

# Generation of voxel planets

George Kenny

Supervisor: Dr Frank Langbein

April 2020

## **Abstract**

The focus of this report is on the creation of a system that handles the generation of real time voxel planets using Unity Game engine. Voxel planets allow for more interesting terrain and interactivity by a user. Information on the background knowledge is provided to better understand how the system works and why certain decisions were made. The design of the system is explained and in particular talks about why marching cubes are chosen for isosurface extraction and also how noise is used to generate the terrain. In addition, the slightly different design of the level of detail system is explained and mentions why the standard approach of octrees is not used. The implementation was done in Unity and uses compute shaders to allow fast processing on the GPU however it is bottle-necked by the the transfer of data between the CPU and GPU. The project was mildly successful in that the system manages to render planets and maintain a framerate of at least 30 frames per second however the resolution achieved was not what was imagined and the terrain is of a lower quality then expected.

## **Acknowledgements**

I would like to say thank you to my supervisor Dr. Frank Langbein for his support and expertise during this project.

I would also like to thank my family for their continued support throughout my education.

# Contents

<b>Abstract</b>	<b>2</b>
<b>Acknowledgements</b>	<b>3</b>
<b>Table of figures</b>	<b>6</b>
<b>Terminology</b>	<b>7</b>
<b>1 Introduction</b>	<b>8</b>
<b>2 Background</b>	<b>9</b>
2.1 Unity Game Engine . . . . .	9
2.2 Procedural Mesh Generation . . . . .	9
2.3 Procedural Terrain using Noise . . . . .	9
2.3.1 Coherent Noise . . . . .	9
2.4 Voxels . . . . .	10
2.5 Isosurface extraction . . . . .	10
2.5.1 Marching Cubes . . . . .	11
2.5.2 Dual Contouring . . . . .	11
2.6 Level of detail . . . . .	12
2.7 General purpose GPU Programming . . . . .	12
2.7.1 Compute Shaders . . . . .	12
<b>3 Design</b>	<b>14</b>
3.1 System Overview . . . . .	14
3.1.1 System Requirements . . . . .	15
3.2 World representation . . . . .	16
3.3 Terrain and Density generation . . . . .	16
3.3.1 Density Generation . . . . .	16
3.3.2 Terrain Generation . . . . .	16
3.4 Isosurface extraction algorithm . . . . .	17
3.5 Chunk generation and Level of detail . . . . .	17
3.6 Shading . . . . .	17
<b>4 Implementation</b>	<b>19</b>
4.1 Unity . . . . .	19
4.2 Chunks . . . . .	19
4.3 Regular grid / Chunk Generation system . . . . .	21
4.3.1 Chunk Ring . . . . .	22
4.3.2 Chunk coordinates . . . . .	23
4.4 The existing Chunks dictionary . . . . .	23
4.4.1 Removal of chunks . . . . .	23
4.4.2 Addition of chunks . . . . .	23
4.5 Density Generation . . . . .	24
4.6 Marching Cubes . . . . .	25

4.6.1	Resolution . . . . .	26
4.7	Shader . . . . .	26
<b>5</b>	<b>Results and Evaluation</b>	<b>29</b>
5.1	Testing methodology . . . . .	29
5.2	Viewer moving performance . . . . .	29
5.2.1	The test . . . . .	29
5.2.2	The results . . . . .	31
5.2.3	summary of the results . . . . .	32
5.3	Planetary Detail . . . . .	32
5.3.1	Gaps . . . . .	33
5.3.2	Skinny triangles . . . . .	33
5.3.3	Seams . . . . .	33
5.3.4	Noise errors . . . . .	33
<b>6</b>	<b>Future Work</b>	<b>37</b>
<b>7</b>	<b>Conclusions</b>	<b>38</b>
<b>8</b>	<b>Reflection on Learning</b>	<b>39</b>
	<b>References</b>	<b>40</b>

