Mythic Labyrinth Game Development Second Semester

Shawnee State University

Thomas Gilman

1st May, 2020

With the work that has already been laid out from the previous semester, having built the foundation to generate 3D mazes and a new world space every time you start a game. This semester's goal was to improve upon the foundation that was set for my game. The goals I had originally set for myself included adding terrain and ceiling mesh generation, adjusting the wall height, fixing wall material mapping, fixing spawning of objects in walls, creating multiple levels, enemy AI, fighting mechanics, more objects, foliage, and settings menu.

From the goals I had laid out for myself this semester, the first goal I had decided to tackle was the way that the wall materials were being mapped to their corresponding mesh objects. Due to the fact that I was creating every room mesh independently from one another, there was no way that I could create a custom UV that would work for this. The reason being, a custom UV only works for objects that are modeled and already created, and every mesh being created is done randomly, creating its own custom UV every time. The only exception is when a seed is used that UV could have already been created, but either way, using a custom UV is out of the question. Another factor that goes into setting a material to a new wall mesh is that the shapes and sizes are all different, with jagged harsh edges and transitions. Just having a standard material and applying it to the wall won't work as the tiling may not be enough, or it could be too much in some cases. Finally, using a standard material on a custom wall mesh, it will not look natural, as the material would just stretch across the edges in the same direction, and will look the same from every perspective.

The Solution to applying a material to a randomly generated mesh is to use a Triplanar Shader. Triplanar Shaders are shaders that take the position along with an object and split the way the object is perceived based on that position's normal axis. This means three separate Texture2D objects will need to be assigned to the material; a front and back image which is projected along the z-axis, a side image which is projected along the x-axis, and a top and bottom image which is projected along the y-axis. From the three different axes, the textures are blended together based on their normal position. The position is used to split up which images are blended together; thus there are three pairings in total; that being top and the bottom image is blended with the side image, top, and the bottom image is blended with the front and back image, and the front and back image is blended with the side image. What determines how much one image is blended with another, is based on the normal position of a given point on the mesh being drawn. The normal determines how much an image is displayed at a given angle, almost like an opacity slider ranging from zero being not present to one being fully visible. Using the normal allows the material to slowly blend textures together along as there are smooth transitions and no jagged edges along with the object, which sadly my walls lack.

Figuring out how to create a Triplanar Shader in Unity took a bit of time to figure out, as I have been using Unity's Lightweight Render Pipeline (LWRP) to provide better visuals to my game. In doing so, this has limited some of the tools I could use, as using Unity's LWRP limits your ability to create custom shaders. The shaders you can use are then limited to the ones provided alongside the LWRP or creating an unlit shader. This inhibits my ability to create my own custom shaders or even learning to do so, as Unity's shaders use purely a variant of HLSL, which is also a bit of a roadblock as we have mainly been taught the OpenGL language GLSL which does not work for this. This limits me to finding resources on if you can create a custom

PBR shader, which I could not seem to find any resources on, or using Unity's Shader Graphs. I inevitably decided on using the Shader Graph approach as there are resources and tutorials available on implementing your own Triplanar Shader and other custom effects using Unity's Shader Graphs.

Unity's Shader Graphs follow very closely to how Unreal has implemented its own Material Graph. This makes it rather easy to create a PBR shader graph to implement a Triplanar Shader. The first step is to create a Position node, getting the sample position on the mesh the material would be added to. Given the position, which is a three-axis vector, you must then split the vector and assign a paired axis to three separate UV vectors, which are two-axis vectors. This means one containing the Y and Z coordinates for the side image UV, X, and Z for the top and bottom image UV, and X and Y for the front and back image UV. Given the UV's, you can then divide them to create a smaller image, and multiply to perform tiling. Afterward, the UV's are used on their corresponding Texture 2D image sample, which outputs a four-axis vector containing the color. Once the color sample has been attained, you must then use the normal to control how much of the image is shown; this is done using a blend, where the normal controls the opacity of the blend color. Blending can be done using a base color, but in this case, the base color I have set is a clear color with the alpha of zero; this way, the output of the blend can then be added together with the other blends. Upon combining the blended colors together, this gives you either your base, specular, or normal color, given the textures you have used for the task. Finally, you create a material from the Triplanar Shader graph, assign the textures to be used, adjust tiling and divider variables, and apply the material to your mesh. Which in this case, I had to set it as the material to use for my mesh-filters to use when creating my custom room mesh objects. Upon the creation of the Triplanar Shader, when the walls have been created, all but one of the textures was displayed correctly on the walls. The single texture that was not displayed was the side angle texture along the x-axis, in which it was rotated sideways by 90 degrees. To fix this, I could figure out how to transform the UV coordinates in the shader graph. Instead, I just made a separate image for the side view, which was also rotated sideways by 90 degrees; this fixed the issue. Another issue also followed, in which was the jagged edges of the walls. Because the walls had harsh transitions and edges, this also caused harsh transitions with the texture blending. To approach this issue, I had attempted to create a function, that during the mesh creation process, after the mesh UV's, vertices, and triangles had been created, would subdivide the mesh creating smoother transitions to the walls and their corresponding edges. This was unsuccessful as the way I was creating my meshes had caused vertices to connect at other ends of the rooms, causing the mesh to become a jumbled mess of triangles. In this case, I have set aside the approach for later when I have more time to tweak and fix it.

