AI for Video Games

Fernando Marcelo Edelstein Fernandez - 16644A

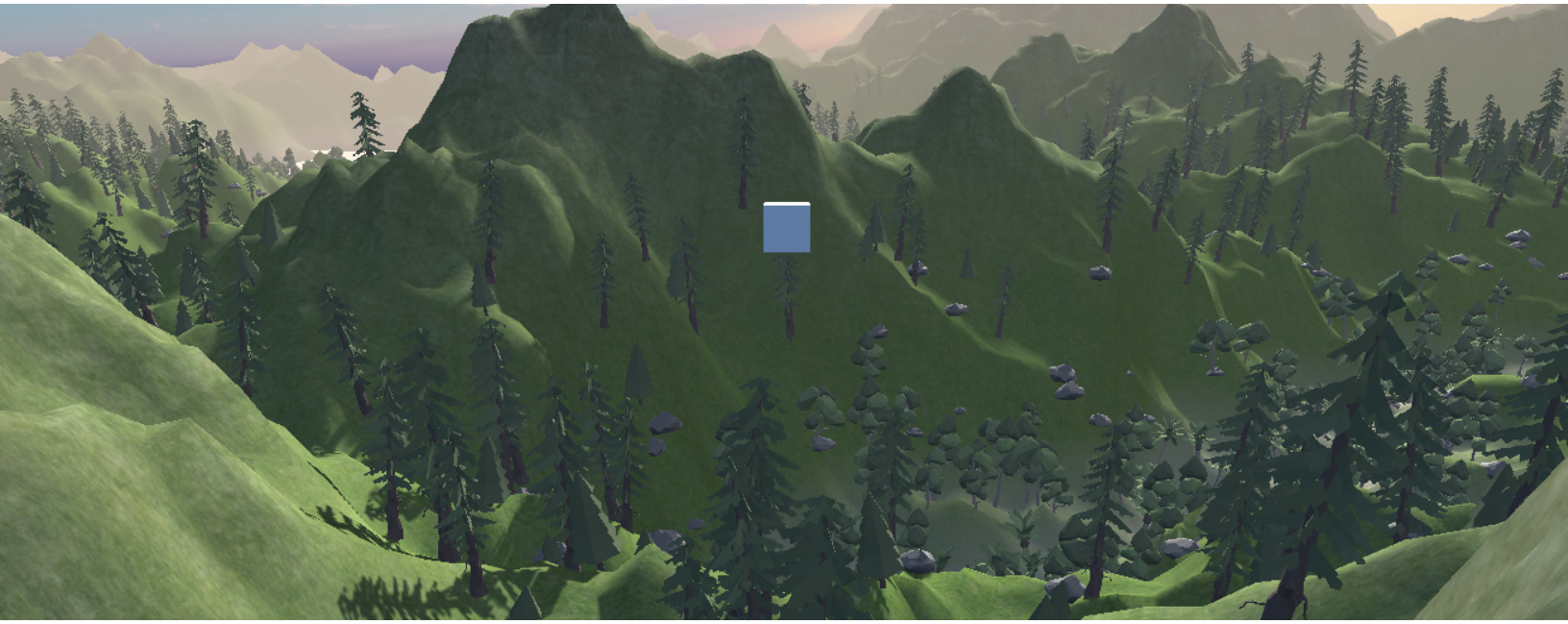
PROCEDURAL TERRAIN GENERATION

INTRODUCTION

The aim of this project is to create a procedural terrain generator that can spawn an infinite landscape type terrain by making use of the concepts of Procedural Content Generation (PCG) discussed in class as well as making use of different optimization techniques and algorithms that will allow for a smoother experience. The environment is generated by combining different techniques in order to make the experience more believable such as Perlin noise, to create realistic-looking terrain features such as mountains, valleys and lakes. Additionally, the system is capable of placing trees, foliage or just any kind of object in the scene, ensuring that the environment is fully fleshed out and believable.

The terrain generation system is meant to be fully customizable, allowing developers and level designers to tweak and adjust the settings to suit their specific needs.

This kind of implementation is inspired by different games such as Minecraft and Flight Simulators.



ABSTRACT

Before explaining how the different classes work, it is important to explain the basic logic behind the code. At the base we have a Perlin noise which is responsible for generating the noise map which will later be used to generate the shape of the terrain. Most of its features are exposed so that it can be customised, but a singular noise map can only produce so many different and unexpected results, so in order to fix this issue, the Noise generation class allows users to add up different Perlin noises that can be customised differently. These different noise layers are then averaged so that the final result is a combination of many features.

Once the resulting noise pattern has been calculated, a class called **HeightMapGenerator.cs** is responsible for generating the different height parameters based on the provided settings and generated noise on a 2D array. This class will save all its calculations on a Struct (due to its lightweight properties) called **HeightMap**, which will be important for performing further operations once the terrain has been generated. In order to create an infinite terrain, the landscape is divided into different chunks of the same size which spawn in different Level of Detail (LOD) depending on the player's position.

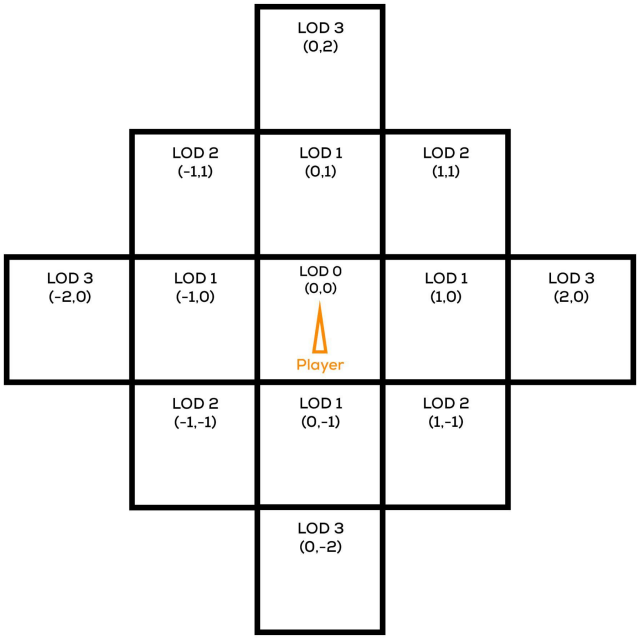


Figure 1.0 Chunks generation based on player's position

In figure 1.1 it is shown how the LOD works. The player has a position of (0,0) in this case, and different LOD meshes are created on its surroundings so that the player is not able to tell the terrain is not really endless. Each chunk is assigned an immutable HeightMap with an established noise pattern which makes the terrain deterministic; the player might be at position (0,0), move 5 chunks to the left, come back and find the same (0,0) chunk there was before.