## List of Figures

1	A simplified class diagram of the overall system . . . . .	14
2	The chunk system rendering chunks at different resolutions . . . .	22
3	The first section of the shader which gets the distance from the centre of the planet . . . . .	27
4	The second section of the shader which colours the vertex . . . .	27
5	A low resolution planet with shading . . . . .	28
6	A graph showing the time between frames averaged over the 5 preset resolution with and without level of detail . . . . .	30
7	A wireframe image of a gap in the planet. . . . .	34
8	A wireframe rendering of part the planet detailing lots of small triangles . . . . .	35
9	Seams at different resolutions at the same location on the same planet. . . . .	36

## List of Tables

1	A table showing the framerate and frame times for different start- ing resolutions without level of detail. . . . .	30
2	A table showing the framerate and frame times for different start- ing resolutions with level of detail running. . . . .	30

## Terminology

- Voxel - A singular value on a regular grid.
- Chunk - 16x16x16 voxels. The spacing between voxels is dependant upon the resolution.
- Cube - A single voxel in a chunk
- FPS - the number of frames per second the application is rendering

# 1 Introduction

Many games produced today typically have open worlds or even contain solar systems full of planets. For a lot of these games they just use simple heightmap terrain as they are efficient to manipulate and easy to render however, they are unable to render more complicated terrain features such as overhangs, cliffs and caves. If heightmap terrains are to be edited by the player at runtime then they are only able to create craters or build hills and spikes however it doesn't allow the player to dig or tunnel through it. This interactivity can greatly enhance gameplay and can lead to many unique new forms of content. By instead using voxel terrain, it is possible to create much more interesting terrain and which can also be easily editable by the player.

If the game or interactive content is trying to take place across multiple worlds or planets or even contain a whole solar system, using heightmap terrain is less than ideal, as planetary terrain systems quickly become very complicated. If used as a flat terrain then transitions and loading screens are needed when traveling from the surface of a planet into space. This can ruin the immersion and be frustrating for players. These planets can then be procedurally generated so little artistic input is needed apart from changing some parameters, this would allow for the fast generation of entire galaxies and save a great deal of time during development. By changing a few parameters it can be possible to have all sorts of planets ranging from barren ice worlds to earth like planets. For an artist to do this manually it could take weeks to get the same level of detail which would delay development of games massively.

Rendering these large planets, can quickly lead to performance issues due to the large number of triangles being rendered, so having a way to handle the level of detail at different distances is also a vital component.

In this project I created a system for generating different terrains and rendering them with an isosurface extraction technique giving voxel terrain. The results were that by adjusting the level of detail of the rendering system based on the distance from the planet an adequate level of framerate can be maintained of around 30 fps and this can allow the whole planet to be seen at once. However, the level of detail must be kept quite low unless the viewer is on the surface of the planet and due to transferring data from the CPU and GPU they can sometimes be stuttering when generating parts of a planet.

This report will be discussing the background knowledge needed to understand how a system that can generate planets will work. It will also discuss the design of the system and why that design was chosen, and then will go into the implementation details of various vital aspects giving insight into how they work at a low level. Finally, it will discuss the performance of the overall system and the results of the terrain generation and how successful it has been at generating interesting planets.



## 2 Background

### 2.1 Unity Game Engine

Unity is a game engine that has a wide variety of features, most of which are not needed for this project but its rendering pipeline makes rendering the procedural meshes very simple. This saves lots of development time. Unity objects can have mesh renderer components and mesh filters. These are needed to render meshes with the mesh filter being assigned with a mesh object to render. The mesh renderer contains the material that should be used to render the object.

Unity also contains an editor that allows you to change variables during runtime. This can allow very quick iterating, as it is possible to see the changes live as they happen. Another key feature is the ability to quickly debug scenes visually. A variety of debug geometry can be drawn, which can allow you to see how objects are being rendered. Finally, Unity also have a powerful profiler which makes seeing how fast code is running very easy.

### 2.2 Procedural Mesh Generation

Typically, in 3D graphics and games, meshes are designed beforehand by artists however, this is very costly in both time and money. More and more companies are trying to add procedural content into their workflow to help speed up development. Part of this includes procedural meshes and objects used to populate the scene.

