Advanced Technologies: Open World Streaming

Alexander Hillman
19021645

University of the West of England

January 9, 2023

This report will cover the process of creating a system within Unity that can handle the world streaming of an open world computer game. The system combines mesh generation and the use of Json files to create an open world game where the world is split into chunks which are saved and then loaded and unloaded based off of distance from the player. Some objects in the world are also saved and loaded from files, this is handled by the chunk that the object belongs to. The result is a open world where only the area around the player is loaded, thus loading other objects such as non-playable characters and world objects that inhabit this world. The system performs well when run and meets the specification with it being a base for a full fledged game.

1 Introduction

The aim for this project is to create a system for streaming in an open world game, it should dynamically load and unload chunks of the world and their meshes as the player moves around. The system should also allow non-playable characters to be loaded and unloaded in the same manner so that they only exist and execute behaviours when in proximity to the player. The stretch goal for this task is to then build upon this system to create a fully fledged open world game.

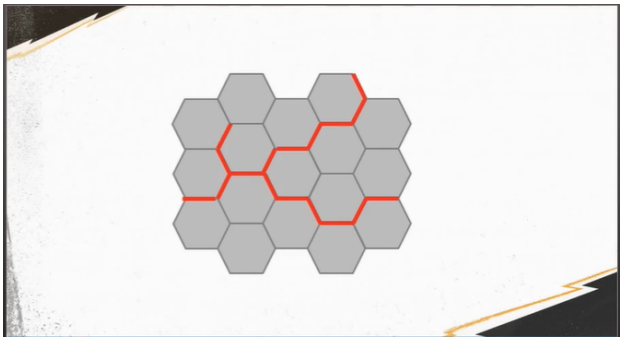

Figure 1: *A map created out of hexagons and the sigh-lines of players moving around the world*

2 Related Work

2.1 Sunset Overdrive

One notable piece of work related to this task is Sunset Overdrive (Insomniac Games, 2014), a third person action-adventure game that focuses on traversing a large fictional metropolis called Sunset City. Elan Ruskin describes how Insomniac Games streamed in their open world level. They began by dividing the level into regions of repeating hexagons to improve performance, this was also used to hide long sightlines. Since most streets ended with buildings, players were less likely to see areas that were either unloaded or at a lower level of detail since geometry obstructed the view (figure: 1).

Each gameplay zone can contain enemies, item spawners, interactables, anchors for dynamic encounters. These are loaded if the conditions are correct when the region is loaded so that some objects that are constantly needed such as intractables are only around

```
Hex_108_gameplay.zone:
{
 "SceneNodes":
  {
    "0x801031e346e3149b":
    {
      "AssetPath":
        "actors/AmmoCrate/AmmoCrate_Random.actor",
      "AssetType": "kModel",
      "Id": "0x801031e346e3149b",
      "LocalTransform":
      {
        "EulerRotation":
        { "Y": 90 },
        "Position":
        {
          "X": -4.92114655e+02,
          "Y": 1.60002728e+01,
          "Z": -2.00374374e+01
        }
      },
      "Name": "AmmoCrate_Random"
    }, ...
```

Figure 2: *Save file for a region of the world, detailing the actors within it*

```
private void CreateShape(int offsetX, int offsetZ, int chunkSize)
{
    vertices = new Vector3[(chunkSize + 1) * (chunkSize + 1)];
    colours = new Color[vertices.Length];

    for (int i = 0, z = 0; z <= chunkSize; z++)
    {
        for (int x = 0; x <= chunkSize; x++)
        {
            vertices[i] = new Vector3(x, heightMap.GetPixel(x + offsetX, z + offsetZ).r * scale, z);

            float height = vertices[i].y;

            if (height > maxTerrainHeight)
            {
                maxTerrainHeight = height;
            }
            if(height < minTerrainHeight)
            {
                minTerrainHeight = height;
            }

            height = Mathf.InverseLerp(minTerrainHeight, maxTerrainHeight, height);

            colours[i] = gradient.Evaluate(height);

            i++;
        }
    }

    triangles          = new int[chunkSize * chunkSize * 6];
    int vertexIndex    = 0;
    int trianglesIndex = 0;

    for (int z = 0; z < chunkSize; z++)
    {
        for (int x = 0; x < chunkSize; x++)
        {

            triangles[trianglesIndex + 0] = vertexIndex + 0;
            triangles[trianglesIndex + 1] = vertexIndex + chunkSize + 1;
            triangles[trianglesIndex + 2] = vertexIndex + 1;
            triangles[trianglesIndex + 3] = vertexIndex + 1;
            triangles[trianglesIndex + 4] = vertexIndex + chunkSize + 1;
            triangles[trianglesIndex + 5] = vertexIndex + chunkSize + 2;

            vertexIndex++;
            trianglesIndex += 6;
        }
        vertexIndex++;
    }
}
```