Chunks are generated on a class called **TerrainChunk** which is the one that makes the request for the Noise pattern, which is later used for the HeightMap calculation and finally used for generating the mesh. It's important to note that chunks use a pre-established noise pattern and its surroundings are generated by shifting the offset parameter of the noise so that the pattern is kept between chunks.

The **MeshGenerator** is the class in charge of generating the mesh that is going to be displayed. Inside of it there is a class called MeshData which contains all the informations related to the generated mesh such as the vertices list, triangles list as well as different functions such as triangles creation, vertex creation and normals calculation (this class will be discussed in detail in further chapters).

The TerrainChunk class is not only responsible for requesting the noise and the mesh according to it, but also it is in charge of constantly updating the chunks and the objects on it. Different LOD meshes are only generated when required, meaning that, in the case of Figure 1.1, the chunk (0,0) will only contain the mesh on a LOD level of 0 (Higher detail mesh) and its lower LOD version will only be generated once the player moves, same thing happens on the surrounding meshes, this is done so that the mesh generation process is faster. These different LOD meshes are stored inside of each chunk so once the required LOD levels are generated it becomes a matter of switching from one to the other (More performance details in further chapters). On the other hand, this class also generates the water in the scene, which is done by skipping most of the mesh generation process and being limited to spawning a plane on the desidered height. Finally this class is the one in charge of generating the objects in the scene and disappearing them when they are not needed.

All these classes are orchestrated by the **InfiniteTerrainGenerator** class which is the one added to the empty object in the scene called MapGenerator. This class is the one that receives all the necessary parameters from the user, such as the amount of LODs to be created, the visible distance of each of these, making sure that every chunk is generated within this distance and disabled when the player moves far enough.

Lastly the **MapPreview** class allows users to preview the terrain mesh that will be generated once they enter in Play mode.

HOW IT WORKS

The system can be customised in various ways, by default, it will spawn a player which can be moved by the user using the W-A-S-D keys, but it can be changed to the viewer object which is disabled. The Map Preview allows users to estimate the shape of the terrain to be generated and also, to preview the generated noise in a plane. These shapes can be saved as assets which can be swapped at any time, the same happens for the Mesh Settings, Object Spawner and Texture data.

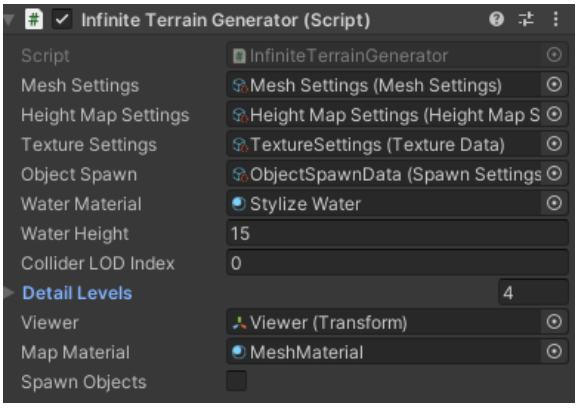


Figure 0.1 Terrain Settings

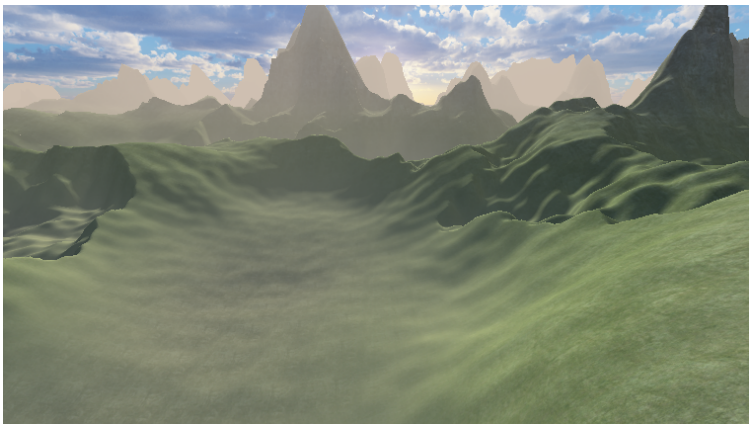


Figure 0.2 First Run

Once in Play Mode, a possible result could be like the one seen on figure 0.2, a terrain on which there are a series of landscapes: mountains, lakes and prairies, customisation is possible by modifying the shape, colours and controlling the elements to be spawned.
Depending on the system's performance and the amount of terrain to be generated, the spawning radius can be set on the Detail Levels array.

