Open World Streaming

Ramy Abousaif
1800394

University of the West of England

April 26, 2021

This report outlines the procedures behind optimising open world games through various methods. All of which will be implemented in a prototype in the Unity Game Engine.

1 Introduction

Due to the sheer size and mass of open world games, there have been many techniques implemented over the past years to ensure that players would have a smooth and immersive experience. These methods include: frustrum/occlusion culling, chunk loading, dynamic LODs (Level of Detail) and, last but not least, JSON parsing. Methods like these have been used in so many open world games such as Horizon Zero Dawn (Guerrilla Games, 2017) [Figure 1] and Minecraft (Mojang, 2014) [Figure 2]. These methods will also be implemented for a prototype game in the Unity Game Engine.

2 Related Work

As aforementioned, the developers behind the popular game Horizon Zero Dawn (Guerrilla Games, 2017) have showed some behind the scenes implementations of systems such as their Frustrum Culling system. Using this method, they managed to only render part of the map and objects that are only visible in the player's camera/view.
A very well known implementation of chunk loading/unloading was explained in a GDC talk from the developers of the game Sunset Overdrive (Insomniac Games, 2014). Essentially, a chunk is a plot of, specific dimensions, of the map that can be simply visualised if the map were to be spliced up into several sections onto a grid (could comes in several shapes/forms such as squares, hexagons and triangles). With this method they managed to load chunks that are relatively close to the player (the distance depends on the settings applied by the player), while chunks that are far away from the player unload. This method goes hand in hand with the JSON parsing method, as it allows data in the unloaded areas to be saved in external JSON files and then loaded back in when the player comes back to those chunks once more. However this method varies in some games, like Sunset Overdrive [Figure 3], in terms of the external file's type as opposed to using JSON (DLL, TXT, etc.).

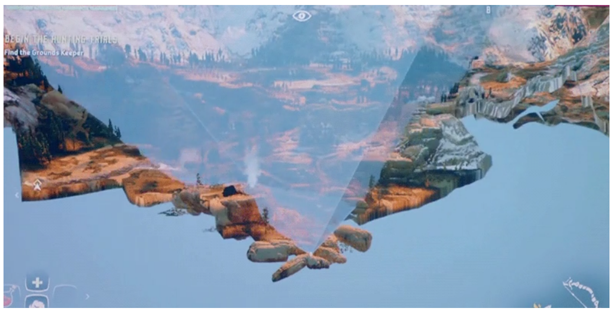
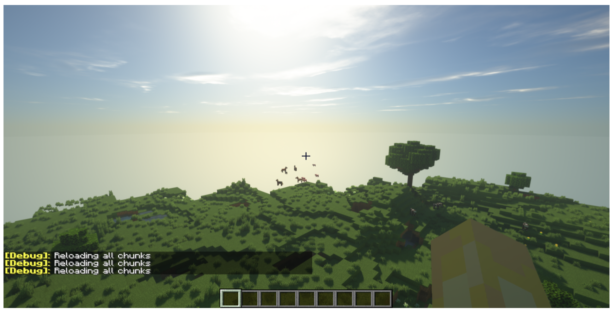
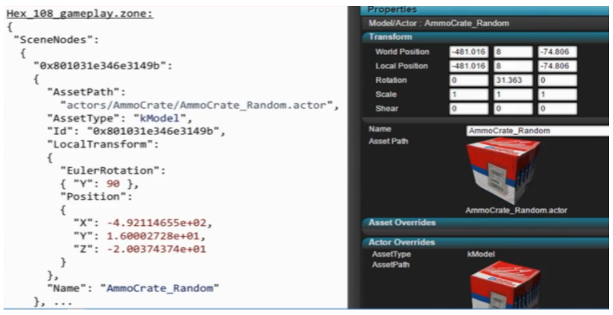


Figure 1: *Horizon Zero Dawn's implementation of Frustrum Culling.*



```
[Debug]: Reloading all chunks
[Debug]: Reloading all chunks
[Debug]: Reloading all chunks
```

Figure 2: *Minecraft's implementation of Chunk loading.*



```
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"
    }, ...
```

Properties

Model/Actor : AmmoCrate_Random

Transform

World Position	-481.016	8	-74.806
Local Position	-481.016	8	-74.806
Rotation	0	31.363	0
Scale	1	1	1
Shear	0	0	0

Name: AmmoCrate_Random

Asset Path: AmmoCrate_Random.actor

Asset Overrides

Actor Overrides

AssetType: kModel

AssetPath

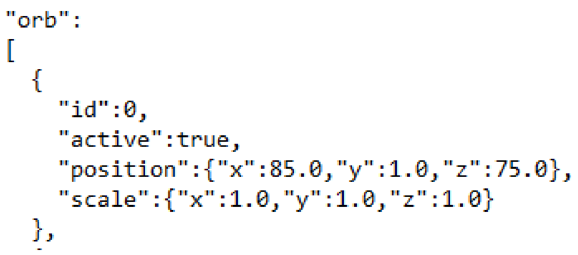
Figure 3: *Sunset Overdrive's external file reading.*