Proceeding to the next issue, in which case was the generation of a floor and ceiling mesh. Using a single plane for a floor may be efficient in reducing the number of triangles in the scene; however, this does not allow for dynamic floors or different height levels to the rooms. Tackling this problem proved to be reasonably easy, as the setup was already there. With how I am creating the wall meshes, I used the exact same approach to create the floors and ceiling. The way my wall meshes are created uses the marching square approach to do the entire setup, in which case a map is passed to the mesh creation process. Inside the map is passed coordinates to where there are and are not walls. For the creation of the walls, a value of one is used to define a wall position, and a zero is used to define a flat empty space. To create the floor meshes, I instead used the zero value to create the mesh, in which case stitched the floor up to where it would match up exactly to the wall's vertices and bottom edges. The same process could be used to create the ceiling, but rather instead, a different height is given, along with inverting the y-direction the mesh would be facing; otherwise, you would see right through the backside of the mesh.

Making the changes that I have to the walls and the creation of the map had significantly improved the visual experience of the game. The walls now have better depth as there are proper textures applied to them; the floor and ceilings are present and fit to the edges of the walls; overall creating a much better experience. The only downside was getting the lighting right, as my game is entirely in a maze and dungeon-like environment. One solution I had tried was taking the light, which is a directional light, and face it 90 degrees upwards, causing everything below it, that being the maze to be extremely dark. I also gave the player a spotlight and put it to the right of the player where a hand could be. Using a flashlight for the source of light and finding your way around the maze for the time being, greatly improved the feel of the game. I also tried adding a little bit of fog to the scene, creating a fading effect to objects further away and reducing the view distance of the player. Making these changes helped with the feel of the game, but it still was a bit unrealistic.

To better improve upon the realism effect of the lighting, as well as improving the players' perspective, I have looked into moving my project into using Unity's High Definition Render Pipeline (HDRP), which is the direction the LWRP has been going throughout its development. The benefit of using the HDRP over the LWRP is that everything is Physically

Based. Meaning it uses real-world units such as Lumens or Lux, for how much light is being projected from a point. However, some big downsides to swapping to HDRP, is that it is still fairly new. Leading to many openings for bugs, visual issues, or problems that are either currently being fixed, or issues where there is little to no community resources to help debug or fix the issue. Another issue is that everything used in the LWRP is unusable in an HDRP environment; meaning everything needs to be swapped over; this includes materials and adjusting a majority of the settings and project preferences. Finally, the last issue is that if your current project is in an older version of Unity, you must port it over to a newer version that uses and supports the HDRP, which appears to be Unity version 2019.3. Upon porting from an older workspace, some functions could be depreciated, or things can break, which is painful and tedious to deal with.

Luckily upon swapping over to a newer version of Unity, my project did not break. This made it very easy to go to Unity's package manager and download the HDRP tools. Unity has been so kind as to include helpful tools in the HDRP installation wizard which helps resolve any issues, conflicts, and give options to port all materials and project objects over to using the HDRP. Upon doing so, everything related to using the LWRP must be deleted and removed from the project. Unity also provides a very helpful demo in its resources page for swapping to the HDRP, where it explains how to include the HDRP scene root, adjust fog and light levels, and create settings overrides. Swapping over to the HDRP also means that you can only use HDRP shaders. This means the Triplanar Shader I had created needed to be remade in an HDRP Shader Graph, which is as easy as copying and pasting, and just reincluding variables luckily enough. By changing to the HDRP, this has allowed me to adjust the brightness of the scene efficiently, apply visual effects quickly such as bloom, depth of field, apply a vignette, add much more realistic fog, and better shadows. By doing so, the visual experience of the game has increased even more from what it was initially.

Upon swapping to the new render pipeline, I also needed to figure out a way to better display the players' trail, objects in the scene, and where the player starts and must get back to win. To fix this, I added a light to the start minimap object as well as changed the spotlight for the goal objects, to a point light that sits inside of it. This causes the goal object to project intricate shadows onto the walls as it spins, which seemed pretty interesting; this also allows them to be visible from the minimap. With the darkening of the minimap, this also helped fix the issue of players playing directly from the minimap. Now, they can't see the map layout too well other than what's around the player, the start icon if within the view of the minimap camera, or the goal objects. However, this also causes the player's trail to be harder to view from the minimap, which may need to be fixed later. Altogether, the move to Unity's HDRP has been rather lovely and has improved the visuals of the game.

