A Chunk Streaming System For An Open World Game

Charlie Evans
18009251

University of the West of England

April 10, 2022

An open-world streaming system for loading and unloading chunks from disk at run-time based on distance. A simple LOD system was implemented with the trees, and a navigation mesh was generated on a per-chunk basis; with JsonUtility utilised to save chunk data to file. Chunk loading alone was found to take up 80 percent of the CPU process.

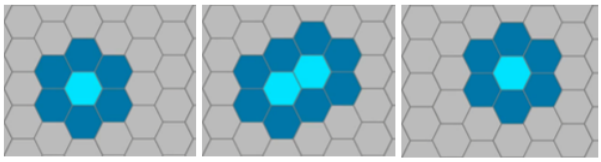

Figure 1: *Hexagonal Chunk Loading In Sunset Overdrive*

1 Introduction

Open-world games boast huge worlds for the player to explore, generally offering a non-linear approach so objectives can be undertaken as the player sees fit. For the game to be able to handle such large and populated worlds a streaming system is needed to ensure that only the entities located around the player are streamed in at run-time. Typically the world is divided up into chunks and only the chunks close to the player are loaded and chunks that are too far are unloaded from memory. By doing this the game can achieve more streamlined performance and provide the player with the illusion of a living and breathing world around them.

2 Related Work

A GDC talk from Ruskin, 2015 discusses the streaming technology used in the game Sunset Overdrive; an open-world, action-adventure game produced by Insomniac Games. The world was divided up into hexagonal chunks using hexagonal tessellation as shown in figure 1.

Only the hex the player is standing on, as well the hexes connected to it, are loaded. When the player moves to a adjacent hex, the adjacent hexes of the new hex are loaded, and the adjacent ones before it are unloaded. As the player can fall faster than the world is capable of being streamed the chunks are streamed on a side to side basis. Their attempt at a streaming system was far from a reliable solution made evident by the reveal that safety platforms were hidden inside the buildings at all possible player spawn locations just in case a building weren't to load in in time.

In a thesis written by Juan Pérez, 2016 the strategy employed for writing chunk data to file was to have a XML file for each cell rather than having one big file representing all the cells. Each file contains information on any entities in that cell such as their positions, texture, size and other such data. This approach is the inspiration for this project whereby each chunk will have it's own data file named by it's row and column position in the terrain. The file will contain data on objects found in the chunk including it's mesh, material, and position.

3 Method

3.1 Mesh Generation

Creating the terrain involved passing in a series of serialised parameters such as the height map, material, and the width and height of the terrain before each chunk mesh can be generated. A chunk is a segment of the terrain which makes loading and unloading of assets more manageable. A list of vertices was made and iterated through which are offset by the height values found in each pixel of the heightmap to produce the terrain mesh from the heightmap. This process was streamlined through the use of a custom editor tool.

3.1.1 Editor Tools

Figure2 shows the simple editor tool that was made for generating and chunking terrain meshes. A terrain can be generated after passing in the width and height, chunk size, a heightmap and the terrain material.

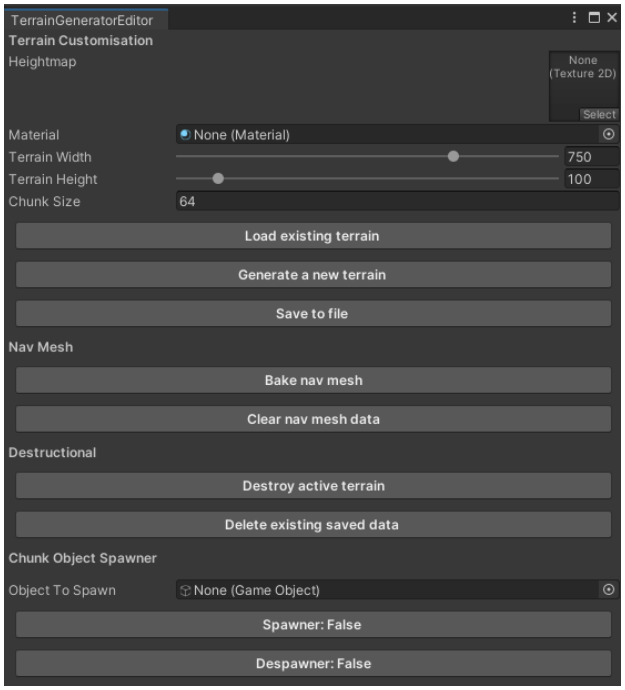

Figure 2: *The terrain editor tool.*

Once a terrain has been generated into the scene the terrain can then be saved to file with the save button. A tool for spawning objects is also included that uses ray casting to instantiate a chosen prefab onto the terrain upon a mouse click which is then assigned to the chunk that was clicked. Upon saving, the chunk objects will also be saved. Finally, there is also an option for baking a navigation mesh that the zombie ai uses.

