Procedurally Generated Multi-level Mazes with Guaranteed Solvability
Justin Shi
Advisor: Holly Rushmeier

Abstract

In this project, my goal was to create a three-dimensional maze game, played from the first-person perspective, that would consist of several levels existing in the game space simultaneously, which the player can move between freely. Each level is generated procedurally, such that the layout of the maze and levels are different each time the game is played, and each level has a "GOAL" block placed in it somewhere arbitrarily; to complete the game, once must collect the "GOAL" block on each level. In the interest of increased complexity/difficulty, each level also may have sections that are arbitrarily surrounded by a barrier, and can only be reached by collecting a corresponding key block; key blocks are in turn arbitrarily placed across levels, with the key block for a given barrier possibly being placed on a level different from that of the barrier. Furthermore, some key blocks may not be spawned into reachable positions until other key blocks are first collected and their barriers deactivated. The game thus includes a method of ensuring that, at any given time, the player is able to progress in the game in some fashion (whether that be collecting a "GOAL" block or deactivating some barrier somewhere). Meanwhile, there are also enemy entities that are spawned across each level, and at regular intervals up to a population limit, that the player must avoid coming into contact with and can choose to combat using a weapon fired from the first-person perspective. These enemy entities utilize an A* search algorithm to find paths to the player, and will begin to follow the player should they come too close. Finally, remaining efforts were focused on applying some basic sound effects, textures, and background music, as well as introducing some graphical effects, to produce a somewhat ominous atmosphere.

Development Process and Key Challenges

Representation of Layout: The primary challenge of this project was, obviously, the implementation of an algorithm that would procedurally generate some sort of representation that could be used to build our maze of cave-like levels. To do this, I started off with assistance from Sebastian Lague's tutorial on procedural cave generation¹, and had a script create a two-dimensional array of integers, randomly initialized to 1 or 0, given some parameters such as fill-percentage, size, and pseudorandom number generator seeds. This 2D array would serve as a sort of floor plan for a level, with a 1 representing a floor tile of a room that the player could exist in, and 0 representing a space outside all rooms and out of bounds. The next step after this was to introduce a smoothing process to this array, to mold the representation from

¹ https://unity3d.com/learn/tutorials/s/procedural-cave-generation-tutorial
seemingly random noise into something more resemblant of a system of caves or rooms. This was done through what were effectively convolutions with a blurring kernel, where, after each cycle, each cell in the array would be set to the majority of its surrounding neighbors (i.e., if 6 out of 8 neighbors of a cell were set to 1, that given cell would be set to 1 after this cycle). By playing around with the fill-percentage, or the approximate percentage of array cells that were set to 1, and the number of smoothing cycles, I was able to produce layouts with some interesting structures² ³.

**Mesh Generation:** The next step was to implement some mechanism for turning this layout into a mesh that could be rendered in the game by the Unity engine and used for collision detection and lighting; I needed to turn my representation into an actual game world. To do this, I used the Marching Squares algorithm with our 2D array to create floor meshes that took the shape of rooms⁴. To quickly recap the Marching Squares algorithm: essentially, each cell in the 2D array is a control node, and squares are formed by 4 control nodes as corners. Each square then can be assigned a number, from 0 through 15, based on the binary string formed from its 4 corners (active (1) or inactive (0)); I then map these numbers to a certain type of mesh shape for that given square. After creating the floor mesh in this way, I simply needed to find the vertices of each room that make up its outline (which I did using the knowledge that two vertices that share an outline edge will only share 1 triangle in the floor mesh), and then create wall meshes by extruding these outline edges vertically, or adding new vertices above them and connecting them appropriately⁵.

**Ensuring Connectivity, with Efficiency:** The issue at this point was that I had a bunch of procedurally generated rooms with properly rendered floor and wall meshes, but there was no guarantee of connectivity. I would commonly see rooms that were completely disconnected from each other, with no way for the player to get from one to another. Thus, the next step was to develop a process for taking a map of disconnected rooms and ensuring that, from any one room, it would be possible to move to any other room. This was done by first running breadth-first searches on the map to remove any tiny holes and to get a list of all rooms, where I have a list of coordinates that make up each room⁶. Important things to note about the breadth-first search are that I can map each coordinate to the room it belongs to and, because it stops adding new coordinates to examine when the neighboring coordinate of a cell is no longer a room, I also have a built in mechanism for recording which coordinates are on the edge of a room, which is vital.