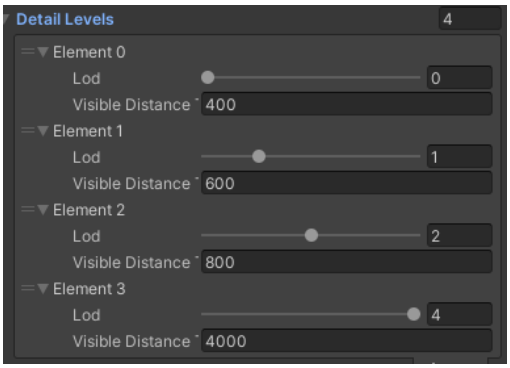


Figure 0.3 Detail Levels

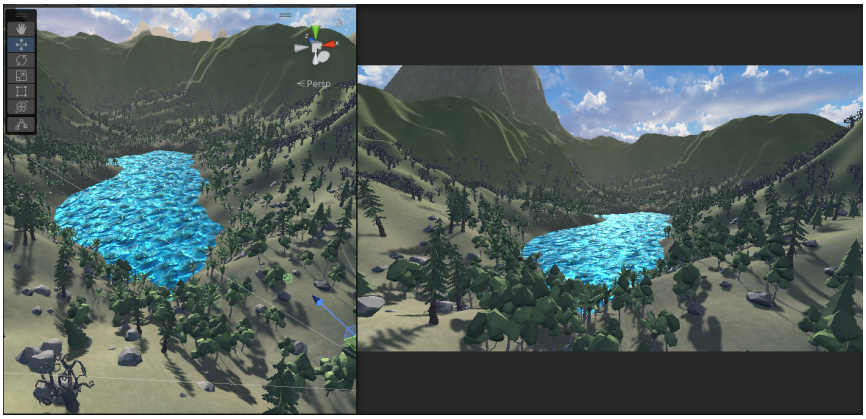


Figure 0.4 World Creation

It is important to keep in mind that the higher the amount of meshes to create and objects to spawn, the slower the system will run. Consider lowering both the parameters in slower cpus.
Notice that in the MapPreview object, if the user tries to spawn objects right after opening the project, he will see a Null Reference Exception, this is because the MeshData has not been generated yet, this is easily fixed by just clicking “Generate” on the MapPreview and then “Generate” in the ObjectSpawn script.

THE CLASSES

Noise
This is a static class which works using another inner class called **NoiseSettings** which exposes all the parameters to the user:

- Normalise Mode
- Scale
- Octaves
- Persistence
- Lacunarity
- Seed
- Offset
- Blend
- A validate method which ensures that parameters are not out of bounds

This class has two methods: a GenerateNoiseMap which returns a float[,] variable containing the generated noise map and a QuickNoiseMap which returns a Texture 2D.

GenerateNoiseMap, this method takes the width, height, noise settings and a sample centre. Firstly each octave is generated on different sample locations so to do so, there is an array of Vector2 called octaveOffsets which contains all the different sampling locations. This method will cycle through the supplied width and height and calculate the Perlin Noise value using the *Mathf.PerlinNoise* expression. It is important to mention that one of the parameters is the *Octaves* which implies that there is another for loop which will be added to the noiseHeight parameter and decreasing the amplitude and frequency values per each iteration, so a much more controlled result is obtained.

```
for (int y = 0; y < mapHeight; y++) {
    for (int x = 0; x < mapWidth; x++) {
        amplitude = 1;
        frequency = 1;
        float noiseHeight = 0;

        for (int i = 0; i < settings.octaves; i++) {
            float sampleX = (x - halfWidth + octaveOffsets[i].x) / settings.scale * frequency;
            float sampleY = (y - halfHeight + octaveOffsets[i].y) / settings.scale * frequency;

            float perlinValue = Mathf.PerlinNoise(sampleX, sampleY) * 2 - 1;
            noiseHeight += perlinValue * amplitude;
            amplitude *= settings.persistance;
            frequency *= settings.lacunarity;
        }
        if (noiseHeight > maxLocalNoiseHeight) {
            maxLocalNoiseHeight = noiseHeight;
        }
        if (noiseHeight < minLocalNoiseHeight) {
            minLocalNoiseHeight = noiseHeight;
        }
            noiseMap[x, y] = noiseHeight;
```

Figure 1.1 Noise Creation

Normalise Mode: This parameter is an enum which can be set to Local and Global. This is necessary in order to normalise the values generated by the noise, the reason why there are two possible settings is that Local mode is ideal for generating local noise maps by cycling through the pre-generated noise and using the InverseLerp function but min and max noise height will have slightly different values from chunk to chunk, making chunks to not align properly. Global mode is useful for when many chunks are being spawned but their min and max height can not be known before creation, so the values are clamped from 0 to int max value so that the height is always positive.

The *QuickNoiseMap* is a method used for spawning objects and its only function is to quickly generate some noise to be used for the density slider (More details will be discussed).

HeightMapGenerator

This class is the one in charge of generating the HeightMap itself based on the Noise parameters. It contains a **HeightMap** struct inside of it that holds the minValue, the maxValue and a 2D Float array that holds the heightValues based on the width and height. The *HeightMapGenerator* contains a method called *GenerateHeightMap* which initially generates one HeightMap based on the first supplied noise. Since a Perlin Noise can only create so much variation, it is possible to add a second noise so that different biomes can be created such as valleys, lakes, mountains and prairies. This second noise will be blended using one of the different blending modes available: PolySmoothMin, SquaredMin, sMinCubic. In order to manually modify the highness, there are two parameters the user can modify, the Height Multiplier and the Height Curve. The curve is a bit problematic when it comes to working with different threads, this is why there is a variable called heightcurve_safe which is used to avoid overlapping.
It is worth noticing that once the curve has been evaluated, the min and max values will be updated and finally a HeightMap object will be returned.

Scriptable Objects - Asset Menus Used for saving the desidered settings

UpdatableData
This class inherits from ScriptableObject and its function is to be the parent class for further Scriptable Objects, allowing them to be auto updated on the editor and notifying when values are changed.

HeightMapSettings
This class specifies all the settings the mesh is going to use in order to generate the noise and the mesh heightness:

- public NoiseSettings[] noiseLayers
- public BlendType blendType
- public float heightMultiplier
- public AnimationCurve heightCurve

This class provides a OnValidate method in order to be sure all noise settings are within acceptable ranges.

