David Hiser

How To Run:

Set bitmap pixel size to 4x4. Set bitmap resolution to 512 x 256. Set base memory address to 0x10040000 (heap). The file “map.txt” must be in the same folder as MARS. Open both files, then go into the Settings tab and select “Assemble all files in directory.” Then, select the “projectmain.asm” file, and compile and run it.

How To Play:

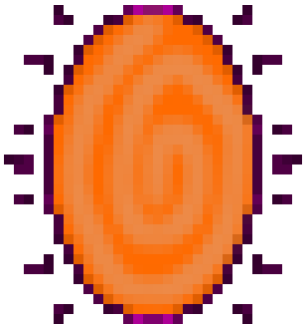
Controls:

W to move forward. S to move backwards. A to rotate left. D to rotate right. Spacebar will end the program.

Objective:

You are trapped in a Labyrinth, with only a very dim torch to light your surroundings. The goal is to find a magical portal that allows you to escape.

The portal looks like this:

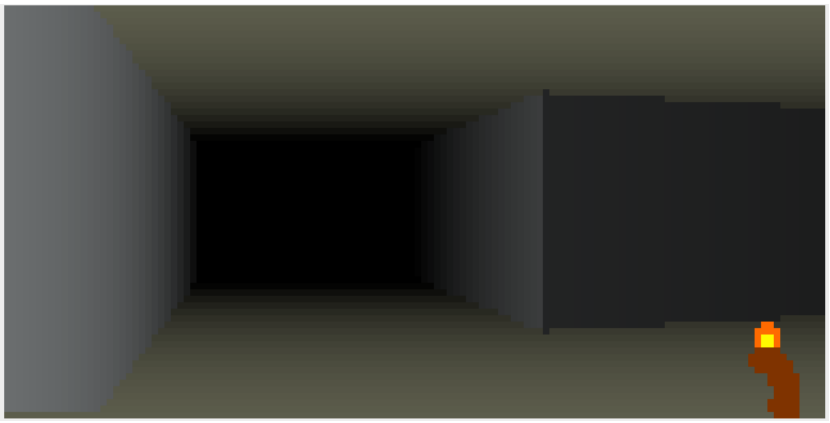


Helpful Hints:

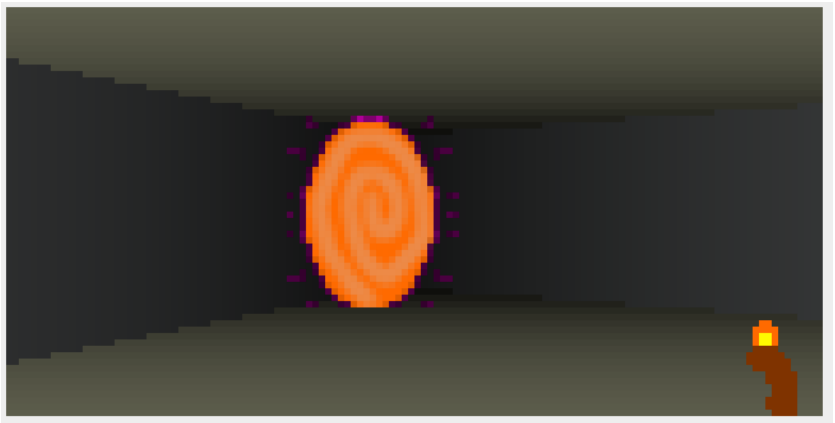
This program runs exceptionally better if MARS is freshly booted before it is run.

You can hold down the buttons to move. However, because of the relatively slow speed of the rendering algorithm, the screen may flash while holding down buttons, so it may be hard to understand your surroundings.

Sample Runs:



A hallway showing a crossroads. The player can go straight, or right.



The portal in the corner of the portal room.

The victory screen.

How It Works:

The main game loop first checks the position of the player. If they are in the tile with the portal, the program jumps to the victory screen. If it does not end, it then checks for input from the keyboard. If none is found, it continues to loop until input is found. If input is found, it is then handled. If it is a space, the program exits. If it is W, A, S, or D, then the movement calculations are done. These involve basic translations of the player position and camera plane based on pre-calculated values of sin(15 degrees) and cos(15 degrees). This fixes the rotation speed to 15 degrees per button press. After the player position and camera plane are translated, the screen must be re-rendered.

The walls are drawn using an algorithm called raycasting. This algorithm creates a 3D rendering of a 2D map. However, looking up or down is not possible. It is often called "2.5D" because it is not truly a 3D world, but rather a first-person POV rendering of a 2D world. The algorithm works by casting a ray out from the camera plane for every column of pixels on the screen using another algorithm called DDA (digital differential analyzer). Once the ray intersects a wall, its distance to the camera plane is recorded. This distance is used to calculate the height of the column of pixels to draw. The distance to each column is then stored in an array in the heap, called the ZBuffer. Every column of pixels is drawn centered vertically. This is the same algorithm used by the 1992 game Wolfenstein 3D, shown below.