Figure 3: *Function to create the vertices and triangle list for each of the chunk meshes*

when the region they are in is needed and encounters can be placed in under certain circumstances.
Scene objects that get added to zones are represented as dictionaries of keys and values. These values can represent object data such as its unique identification value, its asset path or is position. These can theoretically be recreated when their zone is loaded by simply reading each from the data file and applying those values to its components.(Ruskin, 2015)

Regions are loaded based off of player position. The region that they player is currently occupying is loaded along with the chunks that surround it. When the player moves into a new region the regions that are now adjacent become loaded and the ones that are no longer adjacent are unloaded (figure: 2).
Level of detail was used with geometry objects so that objects that were distance were rendered with fewer vertices. These low-resolution hexes were always loaded but could have their level of detail swapped out when needed to ensure that the player could always see a skyline without it being to taxing of memory (Ruskin, 2015).

3 Method

3.1 Generating Mesh Data from a Heightmap

A heightmap is placed into Unity as a Texture2D and passed into a script to generate the mesh data for the world's terrain.
The designer can then set the size of the chunks that the map will be divided into, this means that a map can range from having lots of smaller divisions or less divisions that are larger. For this, a heightmap with a resolution of 1024 x 1024 pixels has been used that is then divided into chunks of size 32 x 32 to produce 1024 individual game object chunks (image required). The function loops through the number of chunks to create an empty gameobject for each and positions it. The script then creates the shape of the chunk, each vertex is positioned with an x and z value relative to the current heightmap pixel's location and a height determined by the rgb value of the pixel. Lighter coloured pixels generate vertices with a greater height compared to pixels with darker rgb values. The mesh generator colours each chunk based off of its vertex data. A gradient can be set within the inspector and when the generator assigns vertices it keeps a track of the minimum and maximum height vertex. Then a function loops through each vertex, normalises the height value between the minimum and maximum values and then assigns a vertex colour based on its height value evaluated on the gradient. This results in vertices of different heights being coloured differently. The chunk object has a save system script attached to it and then calls functions within the script to apply the mesh and material data to the object and to also save this data to a Json file. The triangle list is then created from the generated vertices. The mesh is cleared, has its vertices and triangles set and has its normals and bounds recalculated. Finally, each gameobject is parented to an empty gameobject to act as a container for the entire map.

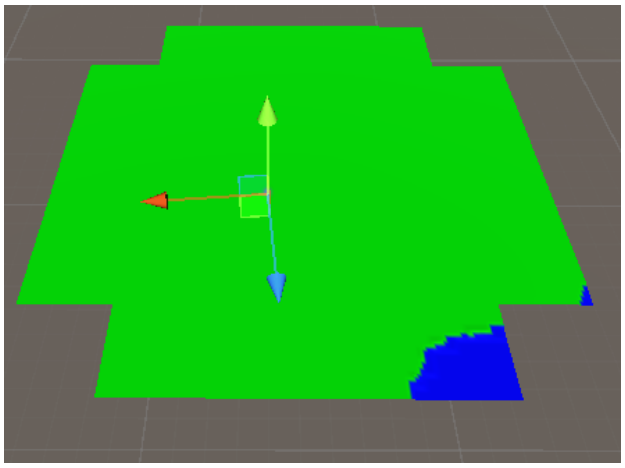

Figure 4: *Chunks loading in a radius around the player*

3.2 Loading and Unloading World Chunks

Since each of the chunk objects has a save script attached to it, it is able to control its own loading and unloading. This script writes the mesh and material data of a chunk to a Json file to save it externally, since this can be done in the editor there is no need for saving to be executed at run time, thus increasing performance.

The player has a draw distance variable to handle the distance at which chunks load and unload which is set to 100 by default, when the player gets too far away a chunk, it unloads itself by destroying its meshFilter, renderer and collider components. When the player gets close enough to an inactive chunk, it loads itself back in by reading from the saved Json file, creating a new structure with that data, re-adding the mesh components and then applying the loaded mesh and material from the structure to these components (figure: 4).