Although not implemented within this project ray casting could also be used to procedurally generate trees at run-time on the terrain rather that methodically spawning or placing them by hand.

3.2 Saving To File

JsonUtility was used in conjunction with the File class to provide a simple way of saving data to disk. The functionality for saving data is handled with a 'SaveManager' script. A 'ChunkData' class was made that modelled the chunk data that was needed for the json file, with public properties such as the name, position, mesh and material of the chunk as well the chunk objects belonging to that chunk. Figure 3 shows the chunk json data.

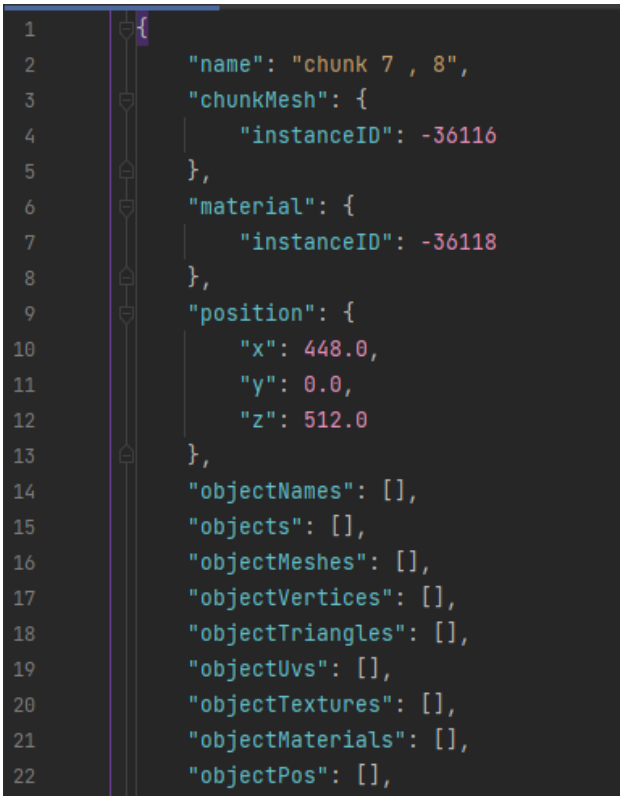

Figure 3: *Example format of a chunk json file.*

```
ChunkData chunkData = JsonUtility.
    FromJson<ChunkData>(File.
    ReadAllText(path));
```

The code snippet above shows the model being passed as a template type to the static method 'FromJson', provided by JsonUtility, to create an object from its Json representation which is being read from file.

```
jsonFile = JsonUtility.ToJson(
    chunkData, true);

 string path = Application.dataPath
    + "/SaveData/ChunkData/" + chunk.
    name + ".json";
 File.WriteAllText(path, jsonFile);
```

In this next code snippet the 'chunkData' object is passed as an argument to the ToJson method where it is converted to json format; with the true boolean there to format the json representation to make it more

readable. This json data is stored in the 'jsonFile' string which is passed as a parameter to the WriteAllText method alongside a file path in order to write the data to disk.

A separate data structure was modelled for the npc data as the chunk data files were beginning to become quite large. By abstracting the npc data to it's own file it made the data much easier to work with and locate. This model contained similar properties to the chunk model, however it also contained information on any quest data that a npc may have should that npc have a quest. Figure 4 shows the format of this npc data as it is represented in json.

NPC_Father(Clone)5.json

```
{
    "npcName": "NPC_Father(Clone)",
    "position": {
        "x": 297.043701171875,
        "y": 45.09804153442383,
        "z": 397.41778564453127
    },
    "questName": "Zombie Kill Quest 4",
    "rewardPoints": 600,
    "zombiesKilled": 0,
    "questType": 3,
    "questStatus": 0
}
```

Figure 4: *Example format of a npc json file.*

3.3 Loading and Unloading of Chunks

After a chunk's data has been gathered and written to file all of it's components besides it's transform component are destroyed when the chunk is unloaded; this also includes any scripts that the chunk may have and the chunk's navigation mesh data.

```
Destroy(obj: chunk.GetComponent<MeshRenderer>());
Destroy(obj: chunk.GetComponent<MeshFilter>());
Destroy(obj: chunk.GetComponent<MeshCollider>());
```

Figure 5: *Components being destroyed upon unloading.*

This leaves only an empty transform with a reference to the chunk that used to be there. When the chunk is loaded back, all of the components it used to possess are added back and the data relevant to that chunk is read back from file and applied accordingly.