---

² MapGenerator.cs: GenerateMap, RandomFillMap, SmoothMap
³ Note that this layout creation would be repeated in a loop 10 times, as 10 levels exist concurrently as the game runs; everything that follows about mesh generation, connectivity, and reachability is likewise repeated per level.
⁴ MeshGenerator.cs: GenerateMesh, TriangularSquare, MeshFromPoints
⁵ MeshGenerator.cs: CreateWallmesh, CalculateMeshOutlines
⁶ MapGenerator.cs: public struct Room, GetAllRegions
I then went through our list of rooms and, given a pair of rooms, connect them. To do this, I pick a random edge coordinate from each room and use Bresenham's line drawing algorithm to carve out a passage between these two edge coordinates, with some small radius. This method of connecting rooms is where I began to strongly diverge from the suggestions in Lague's aforementioned tutorial, in the aims of implementing a quicker and more computationally efficient algorithm for connecting all rooms. Lague's method would have me go through each pair of rooms, and then, for each pair of rooms, go through each pair of edge coordinates, and find the two edge coordinates that were closest, then use Bresenham's algorithm to create a passage between those two coordinates. This would create arguably nicer-looking passageways and room connections, as the passages would be optimal in terms of length, but would require an algorithm with a much higher complexity (assuming no preprocessing), because of all of the repeated looping; furthermore, Lague requires some central room to be selected to ensure all components are eventually connected. I chose instead to implement an algorithm that would, in theory, run in linear time with respect to the number of rooms, disregarding any variations in the time it takes to complete Bresenham's algorithm.

In my implementation, I made use of a flag for each room that marks whether or not it has been "connected". I then picked the first two rooms in the list of rooms. Instead of finding the two closest edge coordinates between these two rooms, I pick any random edge coordinate from both of them, and then carve a passage between them using Bresenham's algorithm as mentioned above. What is important to note here is that this passage may pass through other rooms; as Bresenham's algorithm progresses, however, I can detect any other rooms I pass through by using a mapping from coordinates to room that I created earlier while I was acquiring our list of rooms through breadth-first search. Thus, if the passage being created passes through any other rooms, I can mark that room as connected purely by virtue of the passage going through it. Once this passage is done, I simply found the next unconnected room in the list of rooms (passing over any rooms that may have been connected from a previous passage creation), and repeated the process. In this way, I added at least one room to the connected network of rooms at a time, and possibly more; I also did not need to reconsider any already connected nodes, as I kept track of them accordingly.

Note that all of this passage creating and connecting consist of operations on the 2D array of integers before mesh generation: I simply needed to alter the layout. Because of how I structured the mesh creation from the layout, altering the layout properly would then lead to proper mesh generation without any need for changes there.

**Barriers and Keeping Track of Reachability**: The next item on the agenda was to add some sort of obstacle system to the game; after all, a game is quite boring if nothing attempts to block your progress through it! I wanted to add some method of obstacles or barriers throughout the levels, which would impede player progress until the barrier was disabled. I realized the most intuitive way to do this was to make use of the already-present information of room edge coordinates; even though I had created passages between and possibly through rooms as

---

7 MapGenerator.cs: CreatePassage
described above, I still had information about what the edges of rooms should have been before these passages were added. Furthermore, because the rooms were connected through these passages, the information I mentioned above about passages going through rooms and our ability to know which rooms those were gave me an intuitive notion of which rooms were reachable from other rooms. So, to list out some points:

- I could create barriers by simply creating wall meshes around the outlines of rooms. So, even though these rooms had been connected by carved-out passageways, I would block access to some rooms by wrapping the rooms in another layer of meshes.
- I could add some instructions in the passageway-creation algorithm to record which rooms were directly reachable from which other rooms.
- This gave me an intuitive method of keeping track which rooms were reachable from the player’s position, given that the user starts in some specific starting room on each level, by conducting another breadth-first search of reachable rooms.
- I could also keep track of which rooms the player could get to the edge of but not enter because of their barriers.