3.3 Loading and Unloading World Objects

The loading and unloading of world objects is handled by the chunk that each object is located within. The mesh generator has the ability to run a function from within the editor that loops through a list of all of the chunks. It then conducts an overlap sphere check on each to detect colliders within the vicinity of the chunk, the function then checks the detected object's position to ensure that it is within the correct chunk instead of being added to an incorrect one. Once these checks have been completed the function gets the save system script attached to the chunk and then adds the object to a list of objects inside of it. Once the objects are all added to a chunk the objects are then saved, done in the same way as saving chunks, a new structure is created to store the objects name, mesh and material data as well as its scale, position and rotation. The structure

Figure 5: *CPU usage, focusing on the chunk save system*

is saved to a Json file with a file path generated from the object's instance ID and the data path is added to a list. This means that every item saved should have a unique save file. One downside to the implementation carried out is the duplication of data if multiple objects have the same mesh data because then the same mesh data will be saved multiple times.
When a chunk is loaded it loops trough each object in its list of objects. It reads in the data path from a list, creates a new structure from the related Json file and instantiates a new game object. The object then has a mesh filter, renderer and collider added and set from the file alongside the object's scale, position and rotation. When the chunk is unloaded, the object is destroyed. This results in only the gameobjects that the player immediately requires being active for those chunks that are loaded.

3.4 Non-playable characters

NPCs are handled in a similar manner to world objects, each NPC converts it's position relative to the chunk index it is currently in and then enables or disables its body, colliders and attached canvas according to whether the chunk is loaded or not. NPCs also come with behaviours that are active depending on the status of the world chunks. These behaviours are controlled by a state machine, some custom scripting and Unity's NavMesh system for instance Some are classed as enemies, these NPCs can be idle, they can patrol their designated chunk or they can chase after and flee from the player.

Some NPCs can also provide quests to the player, types of quests can include reach quests where the user has to get to a certain location or object, kill quests require the player to kill designated enemies and collect quests require the player to pick up certain objects.

4 Evaluation

The game runs between 40 to 60 frames per second which fluctuates as the player moves around. As the chunks around the player are being loaded and unloaded, the CPU has to work harder to manage the memory allocation for the newly created components as well as conduct garbage collecting on the memory of the deleted objects. This theory is reinforced by analysing the CPU and memory usage in which there are evident spikes. A couple of seconds into the program first running the spikes are caused by rendering,

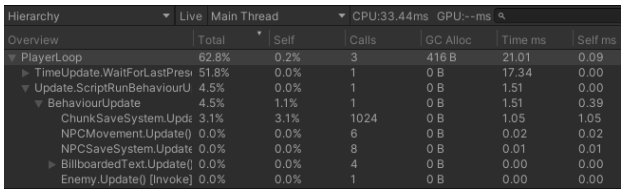

Figure 6: *CPU usage at peak time*

scripts and garbage collection, however later spikes are primary caused by garbage collection alone (figure: 5).

One solution to these problems would be an optimisation of the actual streaming process. This could be achieved through either the use of co-routines or Unity's Job system, either of which would allow for multiple multi-threaded sections of code to be to run simultaneously. This may not solve the problem however due to the large number of chunks being loaded and unloaded at one time as well as the relatively small nature of the code which would produce a large number of threads with a short lifetime, thus reducing its effectiveness. Another solution may be to add a level of detail to the world so that objects that are further away but still being rendered require less processing power, thus improving frame-rate or performance drops. The method in which the check is being run for chunks to load / unload could also be made more efficient since the current method result in each of the 1024 chunks checking the players position relative to their own, a better method would be for the player to determine is own current chunk and then load only the one around it, therefore reducing 1024 checks down to one check for the player and setting only the surrounding chunks to be active (figure: 6).

5 Conclusion

This project has created the basis of an open world game. A system has been created that allows for a game world to be generated from a heightmap and split into sections which can be saved, loaded and unloaded from a file. The world includes objects and non-playable characters that occupy the world and which are only loaded or execute behaviours based off of distance from the player.

References

Insomniac Games (2014). *Sunset Overdrive*.
Ruskin, E. (2015). *Streaming in Sunset Overdrive's Open World*.

6 Appendix

`https://youtu.be/ISwfGYr0vNs` - Week 1 video

`https://youtu.be/ndnbBN2xwd0` - Week 2 video

`https://youtu.be/068rqjmGnF4` - Week 3 video

`https://youtu.be/9L5yLf7mhtw` - Week 4 video

`https://youtu.be/UKtQUULYeHU` - Week 5 video