3 Unity Game Engine

Unity has a lot of useful features that helped throughout the entire development of the prototype. Along with the engine, the Universal Render Pipeline package has been installed to improve the game's aesthetics and visuals. Unity has built in features such as the terrain component which helped for loading and making height-maps for the ground. The terrain automatically has dynamic LODs built in that affects the terrain depending on the camera's position, the terrain tiles further away will have a low poly-count compared to the terrain tiles that are closer. Unity also offers the JSONUtility class, which allows its users to read/write from external JSON files.

4 Method

4.1 JSON



```
"orb":
[
  {
    "id":0,
    "active":true,
    "position":{"x":85.0,"y":1.0,"z":75.0},
    "scale":{"x":1.0,"y":1.0,"z":1.0}
  },
```

Figure 4: *Example of a field in a JSON file.*

In this prototype game, classes were made in advance for each objects that would need to get loaded in and out through JSON. The objects considered in this project were: The tiles for the terrains, trees, collectables (in this instance orbs) and unique objects. Each object would contain properties such as a unique int IDs, Vector3 values for their positions and sizes/scales and a boolean to see whether or not they were destroyed depending on if the object should be destroyed at all (in this case the collectables were the only objects to demonstrate this capability). In some cases some objects needed more properties such as the unique objects having their own separate render distance float. String variables were then created and initialized to read and store data from their respective JSON file. The data from the JSON files will then be stored into the properties in their relative object [Figure 4].

```
void SetJSONFile()
{
    terrainJSON = File.ReadAllText(Application.streamingAssetsPath + "/Load/Terrain.JSON");
    myJsonTerrainTiles = JsonUtility.FromJson<JsonTerrainTiles>(terrainJSON);
}
```

As mentioned earlier some objects are required to have properties saved in runtime. In this case, the collectables/orbs have the "active" bool that will represent whether or not they were collected. Each object that has a property similar to the orbs would need a separate script to set its current properties to the new ones that have been set.

```
[System.Serializable]
public class GameData
{
    public Orb.SaveToken[] orb;
}
private GameData gameData = new GameData();
List<Orb.SaveToken> orbList = new List<Orb.SaveToken>();
public void SaveOrbs()
{
    orbList.Clear();
    foreach (var orb in FindObjectsOfType<Orb>())
    {
        orbList.Add(orb.Tokenize());
    }
    gameData.orb = orbList.ToArray();
    Save(Application.streamingAssetsPath + "/Save/OrbList.JSON");
    Load(Application.streamingAssetsPath + "/Save/OrbList.JSON");
    sorbJSON = File.ReadAllText(Application.streamingAssetsPath + "/Save/OrbList.JSON");
    myJsonOrbs = JsonUtility.FromJson<JsonOrbs>(sorbJSON);
}
```

4.2 Chunk Loading

Because of the sheer size of most open world games, it is difficult to have everything loaded all at once from memory when it comes to the map and its contents. Which is why in most games, they often go with chunk loading. Usually the map would be sliced up into multiple sections or cells. The cells would be then referenced as chunks. The chunks would then be loaded in and out depending on the distance of the player. A similar approach was taken in this prototype. Instead of splitting a whole map into separate cells, the map would instead be made out of separated cells. There were multiple procedures that had to be taken before doing so however. First the making of the map's shape itself was made through the aforementioned terrain system in Unity. The dimensions of the map were 8x8 tiles, with each tile/cell being 25x25 units. Again, using the built in terrain system, height-maps and such were made through painting and brushing over the tiles until a natural bumpy terrain was formed [Figure 5].

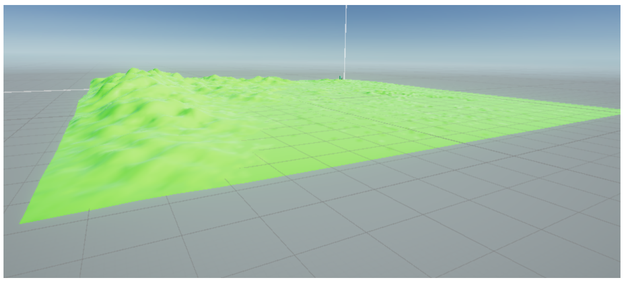


Figure 5: *The terrain in the prototype game.*

The height-maps were then extracted into .raw files, each file having the name "Terrain('Vector3 Position')". The reason for this will be explained later. Now that the terrain formation was done, it was time to load in the terrain depending on the player's position. While looping through each of the terrain tiles/cells the absolute distance between the center of each tile and the player is being stored (through the Pythagorean theorem).

