Labyrinth: Escape the Maze!

Computer Graphics (COS 426) Final Report

Julianne Knott and Charlie Smith

Abstract

A first person game, created using ThreeJs, in which the user navigates through a randomly generated 3 dimensional maze.

Introduction

Goals

Our goal was to create a fun, first person game in which the user tries to find a target in a randomly generated 3D maze. We believe that anyone could have fun playing our game. In order to make it more accessible to various users, we allow users to customize the difficulty of their maze.

When planning our project, we broke down our goals into three phases: the MVP, Next Steps, and Further Extensions. Our MVP goals consisted of generating individual walls in a 2D grid pattern, setting up first person navigation controls, and making sure the user couldn't pass through walls. This made it possible to make a simple predetermined 2D maze. Our Next Steps goals consisted of experimenting with different ways to make the maze walls more visually appealing, randomly generating a 2D maze, adding a start prompt with the ability to customize the difficulty of the Maze, and switching to a lighting scheme to give it the appearance of the user navigating the maze with a flashlight. Finally, our Further Extensions goals consisted of polishing our intro and outro sequences, extending the maze to three dimensions, and adding strategy elements to the game (for instance health / hunger, multiplayer, more sophisticated maze interactions and special abilities).

Prior Work

There are many maze-type games online. Most of these are simple 2D mazes, similar to a maze you might solve on paper. A few are 3D, for instance this maze from mathisfun.com[1], however while the graphics are first person and three dimensional, the maze itself is 2D. We wanted to make a more interesting maze that was truly in three dimensions. This would result in more interesting paths, and would enable the user to navigate up and down in addition to forwards, backwards, left and right.

[1] https://www.mathsisfun.com/games/maze-3d.html

Another work that our project is similar to is the popular video game, Minecraft[2], in the way that the user navigates through the world around them. Both use a first person style perspective, and the controls are similar as well. While both also use a fairly block-based style, Labyrinth uses textured walls for some added visual effects, as well as thinner wall dimensions so as to use space more optimally.

We think that our game will work well for audiences that like puzzle solving type games and that are generally up for a challenge. We believe that the controls are easy and intuitive enough to be used by anyone, and therefore anyone can approach the game and have fun with it. The concept is also fairly intuitive, so the learning curve to the game would be quite small, and thus we believe Labyrinth is able to be appreciated by all.

Methodology

The main pieces that had to be implemented in order to create this game were creating a maze, and allowing the user to navigate the maze. The creation of the maze required first, a way to construct a solvable yet non-trivial maze at random, and secondly a realization of the maze in meshes. Allowing the user to navigate through the maze required both the ability to freely move the perspective with respect to the maze, and also account for any collisions with all surfaces of the maze.

The Maze

To create our game, we first worked on constructing a random 2D maze. To do this, we made an algorithm based on Depth-First Search, because it is known to favor the creation of longer paths, which we believed would result in a more enjoyable maze (in contrast to other algorithms which produce many short deadends).[3] This algorithm considers a 2D Array of "cells" and recursively marks a cell as visited, before recursively visiting its neighbors and removing the walls between it and its unvisited neighbors. The result is a maze where all cells are reachable from one another, along some valid path.

In order to avoid rendering sections of wall only to remove them later, we separated our maze generation algorithm from the rendering of the maze and have it compute the maze before anything is rendered. To do this, our maze generation algorithm outputs a list of wall locations that should be excluded when the walls are rendered. In order to make it efficient to check if a wall's position is contained in the list before adding it to the scene, we sort the list of excluded walls first by x coordinate and then by y coordinate. Then, when we iterate over the walls to add to the scene, we only need to make one pass over the list of excluded walls, instead of having to search the list multiple times.

[2] https://www.minecraft.com/download
[3] https://en.wikipedia.org/wiki/Maze_generation_algorithm

For simplicity, each wall separating adjacent cells in the maze is represented by a Mesh created using a ThreeJs BoxGeometry[4] with a ThreeJs Material[5]. While our first iteration of the Maze was rendered using a MeshNormalMaterial[6], which chose a color based on the normal of the surface, we wanted to make our Maze more visually interesting than a series of smooth, mostly monochromatic walls. The first thing we tried in order to make our maze more visually interesting and pretty was creating triangular subdivisions to the wall geometries and adding noise to the new vertices in order to create texture and also add more colors (since each small fragment would have a different normal vector). This created a fun appearance, but was also fairly chaotic, and if it was applied to both the walls and the floor and ceiling, made it really hard to navigate. Additionally, the MeshNormalMaterial didn't allow for adjustments based on lighting which were integral for extensions to our project which we wanted to implement.

Instead, we used a ThreeJs MeshPhongMaterial[7] with an added ThreeJs Texture[8], which we load from a .png file. We applied the texture as both the material's color and its displacement. This gives our walls a water-like appearance. Additionally, we give our maze material a mysterious purple hue and increase its shininess in order to maximize the effect of the first person "flashlight" effect (which we will discuss later). We also tried incorporating mirrors, reflective surfaces, and lights into the walls of the maze using ThreeJs Reflector[9] and ThreeJs RectAreaLight[10]. However, we found that this substantially slowed down our game without improving the appearance in the way we had hoped. Consequently, the final version of our game does not include these additions. Finally, we made the maze color fluctuate as a function of the timestamp and the distance to the target. Thus, if you are closer to the target, the color will pulsate faster and if you are farther from it, it will pulsate slower. While this doesn't yet work perfectly while the user is moving, it still provides a more interesting level of strategy for solving the maze efficiently and augments the appearance of the maze.