MeshSettings
TerrainChunks can only be spawned in a series of sizes, because the values must be divisible by the amount of supported LODs (By default there are 6 levels of LOD) so this are the supported chunk sizes
public static int[] supportedChunkSizes = { 48, 72, 96, 120, 144, 168, 192, 216, 240 };
Each of these values is divisible by 6 and so, simplification is possible with these parameters.

SpawnSettings
Contains a struct called ObjectType which is exposed to the user and specifies some parameters about the objects to be spawned:

- *public string name;*
- *public List<GameObject> objectList;*
- *public float density;*
- *public float objectMinSize;*
- *public float objectMaxSize;*
- *public float minPositionHeight;*
- *public float maxPositionHeight;*

TextureDataSettings
Contains a class called Layer which is exposed to the user and contains all the information for each layer to be rendered. This class communicates with the material shader and sends the information about the mesh.
Texture2DArray GenerateTextureArray(Texture2D[] textures)
Uses the input array to create a Texture2DArray which will be sent to the Material.

UpdateMeshHeights
This method is used to update the current min and max height from the generated mesh.

ApplyToMaterial(Material material)
Sends all the information contained on each Layer to the input material as well as the Texture2DArray.

TerrainChunk

This class is the one that generates singular terrain chunks for the map. By using the sample centre provided by the constructor, it calculates the mesh position based on the MeshWorldSize parameter and the bounds. By default, terrain chunks are not visible, they are updated once the UpdateChunk method is called. In order to save some performance power, each chunk is only generated on the required chunk at first, so if a chunk is created at LOD = 2 when the game starts, LOD 0 and 1 are not calculated immediately but rather, when the player gets close enough, this distance value is specified on each LOD level.

```
if (visible) {
    int lodIndex = 0;
    for (int i = 0; i < detailLevels.Length; i++) {
        if (viewerDistanceFromNearestEdge > detailLevels[i].visibleDistanceThreshold)
            lodIndex = i + 1;
        else
            break;
    }

    if (lodIndex != prevLODIndex) {
        LODMesh lodMesh = detailLevelsMeshes[lodIndex];
        if (lodMesh.hasMesh) {
            prevLODIndex = lodIndex;
            meshFilter.mesh = lodMesh.mesh;
        }
        else if (!lodMesh.hasRequestedMesh) {
            lodMesh.RequestMesh(heightMap, meshSettings);
        }
    }
```

Figure 1.2 LOD Mesh creation

Creating each chunk

This process has been splitted in different Threads in order to prevent the game from freezing when calculating new chunks in play mode. It is worth noticing that Unity is not Thread safe, so a few measures were considered in order to synchronise Threads and make sure that processes that require to be done in the main Thread do have the necessary variables.
Essentially, two processes are done on separate Threads: The *GenerateHeightMap* and the *GenerateTerrainMesh* method.

```
public void Load() {
    ThreadedDataRequester.RequestData(() => HeightMapGenerator.GenerateHeightMap(mes
}


1 reference
void OnHeightMapReceived(object heightMapObject) {
    this.heightMap = (HeightMap)heightMapObject;
    heightMapReceived = true;

    UpdateChunk();
}
```

Figure 1.3 HeightMap request

```
1 reference
void OnMeshDataReceived(object meshDataObject) {
    mesh = ((MeshData)meshDataObject).CreateMesh();
    hasMesh = true;

    meshData = (MeshData) meshDataObject;
    updateCallback();
}

2 references
public void RequestMesh(HeightMap heightMap, MeshSettings meshSettings) {
    hasRequestedMesh = true;
    ThreadedDataRequester.RequestData(() => MeshGenerator.GenerateTerrainMesh(hei
}
```

Figure 1.4 TerrainMeshRequest

There is a supporting class for this task called the **ThreadedDataRequester** which implements generic methods for different kinds of requests. It has a struct called *ThreadInfo* which specifies a generic readonly callback and a generic parameter. This class works using a queue of *ThreadInfo* which is used to check whether there are requested functions and send them to the requesting class if needed. A method called *RequestData* is the one used by external classes and it delegates the task to an instance of *DataThread* on a separate Thread and starts it. The *DataThread* is an internal method that runs the parsed function and locks the queue while enqueuing the results.

```
static ThreadedDataRequester instance;
Queue<ThreadInfo> dataQueue = new Queue<ThreadInfo>();

Unity Message | 0 references
private void Awake() {
    instance = FindObjectOfType<ThreadedDataRequester>();
}

2 references
public static void RequestData(Func<object> generateData, Action<object> callback) {
    ThreadStart threadStart = delegate {
        instance.DataThread(generateData, callback);
    };
    new Thread(threadStart).Start();
}

1 reference
void DataThread(Func<object> generateData, Action<object> callback) {
    //When calling a method from inside a thread, the method will run on the thread
    object data = generateData();

    lock (dataQueue) {
        //The lock will block the thread so that no other process can call it while it's been already called
        dataQueue.Enqueue(new ThreadInfo (callback, data));
    }
}

Unity Message | 0 references
void Update() {
    if (dataQueue.Count > 0) {
        for (int i = 0; i < dataQueue.Count; i++) {
            ThreadInfo threadInfo = dataQueue.Dequeue();
            threadInfo.callback(threadInfo.parameter);
        }
    }
}

5 references
struct ThreadInfo {
    public readonly Action<object> callback;
    public readonly object parameter;

    1 reference
    public ThreadInfo(Action<object> callback, object parameter) {
        this.callback = callback;
        this.parameter = parameter;
    }
}
```

Figure 1.5 Threading Implementation

The TerrainChunk class has a method called *Load* which makes the request for a HeightMap on a separate Thread, the *ThreadedDataRequester* calls the *OnHeightMapReceived* method once the Thread has finished which runs the *UpdateChunk* method.