A 3D model or mesh can be represented in multiple ways. In Unity, a mesh is made up of vertices, normals, and triangles which contain three vertices per triangle, and the order of the vertices decide their winding order. The winding order decides which side of a triangle is the front and which is the back. This is important as most rendering engines do not render back faces to save on performance. (Michel 2016)

### 2.3 Procedural Terrain using Noise

#### 2.3.1 Coherent Noise

As planets need to be randomly generated, there needs to be a way to generate these random values which can be used to change the surface of the planet. A random number generator is not suitable for this as the variance does not create realistic looking objects as random objects in nature usually have a smooth looking appearance which random number generators do not provide but coherent noise does (Scratchapixel n.d.). Take, for example, a hill or valley, if we were to just sample values, the likelihood of generating a hill would be very low. Therefore we want a system where small changes in input value will give a small change in output value, in return giving a smooth noise.

Noise, in general, is a series of random numbers, typically arranged in a line or a grid (Patel n.d.). This would not be useful, so to create a natural looking terrain coherent noise is used. Coherent noise is a smooth pseudo-random noise. As stated by (Bevins 2005) coherent noise must have three important properties:

1. The same input value will always return the same output value.
2. A small change in input value will return a small change in output value.
3. A large change in input value will return a random change in output value.

This type of noise is much more useful for terrain generation, as by sampling the noise at different frequency's and layering noise, we can produce interesting terrain features. In the case of terrain when we sample the noise we use it to represent the elevation of the terrain at a point. There are many ways that noise can be used and modified to produce terrain features. These include:

1. The Frequency - by changing the frequency of the noise that we sample we can produce less wider hills or by increasing the frequency we can have more narrow hills. One way to think of this is that the frequency changes the horizontal size of the terrain features whilst amplitude changes the vertical size.
2. The amplitude - This changes the height of the feature. E.g. producing very tall mountains.
3. Octaves - This is adding together noise of different frequencies. This gives the terrain more detail and allows the terrain to have a rugged look, rather than a completely unnatural smooth shape

(Patel n.d.)

Noise can be seeded which means that for a particular seed it will always produce the same output for a input. This means to generate different planets a different seed is needed for each planet.

## 2.4 Voxels

## 2.5 Isosurface extraction

An isosurface is a 3D surface representation of points which have equal values in a 3D data distribution. (*Isosurface* n.d.) In particular they can be used to easily represent a sphere as the base of a planet. For example a sphere can be represented by the following implicit equation:

$$x^2 + y^2 + z^2 = 0$$

To render an isosurface the polygonal mesh first needs to be extracted as real time graphics render piplines require a mesh. There are a variety of isosurface extraction techniques but one of the most popular is marching cubes.

### 2.5.1 Marching Cubes

Marching cubes is an algorithm for extracting polygonal meshes of an isosurface from a discrete scalar field. Although it was developed back in 1987, it and its many variations are still popular today due to its simplicity in implementation and its fast performance. This is partially due to its use of precomputed lookup tables (Bourke 1984).

The algorithm works by splitting the world up into cubes and goes through each cube and determines how the user specified isosurface intersects the cube. To work out the surface intersection for a cube, a one is assigned to a cube's vertex if it is greater than the user specified isolevel. This means that the vertex is inside or on the isosurface. Vertices which are below the isolevel are instead outside the surface. After the cube's vertices have been marked it is then possible to know which edges intersect the surface. If an edge has one vertex inside or on the surface, and its second vertex is outside the surface we then know the edge must intersect the surface. Therefore, a point on this edge must lay on the surface. (William E. Lorensen 1987)

Each cube has 8 vertices and each vertex can either be inside or outside the surface which means there are 256 different ways a surface can intersect the cube. Rather than manually calculating surface-edge intersections a lookup table is used. This means that the case number is an 8-bit index where each bit represents one of the vertices. This case number is used to index into the lookup table to get which edges intersect the surface. Case 0 and case 255 can be ignored as this means either all the vertices are inside or outside the surface so no geometry needs to be produced. (William E. Lorensen 1987)

Once index into the lookup table it is possible to get through each edge intersecting the surface and use linear interpolation to get the point along that edge that intersects the surface. Whilst it is possible to use higher levels of interpolation, using linear interpolation produces good results and is easy to compute. (William E. Lorensen 1987)

Bourke 1984 provides a triangulation table which is commonly used in mainly marching cubes implementations. It is a 256x16 array which contains which edges are intersecting the surface in each case with the rest of each case being padded out with negative one. The edges are also ordered so that when they are triangulated the triangles have the correct winding order. This makes implementation very fast and relatively simple.

