Mythic Labyrinth Game Development

Shawnee State University

Thomas Gilman

1st December, 2019

Developing a game throughout a semester is a lot of work. Deciding where to start, what theme the game should follow, the engine to use for the game and the many issues that come along with the engine being used. These steps along with the many challenges with the scope of the project, time management, and issues with the selected engine are just the beginnings to the development of the game Mythic Labyrinth.

Beginning development on the game, a theme had to be chosen including a name. I opted to choose a game revolving around dungeon exploration. The game would follow the premise of an adventurer, being the player, off trying to discover fame and fortune. Upon following a rumor of a labyrinth that has yet to be explored, the player ventures forth in order to make a name for themselves through exploring the labyrinth and returning with the riches that may lay within. Inspiration for this idea had come from similar games involving fantasy style R.P.G dungeon diving mechanics, such as Legend of Grimrock, Legend of Dungeon, and from Greek mythology such as the Minotaur in the labyrinth. For the game to follow this theme, I had decided to create a game revolving around a procedurally generated maze, in which every time the player started the game, the maze would be different from the last time played. Upon making this decision, a few different names were chosen to summarize the game, and I was torn between two names; Labyrinthus meaning Labyrinth in Latin, and Mythic Labyrinth. I chose Mythic Labyrinth as the name to give the impression of a fantasy-based maze and dungeon game, rather than a singular title summarizing the game as just a maze.

After deciding upon the theme and the name of the game, the engine would have to be chosen to build the game in. Due to my lack of experience with both Unreal Engine and Unity, I decided to go with Unity as I would be able to do all of my work in C#. While I understand Unreal provides better visuals with its shaders, I did not feel comfortable using Unreal due to time constraints of learning the engine, and the difficulties I have had in trying to program using its libraries in C++. Unreal would constantly refuse to show me, or even update the intellisense in Visual Studio, and the documentation and examples for programming using Unreals libraries were sparse. However, the parts of Unreal that I did like were the blueprints, which could have been convenient in the future for minor classes or objects. With Unity, I could program the game entirely in C#, which I am rather comfortable using, the intellisense works, and there is a wide variety of examples to get started.

Once the engine was decided on, I had to find an algorithm for creating 3D mazes. One such example that I happened upon that matched this case, was a tutorial on cave generation using both cellular automata and the marching squares algorithm named “Procedural Cave Generation” provided by Unitys Learn site. The tutorial led me through the process of creating meshes through setting vertices and edges based on the surrounding tiles and whether they were ground tiles, or wall tiles. The first part of the program was creating a 2D array of integers, following the cellular automata pattern of having two states, and a fixed grid size. The player could opt for either a randomly generated maze every time or provide a seed of their own to generate the exact same cave layout. A fill percentage would be provided as well, to set the threshold the random function would have to stay within to fill the maze. Using Unity’s random function, the seed is used to set a position within the 2D array to one, based on whether the random number generated is within the range provided. The borders are also set to a one state in order to encapsulate the room mesh being created. After the array has been filled with randomly placed ones, the array is then passed to a function to remove artifacts. The function removes these artifacts based on the cell’s neighbors; the cell is converted if majority of the neighbors are not of the same tile type. After which, the player also provides threshold values for both the room and wall sizes, where all the tiles of the type specified in the region are converted if the number of tiles are under the provided limit. Each room is also stored into a list, with each room being its own list of array coordinates. The room lists are then passed to a function to connect every room together. Before connecting the rooms, we set the first room in the list to be the main room. Once that is done, the function passes through the rooms splitting them into lists between rooms already connected to the main room, and those that are not. After splitting the rooms up, we pass through the rooms not connected to the main room and connect them based on the closest edge tile. Finding the closest tile is done by passing through both rooms edge tiles list; the list containing the surrounding wall tiles, and calculating the distance between each one, storing the smallest distance tile of each room as the destinations to create the passage from. Then once the shortest distance has been found, both tile coordinates are passed to a function inside their game object, where we create a path from the closest edge tile from the internal rooms. This is done by creating passages using a circle with the center point being along the line between both tiles; as we travel along the line from tileA to tileB, every tile within the radius of the center point will be converted to a ground tile. The rooms are then added to each other’s list of connections until every room is connected together through their connections, and is connected to the main room.