UpdateChunk is the method that handles the visibility of the chunk and switches the mesh to the different LOD levels. If the LOD is available, the method simply switches it, otherwise, it calls a class, the LODMesh which contains the mesh itself and a method to run a separate Thread to generate the terrain mesh. Once calculated, the method *OnMeshDataReceived* creates the Mesh and stores the MeshData on a local variable, this will be useful later on for spawning objects. The class works with a reference of an ObjectSpawner which generates objects based on the MeshData, so once the requested mesh has been generated or activated, the class checks for whether or not objects have been already spawned and makes them visible. It is worth noticing that objects will only appear on the LOD = 0 which is where the player is. In order to improve performance, once the mesh increases its LOD level, objects are disabled. This operation is performed inside the TerrainChunk class since it is the one that keeps track of the current LOD of the chunk considering the distance of the player.
Another important method of the class is the *UpdateCollider* which is in charge of generating the collider for the meshes. An important thing to take into consideration is that the collider calculation consumes a significant amount of CPU power, like shown on figure 1.6.

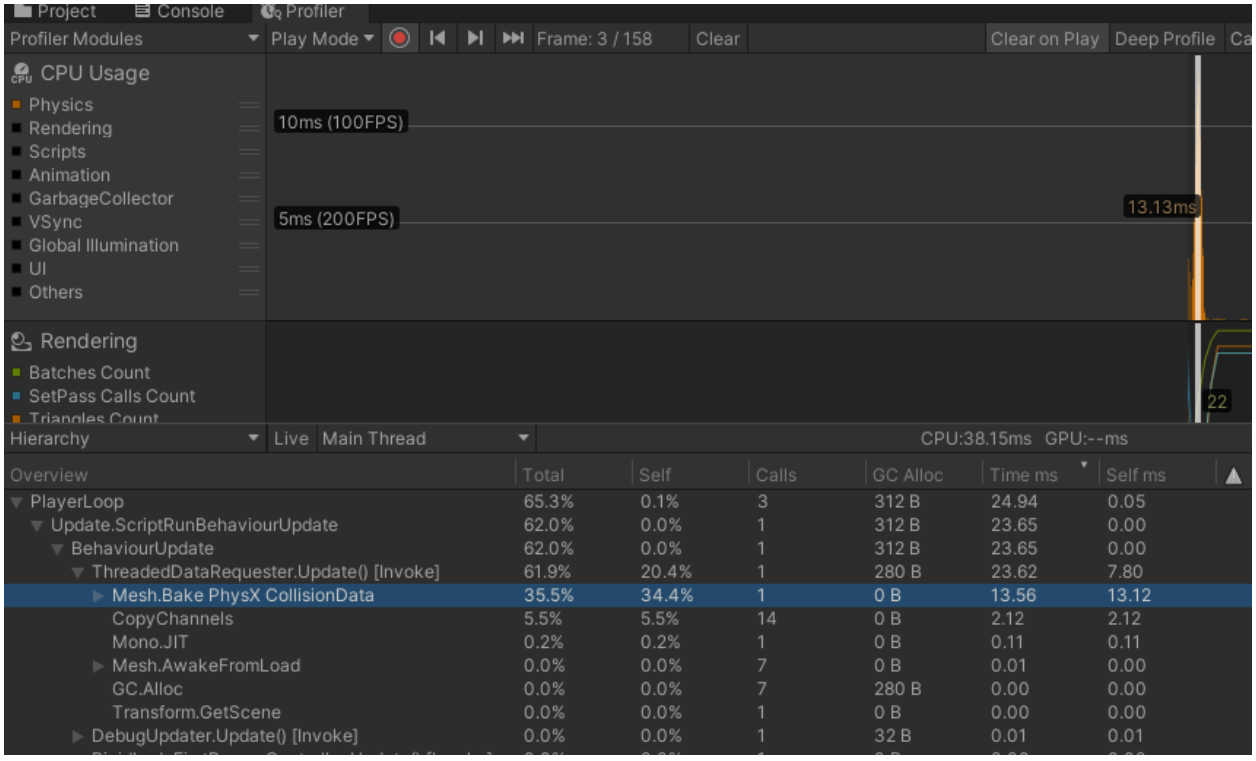


Figure 1.6 Profiler - Collider calculation

In order to make the game run smoother, the collider is only spawned when the player is close enough to the next chunk by taking into consideration the squared distance from the viewer's position and the squared distance from viewer to edge. Once close enough, the

Lastly, this class is responsible for creating the water chunks which are spawned using the second constructor, which works basically as the first one but instead of calculating a mesh, it spawns a plane of the given size. It also has an *UpdateWaterChunk* method that only considers the distance of the player and the distance to the edge.

```
        if (spawnObjects) {
            if (objectsHaveBeenSpawned) {
                if (lodIndex != 0)
                    objectSpawner.setVisibleObjects(false, position);
                else
                    objectSpawner.setVisibleObjects(true, position);
            }
        }
    }

    if (spawnObjects) {
        if (detailLevelsMeshes[0].hasMesh) {
            meshData = detailLevelsMeshes[0].meshData;
            if (!objectsHaveBeenSpawned)
                SpawnObjects();
        }
    }
```

Figure 1.7 Object Spawning

MeshGenerator

Of course, every mesh requires information about the triangles in order to be rendered, this is what this class is for.

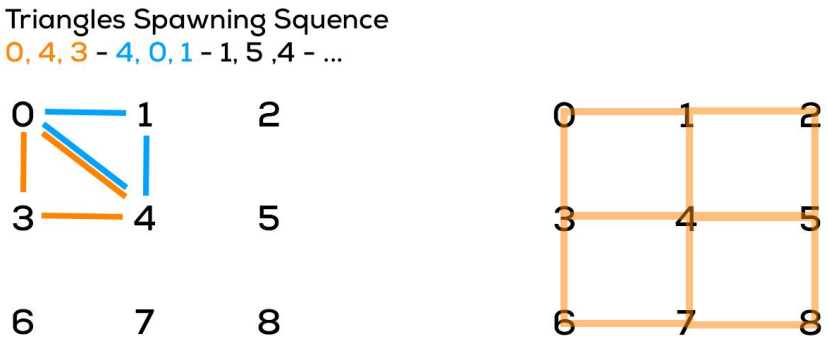


Figure 1.8 Triangles and vertices Generation

First of all, vertices are calculated clockwise, so, like shown on Figure 1.8 the first triangle to be created is the 0 - 4 - 3, next will be 4 - 0 - 1 and so on. In order to store this vertices, it is mandatory to know how many vertices are going to be needed before creating the array

*Vertices = Width * height*.