### 2.5.2 Dual Contouring

Dual contouring is similar to marching cubes in that it is an isosurface extraction technique however it contains a few differences such as being able to render sharp corners and edges. This allows for a more accurate representation of an isosurface. As it is a dual method it takes a different approach by producing vertices on the surface interior to each of the cubes in the grid rather than on

the edges like marching cubes. Another advantage is that it produces polygons of more uniform size. Methods like marching cubes can also sometimes produce small polygons which can produce visual distortion. (Scott Schaefer 1987)

However to obtain the improved detail, more information needs to be known about the function that is to be rendered. The value of the function must be known the same as marching cubes but the gradient of the function must also be known. The extra calculations needed can decrease performance which when rendering a large scene can have a large impact. (Boris 2018)

## 2.6 Level of detail

Level of detail, also known as LOD, is a way of decreasing how complicated a 3D model is as it moves away from the viewer. This typically involves reducing the number of vertices of the model. Although there are many ways to accomplish this, most algorithms are more suited for pre-existing models. Using these algorithms for procedural meshes would end up in worse performance so it is better to generate the mesh at a lower resolution therefore reducing its complexity. (DanielG.Aliaga 2010)

The lower resolution meshes difference in quality is typically hardly noticeable due to its extended distance from the viewer. The benefit of doing this however, is that it is possible to fit more geometry on screen and keep a high framerate due to less vertices per model. In terms of voxel rendering it also means less chunks need to be generated due to the increased size of a lower resolution chunk. This in turn decreases the time it takes to generate.

## 2.7 General purpose GPU Programming

Graphics cards are typically used for rendering images which are output to a display device such as a monitor. They can run complex graphics pipelines part of which include shaders which are programs used for shading 3D scenes. The most commonly used shaders are vertex and fragment shaders. Vertex shaders are run once for each vertex and fragment shaders are used to compute attributes for fragments. Fragments are a unit of rendering affecting at most a single pixel. Although, these shaders are very powerful when it comes to rendering they are also limited to only graphics applications.

Due to the parallel nature of graphics cards, they can be also extremely fast for non graphics applications. One way in which this can be done is compute shaders.

### 2.7.1 Compute Shaders

Compute shaders are a variant of shaders which can accept a variety of data types compared to vertex and fragment shaders which require certain data types. There are a few different types of buffers that compute shaders can accept some of which include appendable buffers and structured buffers. This

gives compute shaders lots of flexibility to work on a variety of problems even ones not related to rendering. The benefit of running these generalised problems on the GPU is that the problem or algorithm then becomes parallelised which boosts performance considerably. The only downside to this is that sending data from the CPU to the GPU and vice versa can be very slow so depending on the problem there isn't always a guarantee of better performance. If the work being done is for rendering though such as the case for marching cubes it is possible to send the vertex or triangle data directly to a vertex/fragment shader which can save time and also increase performance.

Compute shaders are run on groups of threads and each group has a set amount of threads that is specified when writing them. If the number of threads per group is not set with care it can lead to very poor performance due to the GPU not using very many threads and having to wait for threads to finish. When specifying the number of groups and the number of threads you have to specify the number in each dimension up to three dimensions. For example you could have (8x8x8) threads which would be 8 in each dimension. The advantage of this is that it can help map threads to problem and can also help with easier indexing. For example in terms of marching cubes by using 8 threads in each dimension, each thread can be used to calculate the vertices for one block in a three dimensional chunk.

### 3 Design

This section is about how the overall system fits together, what each component of the system does and why each component was chosen for the system.