Some of these issues I have run into while developing my game have made it difficult, and sometimes make me feel like swapping to a different engine at times. There are many things that I have enjoyed about Unity through using it. It is effortless to add new objects to a scene, it's extremely user friendly with setup, and scripting in C# is an enjoyable experience. Unity's visuals are also enjoyable to look at, and being able to create materials quickly and adjust them in shader graphs is visually pleasing. The options for using different pipelines or project settings is very fleshed out and provides a great variety of tools. All of these traits have made the development cycle a pleasant experience. However, some pitfalls I have with this include being limited to how I can optimize my code, not being able to create custom shaders easily and efficiently when wanting to use a pipeline. These issues have caused some difficulties with how I have wanted to set up my project. All of the work I have done has been on the initial setup of the game environment, prior to when the game has started, which means all of the world setup, mesh creation, object movement, all of which are handled by the main thread. There are places as to where I have been able to parallelize some of the work, but this remains limited to loops. I'm sure doing more research into using Unity's compute shaders, I can modify my working code to use the GPU to create the mesh objects rather than the CPU, but I'm not guaranteed to find a solution to this using Unity. Unity is not thread-safe in the slightest, the closest equivalent is the Jobs system, but this is only limited to computational work, and in no way allows for parallelizing game objects. This in no way shape or form is a bad thing for smaller projects, but for older computers or larger projects, this poses a bit of an issue, creating a significant limitation. Depending on the size of the world, this will increase the loading period on startup, which is an unwanted outcome. Some solutions to this would be to dynamically create the world which I intend to do, as the player approaches the border of another room, you can then load and draw that said room. This can still pose issues as it can cause hiccups with the game, stopping to create a new mesh and load in the environment. There are many approaches to fix this, and I will need to explore them to optimize my game better if I continue using Unity, which I still intend to do at this point.

Some other alternatives to using Unity to develop my game further would be to create my own game engine, in which I would most likely create it using OpenGL and SDL, as I am rather comfortable with using GLSL and c++. The downsides to taking this approach would be that I have to create my own game engine, which in itself is a task. But over the course of this semester, I have found that I enjoy working and creating features to a game engine, as I have discovered throughout the Realtime Two course. By creating my own game engine, I could end up learning more about engine development, customizing the engine for my purposes, and use it for any project without any possible licensing issues as long as any resources I use are open-source. This, however, means my engine will only be as good as how I program and set it up. Meaning it will only be as efficient as I make it. But in doing so, I can make sure that it can perform concurrent tasks, I know for a fact, I can create and modify the world and room mesh using a Compute Shader. I can design and decide on what shaders I can use and implement shading techniques easily, such as tessellation. This is something I feel I will explore in the future, as another side project, and something I can use to make my game in if I choose to port it in that direction.

Throughout this semester, I have accomplished a few of the tasks I have set for myself. But not as many as the previous semester in which I set the current foundation of my game. I have, however, improved the visuals and feel of my game, as well as fixed a few of the issues I have faced prior to this semester. The biggest issues faced with developing my game has been time management and being constantly busy as all of my classes have had either multiple projects, or had projects that took up the entire semester that built upon every lab. I do plan to continue development and improve upon my game. My biggest improvement that I intend to make would have to be creating any and all models in another modeling software such as Blender. This is because I cannot use any models that I have created in Maya as it requires I have a license to use it in my game, and It would be rather hard to justify using Maya with its cost as a single developer. Other improvements would be to explore dynamic mesh creation further and add more improvements to my current room mesh creator, including the last goal of allowing for changing height to the rooms, allowing lower different levels to the game. To better improve the playability of my game, I need to add enemy AI, items, and adjust the way the game plays overall.

I am very thankful to have been able to explore and start this process of creating my own game. I would not have been able to do this as easily without the help and support of my colleagues, professors, and the time I have spent while at Shawnee State University. Being able to learn the skills I have and apply them to what I enjoy is very much a blessing. I'm excited to continue building and developing my game to eventually release it, and work on future projects.

Sources

"Shaders in LWRP". (2020, May 1st). https://docs.unity3d.com/Packages/com.unity.render-pipelines.lightweight@5.3/manual/shaders-in-lwrp.html

"Shading language use in Unity". (2020, May 1st). https://docs.unity3d.com/Manual/SL-ShadingLanguage.html

"Getting started with the High Definition Render Pipeline". (2020, May 1st). https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@7.2/manual/Getting-started-with-HDRP.html

"Converting a Project from the Built-in Renderer to the High Definition Render Pipeline". (2020, May 1st). https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@7.2/manual/Upgrading-To-HDRP.html

Flick, Jasper. "Triplanar Mapping". (2020, May 1st). https://catlikecoding.com/unity/tutorials/advanced-rendering/triplanar-mapping/

"Unity Shader Graph: Triplanar Shader Tutorial". (2019, January 16th). https://www.youtube.com/watch?v=yuiQo54Baos

Chaiker. "HDRP Custom shaders". (2018, March 9th). https://forum.unity.com/threads/hdrp-custom-shaders.521102/

Bunny83. “MeshHelper”. (2019, March 22).

http://wiki.unity3d.com/index.php?title=MeshHelper&oldid=20389