Procedurally Generated Maze in Unity

Adrienne Cohrt
s184426@dtu.dk
Technical University of Denmark
Lyngby, Denmark

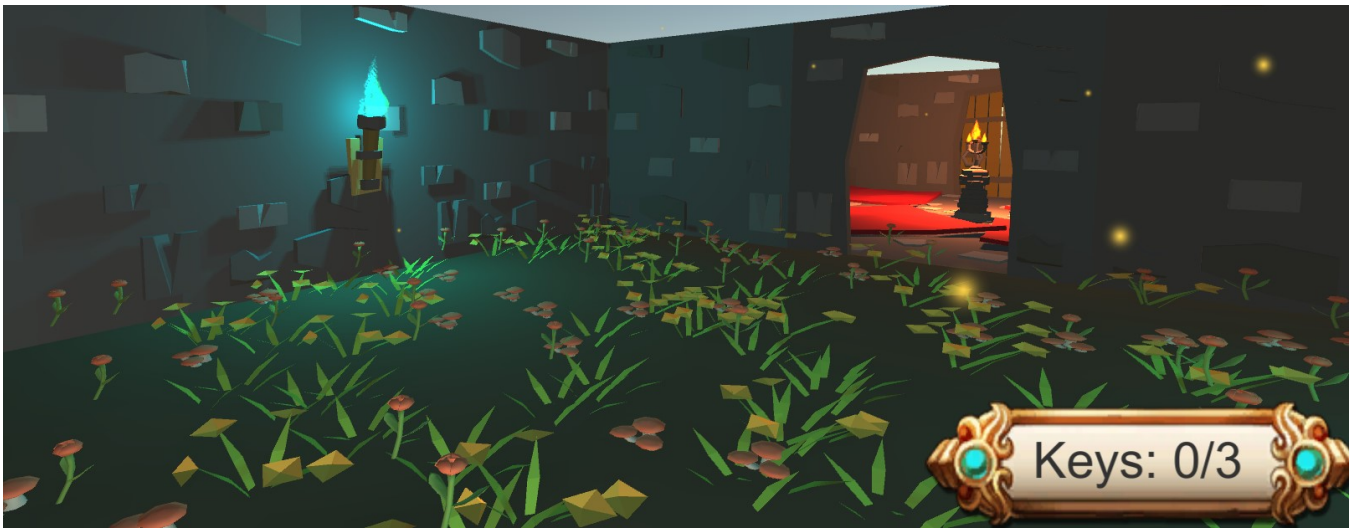

Figure 1: Maze environment generated procedurally

ABSTRACT

This work outlines the development of a simple procedurally generated maze within the Unity platform by myself and Yuting Han. In order to make the game as playable as possible user interfaces, player/maze interactions, and other features have been explored and implemented throughout the development. While many examples of this already exist, this version represents the application of our own knowledge and learning into a working product.

KEYWORDS

Maze; Procedural Generation; Game

1 INTRODUCTION

Many types of video games are designed with specific objectives or tasks for the player to complete in order to progress through the story line. One of the most common of these devices in adventure games is the dungeon or maze setup. While a seemingly basic concept, the complexity and design potential for these have endless possibilities in a platform like Unity. In the process of building our own maze we explored several main aspects which were all incorporated into the finished product including room design, the procedural generation of the maze, player interactivity, lighting, and occlusion.

2 DESIGN AND IMPLEMENTATION

Prefabricated rooms. The playable area of the maze is divided into a grid and populated with prefabricated rooms (see Figure 2). Each room occupies a 3x3 square space so that they can easily be lined up and arranged as needed. In order to avoid a rotation problem or having to create the same room multiple times with multiple orientations, both a doorway and a wall are present at the center of each room side. During generation, once the rooms are placed on the grid these doors and walls are activated or deactivated as needed creating barriers and connections between rooms.

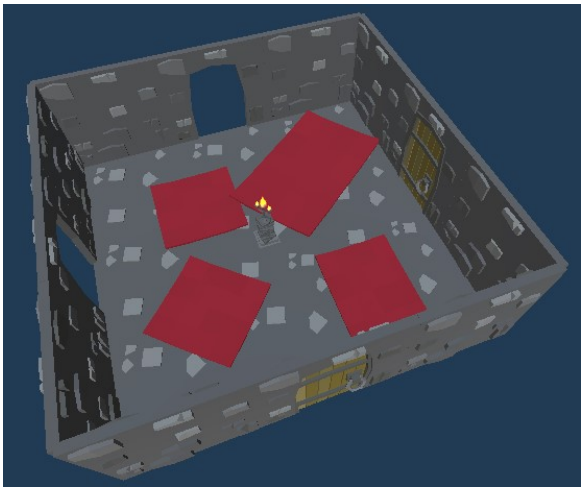

Figure 2: Prefabricated room with all doors active

Maze Generation. The maze itself is generated using a depth first search (DFS) algorithm [4]. The DFS algorithm works by compiling a list of valid neighbors of the current location. A single neighbor from the list is selected and becomes the new current location. The process then repeats until either a goal location is found or there are no more valid neighbors. Since DFS keeps track of where it has been, in the case of no valid neighbors the algorithm will backtrack until it finds a location which does still have valid neighbors and it will proceed from there. In our application we have not set a goal location so that the algorithm will run until the entire available area has been mapped and filled. It is of interest that one of the criteria to being a valid neighbor is that said neighbor has never before been visited. This prevents the algorithm from making connections to already existing paths which results in a maze with only branches. After this, the prefabricated rooms are assigned to each position in the grid based on their location and how many adjacent connecting rooms there are (see Figure 3).