Once we had constructed a convincing 2D maze, we worked on expanding our maze to three dimensions. While we considered expanding the depth first search algorithm to work with a 3D grid instead of a 2D grid, we believed that this would result in unnecessary computational overhead and more complicated code without necessarily increasing the quality of the maze. Instead, we chose a simpler approach. Since we could reach any cell in the 2D maze from any starting cell in the maze, we knew that if we stacked multiple 2D mazes on top of each other, so long as there was at least one opening between the two, all of the cells in the first would be reachable from all of the cells in the second and vice versa. This logic applied for any number of stacked 2D mazes so long as each 2D maze is fully connected and each pair of adjacent 2D mazes has at least one path between them. Consequently, our implementation of a 3D maze is an

[4] https://threejs.org/docs/#api/en/geometries/BoxGeometry
[5] https://threejs.org/docs/?q=phong#api/en/materials/Material
[6] https://threejs.org/docs/?q=Material#api/en/materials/MeshNormalMaterial
[7] https://threejs.org/docs/?q=phong#api/en/materials/MeshPhongMaterial
[8] https://threejs.org/docs/?q=Texture#api/en/textures/Texture
[9] https://sbcode.net/threejs/reflector/
[10] https://threejs.org/docs/?q=Area#api/en/lights/RectAreaLight

(n x n x n) grid consisting of n stacked (n x n) 2D mazes where each set of adjacent “floors” in the maze is separated by walls in all but one cell pairing.

In order for there to be an objective in our game, we also added a floating cube to serve as the target destination. The cube is placed at a random cell in the maze and is rendered as a rainbow wireframe.

The Controls

We tried to make the controls for our game as simple and intuitive as possible, so the user could spend less time learning how to play, and more time playing. This was not trivial, however, given the range of motion that we decided to allow the player to have. This includes rotation up, down, left and right, and translation up, down, left, right, forward and backward. To account for all these possible directions, and to also allow the user to freely move their mouse without concern of moving awry, we used the two commonly known sets of movement keys: WASD and the arrow keys. We assigned WASD to control the forward back, left and right translation, and the arrow keys to control the field of view rotation. Finally, to ascend and descend, we used space and shift, respectively.

We had considered a number of alternatives to the control scheme, but ultimately found that this felt the simplest and most natural. One such example is using the mouse to control orientation. While this is natural, it can be a bit difficult to control well, and a bit annoying at times, and constant attention to the mouse’s position is required. Another design choice we made was to make the translation directions independent of the user’s orientation. As an example, the up key (space) would always send the user in the same direction, regardless of whether the orientation of the user was facing forward or not. We thought this was easiest and most natural because it allows the user to travel parallel to the ground, without worrying about their orientation with respect to the ground.

Results

To evaluate how successful we were at implementing our game, we compared our final product with the goals we had initially set out in our proposal. We were able to reach all of the goals that we initially laid out for our MVP: making a basic, predefined 2D maze that the user could navigate with first person controls. We also succeeded in meeting all of the goals we had specified for our Next Steps: we implemented a random 2D maze generator, added an intro popup that allowed users to customize the maze difficulty, gave the walls textures and color to make the appearance more interesting, and added “flashlight” style lighting to enhance the first person navigation experience. Finally, we were able to implement many of the goals we had left for Further Extensions: extending our random maze generation to three dimensions, making the maze more interactive and visually appealing by making the wall color fluctuate as a function of the distance to the target, and polishing our intro and outro sequences instead of relying on popup notifications.

To further measure the success of our game, we had several of our friends try it and give us their feedback. Overall, they liked the “purple water” theme and that we had added normal mapping with the water pattern so that the walls weren’t boring and smooth. They also liked the pulsing color effect and flashlight effect. They also provided valuable feedback on the need to improve our introductory sequence with clear instructions for how to play the game and our concluding sequence to make it more exciting when you won the game.

Discussion

This project helped us become more familiar with ThreeJs. We also learned more about reflections and collisions (and the difficulties of implementing them efficiently). We also became more familiar with Javascript, which neither of us had substantial exposure to prior to taking this course.

Conclusion

In conclusion, Labyrinth is a fun game to solve a 3D maze puzzle. Since it is randomly generated and has many options for difficulty levels, it can be played many times while still having a fun, enjoyable experience. While we are happy with the overall quality of our game, in the future, we’d like to continue working on our goals for Further Extensions by adding strategic elements to the game (for example: hunger, items, maps), enabling a multiplayer experience, having more interaction between the user and the maze itself, and continuing to make the wall appearance more interesting and varied throughout the maze.

Contributions

Julianne

I worked primarily on the Maze. I wrote the code that generated the 2D and 3D mazes, setup the functionality to allow for custom difficulty levels, and experimented with different ways to style the walls of the maze with various materials and make the color fluctuate as a function of distance to the target (including using Reflectors and Lights, though we ultimately chose not to use any since we didn’t like the way they looked).

Charlie

I mainly worked with enabling navigation, including camera movement, wall collision detection, spotlight movement, and other things involving controls. I also helped with (and am currently working on) developing the intro and outro scenes and the ability to replay levels.

Works Cited

Bradley, Sean. “Reflector.” (https://sbcode.net/threejs/reflector/)
Math is Fun. (https://www.mathsisfun.com/games/maze-3d.html)

“Maze Generation Algorithms.” Wikipedia.
(https://en.wikipedia.org/wiki/Maze_generation_algorithm)

Minecraft. (https://www.minecraft.com/download)

ThreeJs. (https://threejs.org/)

Course Website (https://www.cs.princeton.edu/courses/archive/spring21/cos426/)