the main program can then read. This is, admittely, a fragile-at-best solution.

OS and libraries

This game only runs on Linux. The game was developedd on WSL (Windows) and Lima (MacOS).

The ony library that is used is the C standard library. Specifically, we use the folliwing functions:

- `printf`
- `fopen`
- `fread`
- `fclose`
- `time`
- `sleep`
- `exit`

We also use two system calls. We use `READ` for user input. We use `IOCTL` for getting the dimensions of the terminal, setting the terminal to non-canonical mode and to disable echoing.

Some terminals don't render the game properly, for example, the stock mac terminal. It is confirmed to work on Warp, the VSCode terminal and Windows Terminal.