Cycles and diagonal structure. In an attempt to make the maze more difficult, cycles were added to the maze generation since the DFS itself will not produce them. Cycles are areas in the maze where it is possible for the player to walk in a circle and end up in a room that they have already visited. Since the DFS algorithm does not produce cycles a special function was created which goes through each room in the grid, finds places where there are no doors, checks the suitability of the room on the other side, and then activates a doorway based on a random number generator.

Player Interactivity and Feedback. Once the maze is generated there are several ways the player can interact with the game. The user interface is designed to display an updated count of the players items and inform them when the end of the maze has been reached [1, 3]. Solid doors in the maze are also impassable to the player until a certain criteria has been met [2]. The puzzle rooms contain a floor of lava, generated using Unity fabric, which when touched, teleports the player back to the entrance of that room. Other minor player features include a timer built into the user interface, audio feedback when a player touches the lava, and both a pause and restart option during the running of the maze.

Lighting Light in the maze comes from several sources. Within each prefabricated room are wall sconces which have a flame generated using the Unity particle system [5] and an attached light source. These appear on most center walls in the maze where doors have not spawned. The fireflies which appear in certain rooms also have point light sources attached to them so that they illuminate the ground and walls when they fly close enough. We also made the decision to attach a light source to the player capsule as well. This was originally done to provide additional visibility in rooms with little or no other light sources, but it had the additional benefit of giving the illusion of the player holding a torch as the light moves with the player and illuminates dark areas only as they are approached.

Occlusion One persistent problem throughout development was lag and very low frame-rates during the game play. This was most noticeable when the player would be positioned near the edge of the maze and faced in towards the direction of the rest of the maze.

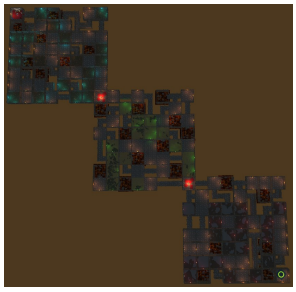

Figure 3: Maze fully generated

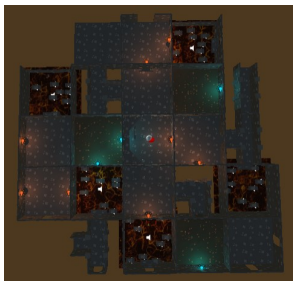

Figure 4: Maze as generated using occlusion

While Unity automatically takes care of frustum occlusion, where objects that are outside the view of the camera are not rendered, it has no built-in function for dynamic occlusion when the game scene is not generated in advance. Since our entire maze is procedurally generated, a new solution to this problem was found. Upon generation of the maze, a list of all rooms is made. Utilizing Unity's built in transform.position function the current position of the player is checked in each frame. Any room located outside a pre-specified distance from the player is deactivated while any room within this distance is activated (see Figure 4). This approach successfully reduced the sheer number of meshes, particles, and other child objects that were active and rendered at any given time in the game scene thus dramatically improving the game play experience.

3 DISCUSSION

A final maze design was run several times with test subjects using both a regular DFS generation and a DFS generation with cycles. The intention was to increase the maze difficulty by making it possible for a player to "get lost" and walk in circles, but we found the opposite to be true. With cycles present in the maze, the player was able to traverse across the maze in a more direct manner without being forced to backtrack along a long branch when finding a dead end. It was also noticed that with the implementation of the occlusion to reduce lag, certain areas of the maze that generated in long straight corridors would allow the player visibility outside the maze and into the void. Despite this we felt that the occlusion was a necessary addition to the project, and given more time this issue could have been fixed with the addition of fog to obscure player vision at a distance.

REFERENCES

[1] Coco Code. 2020. Create great GAME OVER screen in Unity UI - Unity tutorial. Video. (21 November 2020). Retrieved May 14, 2022 from https://www.youtube.com/watch?v=K4uOjb5p3Io.

[2] Code Monkey. 2019. Simple Key Door System in Unity. Video. (21 November 2019). Retrieved May 14, 2022 from https://www.youtube.com/watch?v=MIt0PJHMN5Y.

[3] Vincent Quarles. 2015. Building a Dodger Game Clone in Unity. (2015). https://www.sitepoint.com/building-a-dodger-game-clone-in-unity/.

[4] SilverlyBee. 2021. Procedural Dungeon Generator in Unity [TUTORIAL]. Video. (21 September 2021). Retrieved May 14, 2022 from https://www.youtube.com/watch?v=gHU5RQWbmWE.

[5] Jimmy Vegas. 2014. Mini Unity Tutorial - How To Create Fire - Beginner. Video. (21 November 2014). Retrieved May 14, 2022 from https://www.youtube.com/watch?v=qShjsxopbfQ.