#### 3.1 System Overview

Figure 1: A simplified class diagram of the overall system

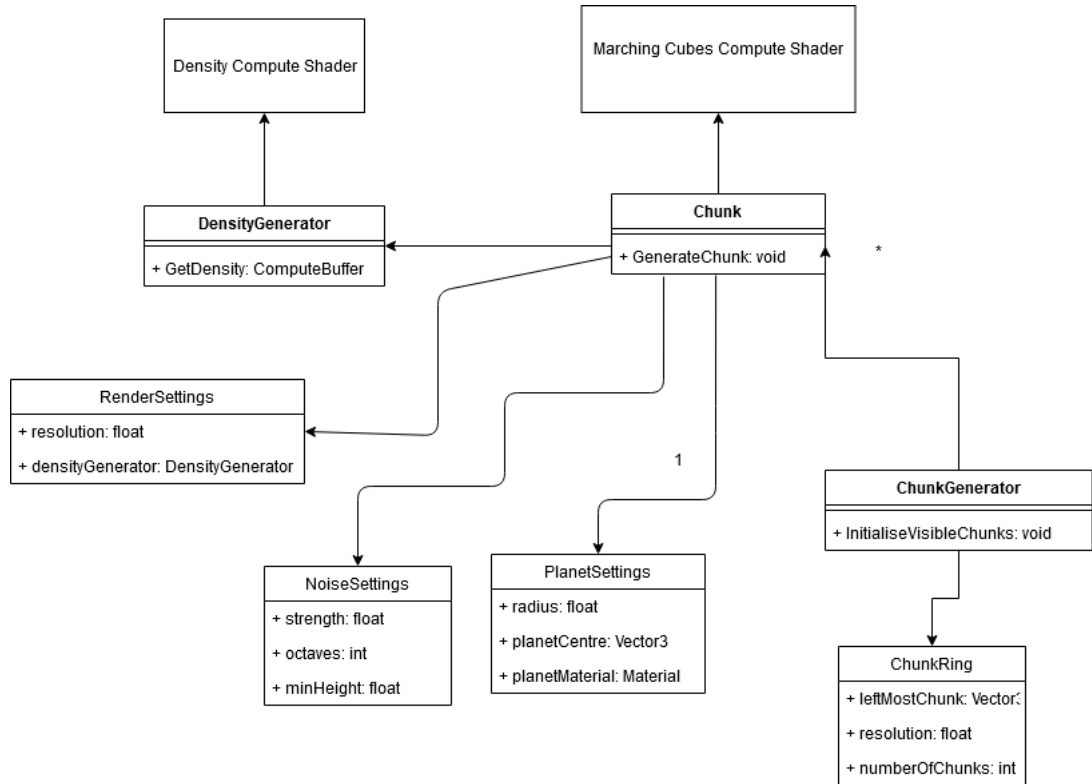


Figure 1 shows the simplified class structure of the system. The chunk class is used to create the individual chunks and generate their mesh. It does this using the marching cubes compute shader. The chunk generator is responsible for the level of detail system and generating the chunks at the correct resolution and position. The settings classes are passed between the other classes and are used as an easy way to store the settings as there are a large number of parameters.

To reiterate the goals of the system are to generate planets for real-time rendering which are rendered with a voxel system so that that can be edited and also contain more interesting terrain features than standard heightmap terrain.

The system also need needs a level of detail system to render the planets in real time.

The system can be split into a few main components:

- Terrain/Density generation
- Isosurface extraction
- Chunk Generation
- Level of detail system
- Shading

Each component in the system requires the previous to have completed its step before the next component can begin its tasks.

This section will detail how each component will be accomplished and it will go through what methods were tried and finally which methods were picked and why.

The system has some requirements that it needs to fulfil to be successful. They can be split into two main categories: terrain generation and voxel rendering.

### **3.1.1 System Requirements**

For the terrain generation of the planets the requirements are:

- The planet must be generated in real-time rather than pre-computed
- The terrain must have different variations such as mountains, valleys, plains etc.
- The terrain should be textured and coloured based on the height of terrain from the centre of the planet to create a realistic look.

For the voxel rendering of the planets the requirements are:

- The view distance should be large enough at all distances so that there are not gaps in the world seen by the viewer.
- The average FPS should maintain at least 60 to ensure smooth game play.
- A level of detail system should be used to increase the detail of the planet when the viewer is close to the planet.
- The level of detail system should be able to render the entire planet when the viewer is far away.
- The terrain should be smooth shaded.

## 3.2 World representation

It was decided that the world would be split into a regular grid, with each 16x16x16 voxel region being known as a chunk. The advantage of this was it allowed the rendering to split done in parallel therefore giving better performance. It also made the level of detail system easier to develop and meant that it was easier to decide to what should be rendered as where.