Once we have connected all the rooms together, we would then need to pass the 2D array of tiles to another class to generate the mesh for the walls. The mesh generation class creates a new mesh object from vertices and triangles that are created from the tile list we provide using the marching squares method. Using the map of tile coordinates from the previous step, we generate a grid of squares based on every tiles neighbors. These square tiles contain four nodes along with four control nodes, to determine the orientation of the square. The nodes are simply positions, whereas the control nodes inherit from the node class and are located within the corners of the square being created, and if they are active help determine how to create the triangles for the mesh. Upon creating the grid of squares, we then create the lists of triangles and vertices of the mesh object. We do this by creating triangles based on the number of control nodes active in each square. This is done with a switch statement for each possible case of active control nodes, adding the vertices to a list of vertex index positions that are associated with every triangle being created. Doing this allows us to generate custom mesh objects based on the tile map created, and thus also allowing us to assign custom collision boundaries that are associated with the triangles and vertices of the mesh. Once we have the triangles and vertices, we assign them to a new mesh object we create, which is a part of Unity’s provided libraries. We then create a mesh collider component that is to be associated with our newly created mesh object to link its vertices and triangles to the generate the collider from. All that we have done has created a 2D mesh, and in order to create a 3D version of the maze, we offset a couple of vertices by a height value multiplied by the vector3.up coordinates to raise the mesh walls, creating a 3D mesh object.

After creating the classes for generating our maze object, the next phase of designing the game was to increase the scale of the maze being generated. From this step moving forward I ran into issues. One issue being that if the 2D array was to large, this would cause a stack overflow due to the recursion of the mesh generation, when it creates the triangles and follows the edges between vertices. In order to get around this issue without having to re write the mesh generation code, I decided to create multiple maze objects, and treat them as rooms. Doing this would bypass the issue with recursion and allow for the creation of as many room objects as one would wish for and would also allow for the world to be procedurally generated as the player goes. So, if I choose to do so in the future, I could have the program generate a room whenever the player moves further and further within the world, as long as they were stitched together and connected. While this solved the issue of scaling the world to whatever size the player wants, another issue came up, being the way Unity handles its public variables.

When using Unity, the tool provides easy access to edit public variables that are provided within a script through Unity’s inspector window. The inspector window is great for easily tuning variables for creating the games maze, and world space. However, this is also an issue due to the fact the Inspector overrides any changes made within the script and takes precedence. This proved an issue, as I would set the variables to what I wanted in the parent script within the scene, but when a child script was created using the same variables upon instantiation, it would overwrite them as zeros removing the desired settings. One work around for this would be to put all my classes and functions within one entire script, but this would create another issue involving code management, organization, and increase the difficulty of reading over my code. Setting the variables to private however may solve the issue of the inspector overwriting them; it would also introduce issues of outside functions not being able to access the variables, and on top of which forcing me to open the script every time to edit them. In order to get around the issue of allowing other files to access the public variables, without them being overwritten by Unity’s inspector, I had to declare them as being public static, which hides them from the inspector and allows them to be accessed by other functions and classes. Along with doing this, I created a helper function that would read in an outside settings file with all of my variable settings I want set, allowing me easy access to these settings, and also allowing the use of a settings menu in the future, to edit these attributes. This was also the only way of setting these variables outside of opening the script they were in.