Now, in order to know how many triangles are formed, it can be useful to know how many squares there are, so the amount of triangles is given by subtracting 1 from the width and height and multiplying the result by 2 since a square is made by two triangles and by 3 since a triangle is made of 3 vertices.

*Triangles = (Width - 1) * (Height - 1) * 2 * 3*

This is the elementary principle of how triangles and vertices are created, however, it will be shown that for practical reasons and issues, this is not the actual implementation that can be found in the code. It is convenient to store all this information in a separate class called **MeshData** which is useful for spawning objects in the scene and handling operations such as calculating normals. This class contains both these arrays and other functions that will be further explained.

LOD

An important optimization step is to allow for different LOD of the mesh. This process works by simplifying the mesh vertices.

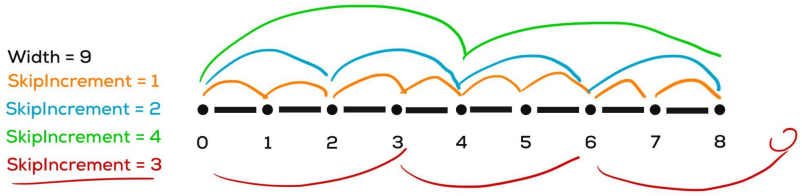


Figure 1.9 Skip Increment

On Figure 1.9 There is an example of how the simplification works, it is worth noticing that not every number is admissible since, as seen on *SkipIncrement = 3*, there is no 9th vertex in this case, so if, for example, *width = 9*, the skip increments must be a factor of *width - 1*, in this case, 1, 2, 4 and 8 are admissible. The formula for this is *(w-1)/i + 1.*

Since Unity has a limit of *16 bit index buffer = 65535* vertices per mesh, a workaround for this could be to implement a GPU based calculation which allows for bigger chunks, however it is better to make this system as compatible as possible across multiple systems. It is important to have a fixed squared size for the mesh chunks. When the *i = 1*, the vertices *v = width * height*, which can also be written as *v = w^2*. Because of Unity's limit, the maximum supported size is 255, so *width <= 255*, the chosen maximum width is 241 since it has the property of being divisible by all even numbers up to *12 (2,4,6,8,10,12)*.

One of the main problems of this system is that chunks of different LOD did not align properly, leaving the map with some visible gaps between chunks. In order to solve this issue, the system generates a higher LOD border and a lower LOD in the centre so that chunks' edges will match properly, like shown in figure 2.0.

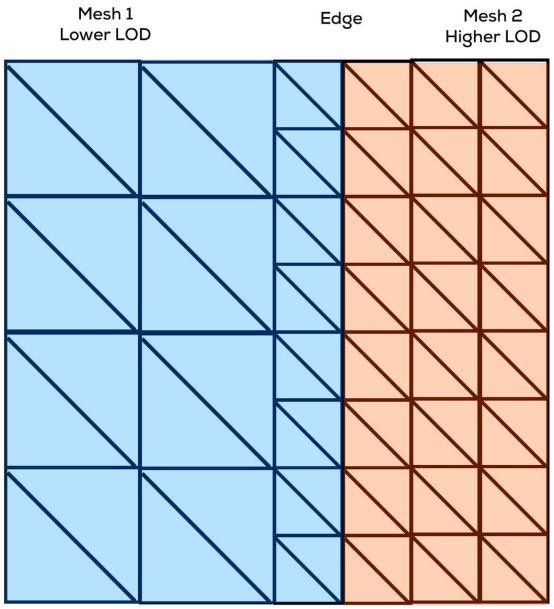


Figure 2.0 Mesh Edges

This solution is not perfect though as it does show inconsistent edges when applying it on lower resolution meshes but this error does not happen on the highest levels of detail so it is an acceptable solution since the player will not be able to see the mesh division.

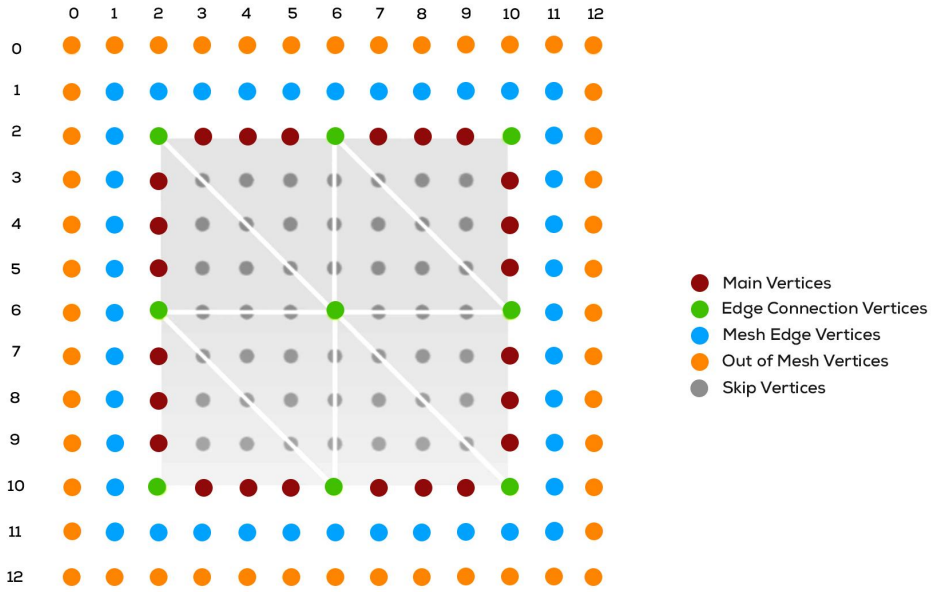


Figure 2.1 Mesh Division Diagram

In figure 2.1 there is a graphical explanation of how the different edges are generated. Essentially, if, for example, there is a mesh of 9x9, the mesh is initially composed of the *SkipVertices, EdgeConnectionVertices* and the *MainVertices*. Around this mesh, there is a Out of *MeshVertices* surrounding that is used to calculate normals. What is necessary is to add a layer of vertices in between the two, the *MeshEdgeVertices* which will be generating all the connection points for the high resolution edge vertices by connecting vertices to the *MainVertices* and the *EdgeConnectionVertices* (also used for mesh simplification).