All of this information allowed me to create a system of spawning key blocks that would deactivate room barriers in locations that were reachable. This ensured that the game, at any point, can always be progressed through. Furthermore, because I could keep track of this information per level and I allow the player to move between levels freely, I could create conditions for progression across multiple levels; that is, the key block needed to deactivate a barrier on one level may be spawned at some point on a different level. All this required was an additional hierarchical level in organizing the reachability information.

Enemies: At this point, I decided to add an additional dimension to the game by introducing enemy entities, a points system, and an option for combat by firing a weapon from the first-person perspective. This required some basic UI to be added, code for keeping track of points, and code for bullet generation and collision detection, which I believe remains simple enough to not need further explanation. The more complex thing to be done was to add some simplistic AI to the enemy characters that would allow them to follow the player upon coming close enough, and navigating around walls-obstacles to do this.

In the interest of this, I utilized the A* Pathfinding Project by Aron Granberg, which is a Unity package that provides an implementation of the A* search algorithm for pathfinding purposes. This package allowed me to scan each of my levels (updating when necessary, such as a barrier being deactivated), producing graphs of obstacles and open space in rooms that are then used by the algorithm to find paths between enemy game objects and the player. Thus,

---

8 MapGenerator.cs: MakeSwitchReachable/MakeSwitchUnreachable, InitiateSwitches, SwitchHit
9 But, in case you’d like to know more, see: BulletController.cs; EnemyController.cs: OnCollisionEnter; PlayerController.cs: OnGui, ActionButton, etc.
10 https://arongranberg.com/astar/
enemies will seek paths to the player if the player comes close enough to them, and follow them.

**Textures/Sound/Graphical Effects:** Textures used included those for key blocks\textsuperscript{11}, for walls\textsuperscript{12}, for the crosshair\textsuperscript{13}, and for the ground mesh\textsuperscript{14}. Sound effects\textsuperscript{15} were used for walking, weapon firing, player and enemy elimination, teleportation between levels, and item pickup. A quick note: sound effects for enemy walk and elimination are actually spatially arranged. Thus, if one plays the game with headphone or a surround sound system, you can localize which direction an enemy might be approaching you from. Background music was composed by Peter B. Helland\textsuperscript{16}. Finally, I included some graphical image effects in an attempt to establish a sense of atmosphere; the camera utilizes a depth of field effect, where, if it raycasts in the forward direction and hits an object within 5 units, the focal length is set to just to the distance to the object, creating a bokeh effect in other planes of distance. Additional color correction, noise/grain filters and anti-aliasing effects were added to the camera object as well.

**Issues and Directions for Further Development**

- While a sense of atmosphere (despite the absurdity of the self-drawn cartoonish textures) seems to have been decently established, the game has quite a level of difficulty; there is not currently any method of scaling this difficulty.
- While the enemies/points were added to provide an additional dimension of interaction, they seem ultimately on the pointless side, if the objective of the game is to reach completion. Perhaps a timing component would be a more worthy future dimension to examine.
- Enemy entities, while for the most part adhering to the paths calculated by the A* search pathfinding algorithm, have a couple of issues from time to time with collisions. The enemies seem to have slightly overestimated collision detectors, and also disrupt each other's paths when clumped together, leading to strange standoffish situations where they become stuck in front of the player.
- Following from the earlier comment on difficulty: in the future, a map may be helpful.

\textsuperscript{11} https://www.key.com/kco/images/key_social_logo.png
\textsuperscript{12} https://t00.deviantart.net/GzHyHgTUnl-bG7-ybi62nVEstbcQ=fit-in/150x150/filters:no_upscale():origin/()pre00/1017/ith/pre/i/2005/120/5/leblue_floor_tiles_texture_by_deity37.png
\textsuperscript{13} https://static2.gamekit.com/i/650/450/upload/15/11/11/26/04/25/60425_-jpg.jpg
\textsuperscript{14} https://i.pinimg.com/originals/fe/51/1fe51d1f09864e3c1e0bcec8abe274e38.jpg
\textsuperscript{15} All from https://freesound.org/browse/tags/sound-effects/.
\textsuperscript{16} https://www.youtube.com/watch?v=itPtO4EaEXU