Deciding which chunks should load and unload involved calculating the distance between each chunk and the player as demonstrated in figure 6. Should the distance exceed a specified distance and the chunk is loaded then the chunk is unloaded from memory. If the distance is less than a specified threshold and the chunk is currently unloaded then the chunk is loaded in memory. A layer of fog was also added so that the distance at which chunks could be unloaded could be decreased.

```
// If distance between player and center of chunk is more than maxViewDistance AND chunk is loaded, unload it
if (Vector3.Distance(a: playerPos, b: chunkCenterPos) > maxChunkLoadDistance)
{
    if (chunk.GetComponent<Chunk>().IsLoaded())
    {
        SaveManager.UnloadChunk(chunk);
    }
}

// If distance between player and center of chunk is less than maxViewDistance AND chunk is unloaded, load it
if (Vector3.Distance(a: playerPos, b: chunkCenterPos) < maxChunkLoadDistance)
{
    if (!chunk.GetComponent<Chunk>().IsLoaded())
    {
        SaveManager.LoadChunk(chunk);
    }
}
```

Figure 6: *Chunk Loading Script.*

3.3.1 Chunk objects

Chunks maintain a list of chunk objects such as trees, bushes or houses. As a chunk is loaded or unloaded from disk it's chunk objects are iterated through as well, destroying or adding components for each chunk object listed.

Npcs are motionless ai that can contain a quest component as seen in figure 7 that holds data on the quest name, quest type, reward points, stats and completion status.

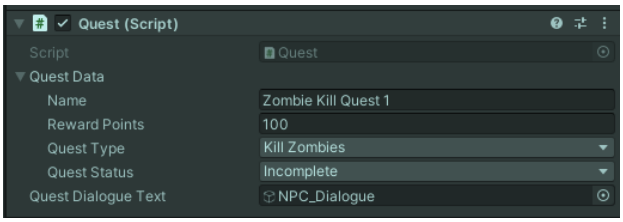

Figure 7: *The quest component attached to each npc*

When a npc is unloaded, the state of the quest data for a npc at the point at which the chunk was unloaded is written to file and then loaded in from disk again when that chunk is re-loaded.

Zombies are spawned through enemy spawners which are tied to chunks, however the zombies themselves are dynamic objects in that they do not belong to any one chunk. Because of this they can path-find through chunks even if the chunk they spawned on has been unloaded. When moving to a new chunk they are removed from their previous chunk and added to their new chunks list of chunk objects.

3.3.2 LOD

A simple LOD system was implemented for the trees which involves swapping out the tree model with less detailed ones with fewer vertices.

Figure 8: *Tree LODS of varying levels of detail*

Each tree has a custom LOD script which takes three different level of detail meshes as serialised parameters. The trees will cycle through these simpler meshes depending on the how far away the player is before fully unloading when the chunk unloads. Figure 9 shows the LOD component.

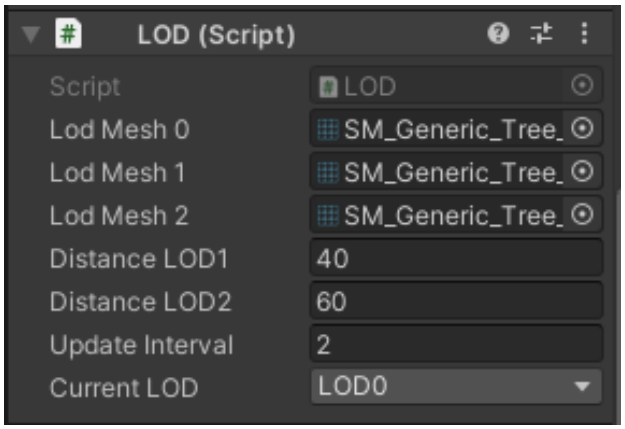

Figure 9: *The LOD component attached to each tree*

3.3.3 Navigation Mesh

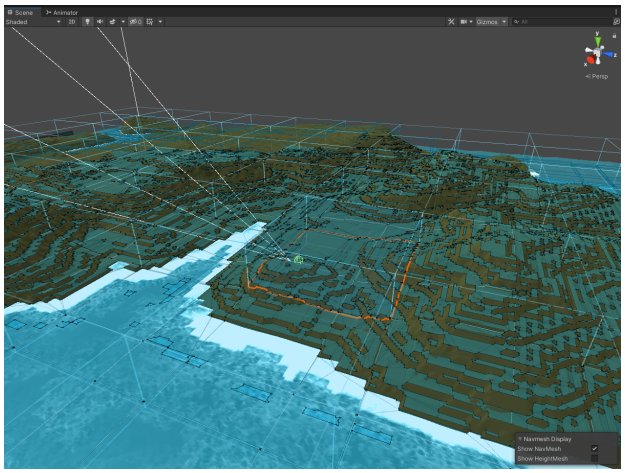

Figure 10: *The terrain nav mesh*

Rather than generating one big navigation mesh a small optimisation was made in which each chunk was assigned a navigation mesh surface component. These components then baked a navigation mesh for each individual chunk when loaded and deleted the navigation mesh data upon the chunk unloading as the navigation mesh isn't needed to be baked in areas where the player isn't.