The GenerateTerrainMesh identifies which vertices belong to the OutOfMesh vertices, then taking the figure 2.0 into consideration, since triangles are created from the upper left corner, they are spawned at every point but the lowest row and rightmost column of the OutOfMesh vertices and the leftmost column and upper row of the Main Vertices like shown on figure 2.2

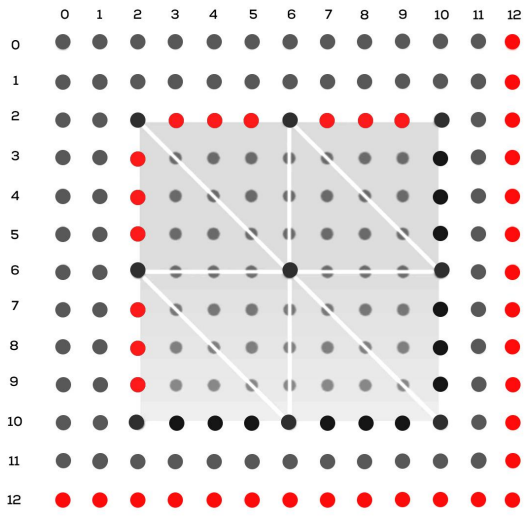


Figure 2.2 All the vertices that are not considered for creating triangles

It is important to create the triangles in their correct size, so the simplified triangles are created on the EdgeConnectionVertices shown on Figure 2.1 except for the lowest row and rightmost column.

```
bool createTriangle = x < numberVertexPerLine - 1 && y < numberVertexPerLine - 1 && (!isEdgeConnectionVertex || (x != 2 && y != 2));

if (createTriangle) {
    int currentIncrement = (isMainVertex && x != numberVertexPerLine - 3 && y != numberVertexPerLine - 3) ? skipIncrement : 1;
    int a = vertexIndecesMap[x, y];
    int b = vertexIndecesMap[x + currentIncrement, y];
    int c = vertexIndecesMap[x, y + currentIncrement];
    int d = vertexIndecesMap[x + currentIncrement, y + currentIncrement];

    meshData.AddTriangle(a, d, c);
    meshData.AddTriangle(d, a, b);
}
```

Figure 2.3 Triangles creation

The *MeshData* class contains a few methods that provide functionalities such as triangle creation and a Finalise method which will calculate Normals, this is done by extracting the vertices from the Vertices Array and then using the Cross Product of the sides of the vertices.

```
2 references
Vector3 SurfaceNormalFromIndexes(int indexA, int indexB, int indexC) {
    Vector3 pointA = (indexA < 0) ? outOfMeshVertices[-indexA - 1] : vertices [indexA];
    Vector3 pointB = (indexB < 0) ? outOfMeshVertices[-indexB - 1] : vertices [indexB];
    Vector3 pointC = (indexC < 0) ? outOfMeshVertices[-indexC - 1] : vertices [indexC];

    Vector3 sideAB = pointB - pointA;
    Vector3 sideAC = pointC - pointA;

    return Vector3.Cross(sideAB, sideAC).normalized;
}
```

FIgure 2.4 Normals Calculation

once triangles and vertices have been created. Finally this class also contains a method called CreateMesh which will essentially create the mesh itself with all the generated information.

InfiniteTerrainGenerator

This is the main class that commands all the others, essentially it takes the MeshSettings, HeightMapSettings, TextureSettings and ObjectSpawn assets, and generates the terrain based on the supplied parameters. This class works with a Viewer which is also supplied in the inspector, as well as a list of all the desidered LOD and distances per each of them.
Its main task is to update the TerrainChunks in the scene, to do so, it uses the information from the supplied viewer (The Player) and constantly checks its position.

```
private void Update() {
    viewerPosition = new Vector2(viewer.position.x, viewer.position.z);

    if (viewerPosition != prevViewerPosition) {
        foreach (TerrainChunk chunk in visibleTerrainChunks) {
            chunk.UpdateCollider();
        }
    }

    if ((prevViewerPosition - viewerPosition).sqrMagnitude > sqrViewerMoveThresholdChunkUpdate) {
        prevViewerPosition = viewerPosition;
        UpdateVisibleChunks();
    }
}
```

FIgure 2.5 InfiniteTerrainChunk Update Method

There are three basic operations to be done in this system, the first being checking the viewer's position and updating the corresponding chunks, spawning chunks and updating the collider. The UpdateVisibleChunks is responsible for spawning the TerrainChunks and the WaterChunks (They are essentially the same class but in the code they are called differently for clearance). The class uses a *Dictionary<Vector2, TerrainChunk> terrainChunkDictionary = new Dictionary<Vector2, TerrainChunk>();* And a List