The color of the column of pixels drawn is determined in my program based on the side of the wall it is on, X-facing or Y-facing, as well as the distance from the player. The color is first divided by a small value so that X and Y facing walls are a slightly different shade. The color is

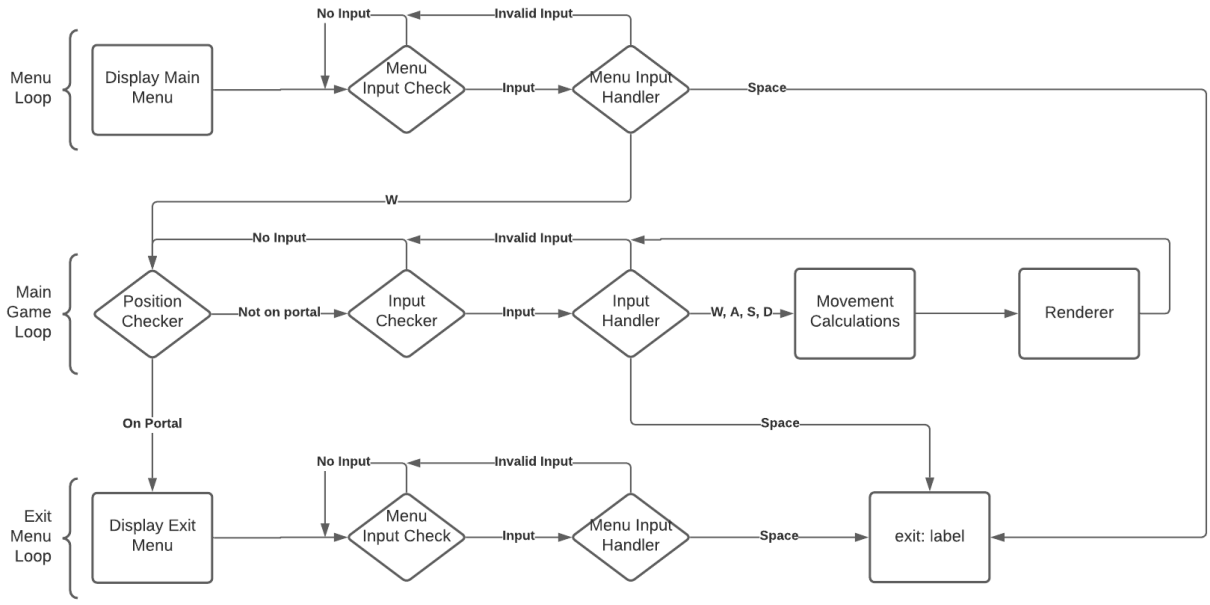
then divided by a function of the distance, leading to it fading to black past a certain distance. This simulates the lighting from the torch the player is holding. The color division is performed by separating each R,G, and B component of the color and dividing them, then recombining them.

Before the walls are drawn, however, the floor and ceiling are drawn. The distance from the player to the row of pixels on the floor and ceiling is calculated, then the ceiling color is divided by a function of the distance, like the walls are, simulating the torch lighting. The floor and ceiling are mirrored, so the calculation is only done once, then the color drawn to 2 different rows. The walls are drawn over top of the floor and ceiling.

After the walls are drawn, the portal must be drawn over top of the walls. The portal texture is a sprite held in memory. The portal's location relative to the player is calculated, then it is transformed to find its position and size on the screen. Then each column of pixels in the screen-mapped sprite is checked for 4 conditions. The first 2 check whether it is in bounds. The third checks whether it is in front of the camera, rather than behind it. The last then checks whether the column is in front of the nearest wall by comparing its distance to the player to the distance stored for the wall in that column in the ZBuffer. If all the conditions are passed, the sprite texture coordinates are mapped to the screen and drawn for that column.

Lastly, after the portal is drawn the torch is drawn over everything. This involves a simple function that loops through each pixel value stored in the memory for that texture, then writes it to the memory location on screen.

Program Flowchart:



DDA Pseudocode:

```
deltaDistX = abs(1 / ray X component)
deltaDistY = abs(1 / ray Y component)
if ray X component is negative
       stepX = -1
       sideDistX = (posX - mapX) * deltaDistX
else
       stepX = 1
       sideDistX = (mapX + 1.0 - posX) * deltaDistX
if ray Y component is negative
       stepY = -1
       sideDistY = (posY - mapY) * deltaDistY
else
       stepY = 1
       sideDistY = (mapY + 1.0 - posY) * deltaDistY
while wall hit == false:
        if sideDistX < sideDistY
             sideDistX += deltaDistX
             mapX += stepX
             side = X     #records the side of the box the ray is hitting
        else
             sideDistY += deltaDistY
             mapY += stepY
             side = Y     #records the side of the box the ray is hitting
        if Map[mapX][mapY] = wall
             wall hit = true
```