## 3.3 Terrain and Density generation

### 3.3.1 Density Generation

To generate a terrain mesh, the chosen isosurface extraction technique needs to be provided with density values. The density values could be precomputed and saved to disk and then loaded when needed for when a chunk needs to be rendered. This approach however can be tricky to get right due needing to read the data quickly from disk so that it doesn't become a bottleneck on the system. The necessary performance can typically be achieved by using compression however due to the time constraints of the project this was deemed to be a more complicated approach that may have used too much time.

Due to this it was decided that the terrain and therefore densities for the terrain would be generated at run time. To do this the density generation will be run on a compute shader as the density data can be stored in a compute buffer which can then be sent to the isosurface extraction compute shader. As the compute buffer will already be on the GPU no time has to be wasted in transferring the data. It was originally trialed on the CPU however transferring the data to the GPU was particularly slow and caused major performance issues.

### 3.3.2 Terrain Generation

The density generation technique allows anything to be rendered however for planets some thought was needed to decide the best strategy for calculating density values that would give interesting terrain.

Before the terrain can be calculated a sphere is needed to represent a planet. One method that was considered was working in cube space or in other words using 6 faces of a cube and then mapping this cube to a sphere. One advantage of this is the easy generation of a navigation mesh. (Zackary P. T. Sin n.d.) Instead of this more complex method, I generated the density function for a chunk by calculating the implicit equation of a sphere. This isn't perfect due to rendering small triangles but it was quick to implement.

To modify the sphere to create interesting terrain for the planet, OpenSimplexNoise was chosen over Perlin noise due to it being easier to extend to three dimensions and it also being more efficient. Three dimensional noise was needed as it better mapped to the sphere and also allowed for the more interesting terrain features which 2D noise wouldn't have been able to do.



### 3.4 Isosurface extraction algorithm

Marching cubes was the chosen isosurface algorithm over dual contouring due to marching cubes being easy and fast to implement and also relatively simple to understand. This means a working prototype could quickly be developed and more time could be spent on optimizing the rendering to increase performance for large planets. Marching cubes is also easy to run on a compute shader as the algorithm is very easy to parallelise due to the algorithm running once for each cube.

The downside of using marching cubes is that it cannot properly represent sharp edges and corners such as rectangles and cuboids. This can lead to terrain looking too smooth and not natural. However the ability to quickly implement the algorithm outweighs this as by using the right combination of noise it should be hard to notice the lack of sharp edges. (Boris 2018)

Using marching cubes helps to fulfil the requirements for the project as it has better performance when compared to dual contouring which is vital when trying to maintain a good framerate. This will ensure the viewer has a smooth experience which will help to create immersion.

### 3.5 Chunk generation and Level of detail

One method for level of detail was to just generate and render chunks for the entire planet and then scale the resolution based on the distance from the viewer. This could have been done using an octree. This wasn't very efficient and as planets got larger generating all the terrain for all the chunks began to get very slow. The one upside of this is that the viewer could always easily see the planet.

I eventually moved to a system where only visible chunks around the viewer are generated and the chunks closer to the viewer will be at a higher resolution and as they get further from the viewer they will get progressively lower resolution. The downside of this is that the resolution of the chunks has to be balanced with the distance the player is from the planet to ensure the planet could be seen from extreme distances.

### 3.6 Shading

To make the planet seem realistic and better quality I chose to use a simple surface shader that coloured and textured the terrain based on its height from the radius of the planet. As the maximum height of the terrain is unknown due to only part of the planet being generated at once it is up to the user of the system to set the maximum height of the terrain of the planet. Based on this it is then possible to get distance of each vertex as a percentage with zero percent being at sea level, which is also the radius of the planet, and one hundred percent being at the maximum height. The user also needs to set a gradient of colours which is done within the unity inspector. By sampling this gradient the vertices colour is then set. Although, this is quite a simple approach and doesn't allow

the textures it still gives enough information about sort of terrain would be at this point on the planet.