Having Unity read in files has issues of its own, as Unity is not allowed to read in from locations other than within the projects scope. For Unity to do this, the settings file had to be placed within Unity’s assets folder. Once this was done, every room could be generated randomly with the same variable settings, and from the seed provided or from the systems date generated into a hash code, allowing for a grid of rooms with their own collision boxes and layouts. The rooms would then be separated based on their width and depth multiplied by their location within the 2D array of room objects, allowing the rooms to neighbor one another. Now that both of those issues were solved, connecting the rooms together was the next step. This had to be done through creating passageways between the different rooms similarly to how each game object generated a passageway to connect its internal rooms, and it would have to be done before the associated mesh was generated. During this process, I had the program connect the rooms together in a way that it would create a perfect maze. A perfect maze is one where there are no loops, and every room is accessible from the starting point. This is very similar to how the rooms connected to one another when generating the mesh. For this I created a struct containing the room game object, including whether it was connected to the main room, and its list of neighbors that are unconnected, and an empty set of connected rooms. The list of rooms were then passed to a function where, while there was a room with possible neighbor connections, the loop would randomly select a room from the list as long as it had possible connections, and connect it to one of its neighbors, removing the neighbor from its neighbors list for both rooms, as well as setting themselves as connected to every other connected rooms connections and removing themselves from the other connections neighbor list if present; and if the room was the main room, it would set its connection to the main room being true. Upon making a connection to a neighboring room, both rooms would be passed a point along their border between them to make a passageway to. This would be done with the use of Unity’s send message function where a function could be called in another game object’s script along with a single variable passed to the function. The function being called would check through each of the internal rooms edge tiles for the closest possible point to the border point and create a passageway to that point. This would repeat for every room until every room has been connected together, with passageways connected between them. Once all the rooms were connected, each rooms game object would have its mesh generation function called using Unity’s send message function, creating the appearance of one giant maze object.

Now with the world being generated, a simple player object could be created, with just a box collider and basic movement, and if it has a rigid body, it would collide with the walls of the generated meshes. Due to a lack of time before the Shawnee Game Conference, I opted to add some basic gameplay functions, in order to show off the work I have done thus far. To start off, I created a basic ceiling and floor object from plane objects that would serve as place holders for the time being; this way the player would not fall out of the world. For the player to spawn in, a spawner game object was created that would be placed within the boundaries of the main room, serving as both the starting and exit point for the player. The player would not be allowed to win the game if they did not return with treasure or goal objects found throughout the maze. These objects would be placed similarly to how the spawn was placed, by scattering them throughout the boundaries of each room randomly, and by a set number of them. A finale goal object was also placed at the furthest room object to serve as an end goal to find. To create a challenge for the players, a timer was created that served as the players life and score. The timer would constantly run down as the player explored inside the maze and could only be increased by finding the game objects within the maze. Once the player found an object, they could then proceed to return to where they had started and leave in order to receive a score. However, if the player chose to, they could explore more to further increase their score through collecting more goal objects. But on the contrary, risk losing their way and running out of time. In order to help the player, find their way around the maze, I created a mini map that served as a top down view, which would follow along with the players location. Showing the players surroundings, and highlighting the game objects and starting point, so that the player could find them easier. This is done by creating highlight objects on each object, associated with a tag that could only be seen by the mini map’s camera. The mini map camera would then draw to a texture that is applied to an image attached to the player overlay panel. To help the player find their way back, and show where they have been, I added a trail effect to the player that could only be seen by the mini map camera, and would not dissipate; as well as miniature cube objects the player would leave behind in the game space. These cube objects served as visual markers for the player, that way they could find their way back out of the maze or identify where they have been. On the other hand, the more the player travels around the maze, the difficulty of finding their way back increases if they backtrack a lot.

Through adding these minor gameplay details, along with a win screen, lose screen, and menu screen, I was able to create a working demo for SGC. Further I attempted to optimize parts of my code to decrease the amount of time it would take to generate all the room meshes, as I wanted to reduce loading time. Creating many vertices and triangles each time tended to take a significant amount of time, depending on the number and scale of the rooms. To try and reduce the amount of time taken, I had attempted to parallelize Unity’s Instantiation function. This did not work due to Unity not being thread safe and would not work with assigning Instantiations as tasks. Through doing some research I found that the only methods that provided for parallelizing work, happened to be Unity’s Jobs System, this seems to only work for computational tasks for now, which does not help with creating game objects in parallel. Instead I converted for loops into parallel foreach loops in multiple places, especially where most of the work was being done. This helped reduce the time each Instantiation takes, allowing for larger rooms and a larger world space for the demo. Finally, to prepare the game demo, I created a game object for the goals using Maya, applied textures to the walls, floor, and ceiling using free textures and a free background sound effect, all provided from the Unity asset store.