$$a^2 + b^2 = c^2 \tag{1}$$

If the distance is larger than the set render distance, the tile will be not be created and the tiles within the render distance shall remain. As for implementing the height-maps that were mentioned earlier. Essentially, the terrain is being created in runtime and after it has been created, the tile is given specific properties that are predetermined such as the resolution, width, length and height. Once that's done the height-maps that were extracted get read and applied to the tile depending on the tile's position. After all the terrain data's details have been completed, it is then formed into a proper gameobject with its position, scale, material and ID passed into it. With all the knowledge leading up to this point, multiple JSON files can be used to load and unload other objects depending on the render distance such as trees, collectables and unique gameobjects (each of them can even have their own render distance set in the JSON files themselves).

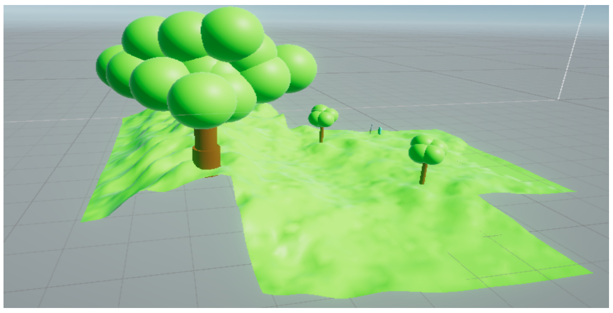


Figure 6: *Chunk of map including unique big tree.*

4.3 Culling

There are 2 types of culling when it comes to rendering objects in the player's camera view. Frustrum Culling and Occlusion Culling. They both serve the same purpose of only rendering gameobjects within the player's view instead of rendering the gameobjects throughout the scene all the time. The main difference between Frustrum and Occlusion culling however, is that occlusion culling will also not render objects hidden from the player's view, as in behind walls while Frustrum culling will do otherwise. For this prototype, Frustrum Culling was implemented. Once again Unity supplies its users with a function that essentially returns an array of planes (6 planes to be exact) from the camera's view frustrum. With these planes, it's possible to obtain a list of objects that are within the volume. The system will then loop through each object and check whether or not they are within that volume, if not disable their mesh renderers.

```
bool IsObjVisible(Renderer renderer)
{
    Plane[] planes = GeometryUtility
   .CalculateFrustumPlanes(cam);
    return (GeometryUtility.
   TestPlanesAABB(planes, renderer.
   bounds)) ? true : false;
}
```

4.4 NPCs

In this prototype game, the NPCs will not be using the built in pathfinding system from Unity. Instead, a custom made script will be used to allow the NPCs to traverse throughout the chunks. The enemy NPCs will have 3 states: wandering, chasing and attacking. In the wandering state, the enemy will lerp towards a random nearby location. The main issue that comes with this was that after chunks would unload, the enemies would occasionally fall through the map.