List<TerrainChunk> visibleTerrainChunks = new List<TerrainChunk>();
The first one is used to store the information of the already spawned chunks and the list is used to keep track of the visible chunks in the scene. Notice that the procedure is the same for the water chunks, since a Dictionary can not contain two keys with the same position, it is mandatory to use a second Dictionary. The *UpdateVisibleChunks* method is responsible for the constant update of the chunks themselves, firstly, it iterates over the visible chunks and calls the *UpdateChunk* method from the *TerrainChunk* class in order to get the right LOD and make it either visible or invisible, then, it calculates the current coordinate for the spawning chunk.

```
for (int yOffset = -chunksVisibleInViewDistance; yOffset <= chunksVisibleInViewDistance; yOffset++) {
    for (int xOffset = -chunksVisibleInViewDistance; xOffset <= chunksVisibleInViewDistance; xOffset++) {
        //Spawn terrain chunk on surounding coordinates
        Vector2 viewedChunkCoord = new Vector2(currentChunkCoordX + xOffset, currentChunkCoordY + yOffset);

        //Check if chunk has been updated already
        if (!alreadyUpdatedChunkCoords.Contains(viewedChunkCoord)) {
            //Check if chunk exists already
            if (terrainChunkDictionary.ContainsKey(viewedChunkCoord)) {
                terrainChunkDictionary[viewedChunkCoord].UpdateChunk();
            }
            else {
                TerrainChunk newChunk = new TerrainChunk(viewedChunkCoord, heightMapSettings, meshSettings, meshWorldSize, c
                terrainChunkDictionary.Add(viewedChunkCoord, newChunk);
                newChunk.onVisibilityChanged += OnTerrainChunkVisibilityChanged;

                newChunk.Load();
```

Figure 2.6 InfiniteTerrainGenerator - TerrainChunk creation

The method checks for whether or not the chunk has already been created, if it was, then it updates its visibility and if not, it creates a new one.

ObjectSpawner
This class is the one responsible for spawning every desidered object in the scene, based on the highness of the mesh.
The process is quite simple, it uses an instance of **ObjectType** which specifies the density of the object to be spawned and the min and max size and position of it.

```
for (int i = 0; i < vertices.Length - 1; i++) {
    float noiseMapValue = noiseMapTexture.GetPixel(i, Random.Range(1, i)).g;

    if (noiseMapValue > 1 - Random.Range(0.0f, objectSpawn.Spawner[j].density)) {
        if (vertices[i].y >= (objectSpawn.Spawner[j].minPositionHeight + Random.Range(0,10)) &&
            vertices[i].y <= (objectSpawn.Spawner[j].maxPositionHeight) + Random.Range(0,10)) {

            int randomPrefab = Random.Range(0, objectSpawn.Spawner[j].objectList.Count - 1);

            //Add a random number within a certain range to make the spawning more natural
            Vector3 pos = new Vector3(vertices[i].x + Random.Range(-1f, 1f) + coordinates.x, vertices[i].y, vertices[i].z + Ran

            GameObject go = Instantiate(objectSpawn.Spawner[j].objectList[randomPrefab],
                            pos,
                            Quaternion.Euler(new Vector3(Random.Range(-10, 10), Random.Range(0, 360), Random.Range(-10, 10))),
                            parent.transform);

            go.transform.localScale = Vector3.one * Random.Range(objectSpawn.Spawner[j].objectMinSize, objectSpawn.Spawner[j].
```

FIgure 2.7 ObjectSpawner

In order to get a more controlled result, the density slide works by using a QuickNoiseMap and comparing it to 1 - density, this way, it is clear for the class where to spawn the objects.

This class uses the already created vertices in order to find the right heightness of the object. In order to avoid having all objects spawned on a vertex point and get a predictable result, a slight variation is added to every position so that the result looks more natural.

TerrainShader

The shader has been manually coded. It basically works by considering the height values of the generated terrain, using different colours and textures. In order for the shader to get this information, it has to be supplied somehow, this is where the *TerrainData* script takes place. The most important function inside the Shader is the *Surf* method, which is the one to render all the supplied settings. The texture is applied using the TriPlanar method, the idea is that the texture is mapped three times along X, Y, Z axes, then blend them based on the angle of the face.

```
float3 triplanar(float3 worldPos, float scale, float3 blendAxes, int textureIndex) {
    float3 scaledWorldPos = worldPos / scale;

    float3 xProjection = UNITY_SAMPLE_TEX2DARRAY(baseTextures, float3(scaledWorldPos.y, scaledWorldPos.z, textureIndex)) * blendAxes.x;
    float3 yProjection = UNITY_SAMPLE_TEX2DARRAY(baseTextures, float3(scaledWorldPos.x, scaledWorldPos.z, textureIndex)) * blendAxes.y;
    float3 zProjection = UNITY_SAMPLE_TEX2DARRAY(baseTextures, float3(scaledWorldPos.x, scaledWorldPos.y, textureIndex)) * blendAxes.z;

    return xProjection + yProjection + zProjection;
}


void surf (Input IN, inout SurfaceOutputStandard o) {
    float heightPercent = inverseLerp(minHeight, maxHeight, IN.worldPos.y);
    float3 blendAxes = abs(IN.worldNormal);
    blendAxes /= blendAxes.x + blendAxes.y + blendAxes.z;

    for (int i = 0; i < layerCount; i++) {
        float drawStrength = inverseLerp(-baseBlends[i] / 2 - epsilon,
            baseBlends[i] / 2,
            heightPercent - baseStartHeights[i]);

        float3 baseColour = baseColours[i] * baseColourStrength[i];
        float3 textureColour = triplanar(IN.worldPos, baseTextureScales[i], blendAxes, i) * (1-baseColourStrength[i]);


        o.Albedo = o.Albedo * (1 - drawStrength) + (baseColour+textureColour) * drawStrength;
    }
```

FIgure 2.8 Triplanar and Surf function

The surf function will loop through the layers and draw the textures based on the highness of the map and assign the textures with a blending parameter which can also blend to a base colour per each layer.

CONCLUSION

Depending on the system on which this project is running it allows for fewer or more chunks to be spawned, it is important to consider that one of the most demanding processes is the object spawner which can be customised in order to either spawn less objects or reduce the LOD = 0 radius. Ultimately, this project could be further improved by implementing different kinds of noises such as Worley Noise to be blended with the Perlin Noise or modifying the shader in order to allow for Normal Maps, Specular and other sorts of textures.