During the Shawnee Game Conference, I received a lot of positive feedback and criticism of my game. A lot of people opted to play the game primarily from the mini map, due to the way the textures applied to the UV’s of the wall meshes, the texture was not applied properly making it difficult to look at the surroundings. The map also showed where the goal objects were, as well as the start, allowing the player to navigate through the maze just from using the map. Other critiques involved adding enemies, other objects to find within the maze, a proper tutorial, fixing the textures applied to the wall, balancing the time and score correlation, and the player controls. Upon learning these critiques of the game, I plan to add an option for player controls settings when I implement the settings menu, that way the player can bind their keys, and adjust their look sensitivity. Other than fixing the wall textures, I also plan to add more variation to the wall height and surroundings. In order to fix the mini map issue, I plan to set the map to show as a panel overlay rather than a mini map in a corner. Setting the map as a panel that pulls up upon a keypress would take away the use of playing from the mini map and could then be made into a larger map image showing where the player has been, and what the player has discovered. This would force the player to pay more attention to their surroundings rather than a map.

From the goals I have set out to accomplish, I feel as though I have met a large portion of the goals, I have set for myself and for the project. While the original timeline I had set, included implementing enemy AI and an inventory system, it was outside of the range of time it would take for me to complete along with staying caught up with all my other classes within a single semester. Throughout this semester, I have accomplished the world generation part of my game, including base gameplay functionality. With the way the world generates based on creating meshes; this opens many possibilities to improve upon the environment that is generated. The goals I plan to reach with furthering my game would be to add terrain generation using the same process for the walls and attempt to implement the diamond square algorithm for generating the floors and ceilings. I also plan to adjust the height of the walls being created, to give more depth to the maze, possibly allowing multiple levels. Another goal would be to add basic artificial intelligence to the game through random enemies that could spawn within the maze. The enemies could then affect the players health and score based on damage sustained from fighting and defeating them and their associated difficulty value. On top of these goals, I will need to add more objects to the game, this includes a player model, weapon models, possible foliage or dungeon objects, and enemy models. Finally, gameplay will need to be adjusted to push the player to explore and search the dungeon being created; this could be done through a story, or lore scattered throughout the dungeon.

Altogether I feel as though I have accomplished a lot in terms of building a solid foundation to my game. Starting from nothing, I have learned a lot about Unity, and generating mesh objects, the issues with Unity and bypassing them, along with better time management skills throughout this semester. Continuing forward, I am excited to expand the game Mythic Labyrinth, and implement greater goals throughout the second semester of developing the game.

Sources

Lague, Sebastion. “Procedural Cave Generation”. <https://learn.unity.com/project/procedural-cave-generation-tutorial/?tab=overview>

Olsen, Jacob. 31st October, 2004. “Realtime Procedural Terrain Generation”. <http://web.mit.edu/cesium/Public/terrain.pdf>

Losh, Steve. 27th June, 2016. “Terrain Generation with Diamond Square”. <http://stevelosh.com/blog/2016/06/diamond-square/>

24th July, 2019. “Diamond-square algorithm”. <https://en.wikipedia.org/wiki/Diamond-square_algorithm>

Celedon, Miguel. 30th June, 2017. “Exploring the game development world: Procedural terrain generation -- Theory”. <https://medium.com/@miguelceledon/exploring-the-game-development-world-procedural-terrain-generation-theory-849d6e54f3a1>

Pullen, Walter. 20th November, 2015. “Maze Classification”. <http://www.astrolog.org/labyrnth/algrithm.htm>

30th November, 2019. “Maze generation algorithm”. <https://en.wikipedia.org/wiki/Maze_generation_algorithm>

Neounix. 18th June, 2018. “How to Instantiate() GameObjects in a job?”. <https://forum.unity.com/threads/how-to-instantiate-gameobjects-in-a-job.536606/>

Editorial review. 27th March, 2018. “C# Job System”. <https://docs.unity3d.com/2018.3/Documentation/Manual/JobSystem.html>