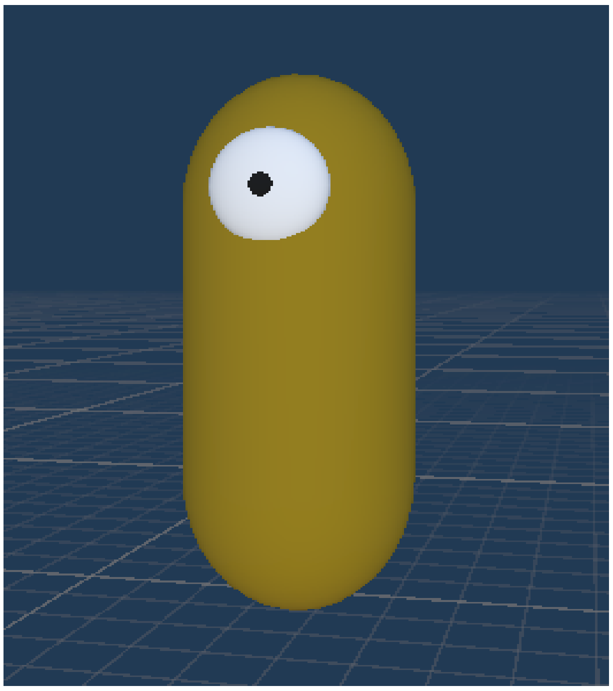


Figure 7: *Enemy cyclops in-game.*

To fix this issue, extra checks were made to ensure that the new random position would check whether or not there was something above or below that point. That way if the position is above the ground or in some cases below a high peaking location, it would always hit something vertically, if it doesn't then the enemy would move to another random location that meets those requirements. Another point worth mentioning in this state is that when the path is blocked by something and is unreachable, a new random location would be determined. The enemy has a view cone that has variables such as length and angle of view cone. If the player is within the view cone, the enemy enters the chase state. The chase state is pretty self-explanatory, the enemy will see the player, and will move towards them. Once the player reaches a certain distance, the enemy will go into the attack state. In the attack state, the enemy will of course damage the player and will retreat slightly backwards by applying a force, that is determined by a vector direction (obtained by subtracting the position of the enemy from the player's position and normalising it), while also applying some knockback to the player themselves. The enemy will then go back to the chase state and will repeat the same process until the player or the enemy is defeated.
Enemies will only spawn when a chunk is loaded, there is a predetermined chance (which is also defined in the JSON file that created the chunk). By default every time a chunk is loaded in, there's a 1/10 chance that the enemy will spawn anywhere within the boundaries of the chunk.

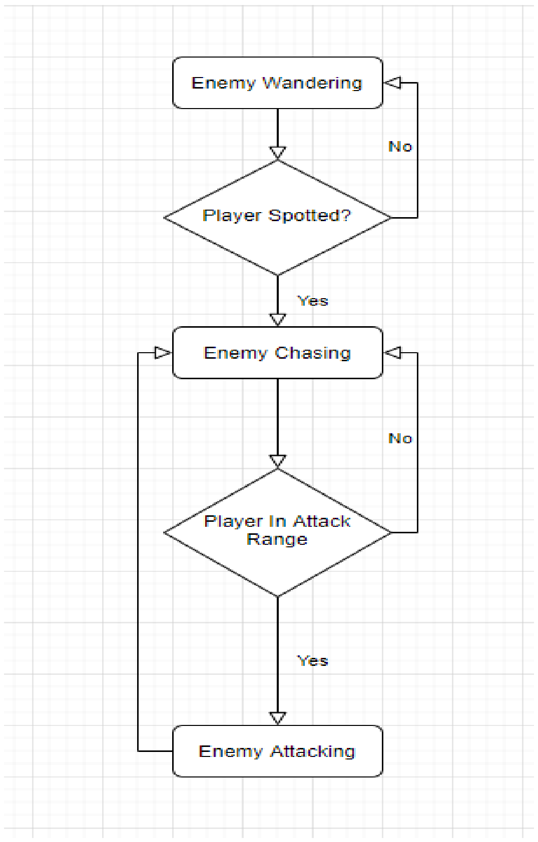


Figure 8: *Enemy states represented in a flowchart.*

5 Evaluation

An evaluation was made to ensure that the prototype can be further optimised in areas needed. For instance, loading and unloading data and chunks was occurring in the update function, meaning that everything explained earlier was happening per tick. To remedy this, coroutines were used and were running every 0.5 seconds or so. In terms of in-game experience, there's little to no effect due to the player's speed when it comes to moving close enough to load another chunk. This saved around 20 fps and made the game feel significantly more stable [Figure 9 and 10].

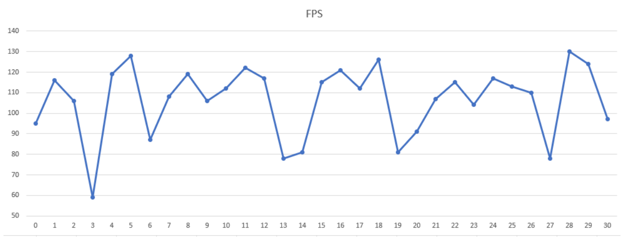


Figure 9: *FPS over 30 seconds before systems were implemented.*

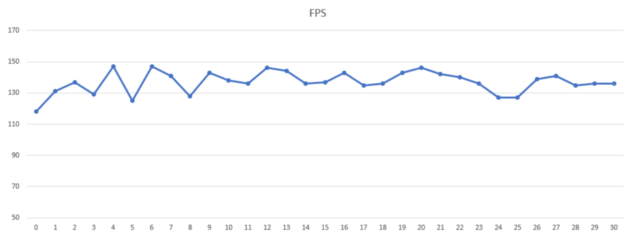


Figure 10: *FPS over 30 seconds after systems were implemented.*

6 Conclusion

This report described the methods and systems such as: JSON parsing, Chunk loading/unloading, Culling and NPCs (behaving to the ever-changing environment). All of which were implemented in a prototype game in the Unity Game Engine. The calculations, code examples and images referencing the prototype are given to further clarify these implementations.

Bibliography

Guerrilla Games (2017). “Horizon Zero Dawn”. In: URL: `https://www.slideshare.net/guerrillagames/decima-engine-visibility-in-horizon-zero-dawn`.

Insomniac Games (2014). “Sunset Overdrive”. In: URL: `https://www.gdcvault.com/play/1022268/Streaming-in-Sunset-Overdrive-s`.

Mojang (2014). “Minecraft”. In: URL: `https://www.minecraft.net/en-us/`.