4 Evaluation

The chunk loading as it is currently loops through each chunk and performs a distance calculation. A more optimal solution would be to only check and load chunks that are adjacent to the player's chunk, such as how Ruskin, 2015 discusses in his GDC talk on Sunset Overdrive's streaming technology.

Furthermore, it was unnecessary to check the player's position every frame in update. By using the 'InvokeRepeating' method, the distance check method could instead be called at select intervals.

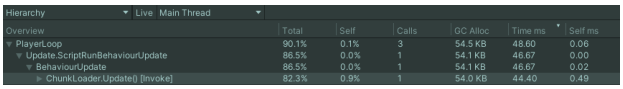

Figure 11: *Profiler results of chunk loading in the update method.*

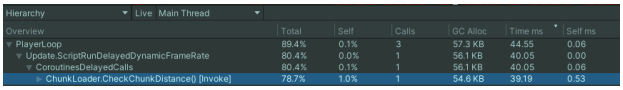

Figure 12: *Profiler results of chunk loading with the 'InvokeRepeating' method*

Figure 11 and figure 12 show the difference in performance between both methods. The second method appeared to have performed 5.21 milliseconds faster every frame than the former method. Interestingly, around 80 percent of CPU time is spent on performing this function call alone, regardless of the method used. This makes sense as disk reading and writing is a performance heavy task as indicated in figure 13 below, where spikes and frame drops occur when disk reading and writing operations happen.

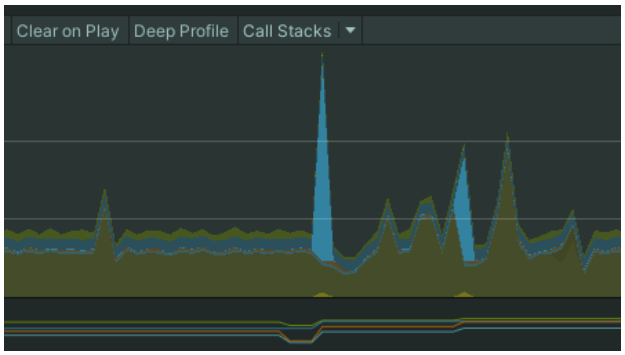

Figure 13: *Profiler 2.1.*

When loading a chunk the chunk data for that file must be read from disk which can result in a slight but noticeable bottleneck which is perceived as a freeze for a few frames by the player. This is not optimal in the long run especially if more chunk objects were to be added and more data is required to be read from disk. Juan Pérez, 2016 discusses a method of mitigating this by utilising another thread. This new thread would handle any disk reading operations and allow the main thread to focus on more important tasks such as rendering.

5 Conclusion

Overall the project has been a success. A open-world streaming system has been built that is showcased within a simple game. A working LOD system is present that is utilised by the trees. However for optimal performance the disk saving process should be handled on another thread. The save system works well but became quite messy by the end of the project as more functionality was added, so there is still room for abstraction.

Figure 14: *The final product.*

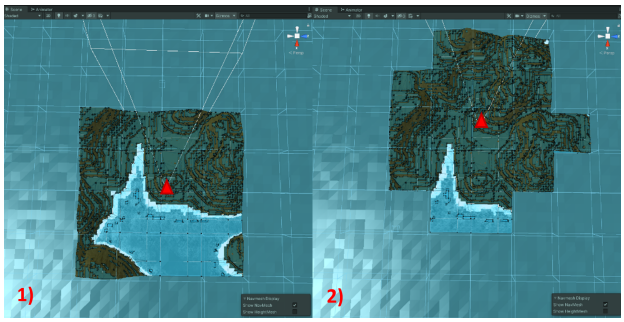

Figure 15: *Chunks being streamed in and unloaded from disk.*

6 Appendix

Video Log 1: `https://youtu.be/GOreetdpaUE`
Video Log 2: `https://youtu.be/kx1tO-_tkOU`
Video Log 3: `https://youtu.be/axXEuLii1Po`
Video Log 4: `https://youtu.be/ix3yvOgGnmU`
Video Log 5: `https://youtu.be/fO-r7f2Hunw`

Bibliography

Juan Pérez, Alejandro (2016). “Open World Streaming:Automatic memory management in open world games without loading screens”. PhD thesis. Universitat Politècnica de València.

Ruskin, Elan (2015). “Streaming in Sunset Overdrive’s Open World”. In.