The alternative would be to use a triplanar shader which would allow textures to be used as there would no longer be any distortion. A triplanar shader is needed for textures as it would not be possible to UV unwrap the procedural mesh which is needed to stop the textures from stretching. Creating a shader like this is more complicated and would have taken a lot longer so I instead went with the simple colour approach to allow more time to be spent on the rendering system.

## 4 Implementation

This section is going to talk about each system from the design section has been implemented and what issues came up during implementation of these features.

### 4.1 Unity

All implementation has been done in the Unity Game Engine. The reason for this is that Unity allows rapid prototyping without having to handle all aspects of rendering. For example it has a built in mesh class that requires vertices, normals and triangle indices but when provided with these the mesh can be passed to a mesh renderer and the game engine handles the rest. If this was developed from scratch it would require a lot of low level rendering code which would have slowed development and it would have taken focus away from the project as time would have had to of been spent implementing basic features such as lighting.

Some alternatives to Unity could have been writing a custom engine to handle rendering, lighting, and other key features my self however this approach was deemed to be not a viable option due to how long it would have taken to get the basic rendering framework working and that could have instead been spent on the project. By using Unity it means I do not have to worry about key features such as rendering containing bugs which could have easily arisen with a custom solution and this could have led to issues with not knowing whether planet system or if the basic renderer was causing the bug.

### 4.2 Chunks

The chunk class is used to represent a chunk and its data and also constructs the mesh from the marching cubes compute shader. It contains an Initialise method which is used to set the planet settings, noise settings and voxel render settings all of which are needed by the class to generate the mesh for this chunk. The reason a method is used rather than the constructor is that sometimes the chunk needs to be regenerated with new data so its an easy way to set all the data.

The class also contains a "GeneratePlanet" method which is responsible for setting up the marching cubes compute shader and running it. The compute shader has an appendable triangle buffer and this triangle buffer is then copied to an array of the type triangle. The triangle struct just contains three Vector3's and looks like this:

```
public struct Triangle
{
    public Vector3 A;
    public Vector3 B;
    public Vector3 C;
}
```

Now that the array of triangles is obtained the mesh is constructed but duplicate vertices have to be accounted for so that when the mesh's normals are calculated they take into account the neighbouring vertices otherwise you end up with non smooth shading.

```
Dictionary<Vector3, int> vertexIndex = new Dictionary<Vector3, int>();
foreach (Triangle t in triangles)
{
    if (!vertexIndex.ContainsKey(t.A))
    {
        vertexIndex.Add(t.A, verts.Count);
        verts.Add(t.A);
        tris.Add(verts.Count - 1);
    }
    else
    {
        tris.Add(vertexIndex[t.A]);
    }

    if (!vertexIndex.ContainsKey(t.B))
    {
        vertexIndex.Add(t.B, verts.Count);
        verts.Add(t.B);
        tris.Add(verts.Count - 1);
    }
    else
    {
        tris.Add(vertexIndex[t.B]);
    }

    if (!vertexIndex.ContainsKey(t.C))
    {
        vertexIndex.Add(t.C, verts.Count);
        verts.Add(t.C);
        tris.Add(verts.Count - 1);
    }
    else
    {
        tris.Add(vertexIndex[t.C]);
    }
}
```

This snippet creates the mesh with smooth normals by creating a dictionary of vertices which stores the index of each vertex that is in the triangle array. It

loops through each triangle and checks to see if the dictionary already contains the vertex if it doesn't, it adds it to dictionary and the value is set to the length of vertex array as that's the vertex's position. The vertex is then added to the vertex list and the triangle index list adds the vertex position.

If the dictionary does already contain vertex then there is no need to add the vertex to the vertex list or the dictionary and instead vertex index position stored in the dictionary is added to the triangle index list. These steps are repeated for the other two vertices in the triangle.

A new mesh object is made and the vertices and triangles of the mesh are set to the corresponding lists. Finally the mesh normals are recalculated this is done using the built in unity method however better performance could be gained by doing this in the marching cubes compute shader.

### 4.3 Regular grid / Chunk Generation system

The world is made up of a regular grid of chunks and each chunk is 16 cubes x 16 cubes x 16 cubes. This gives a chunk size of 4096 cubes however the size of a chunk's cubes depend on the resolution of the chunk. A higher resolution means the chunks are physically smaller. As generating chunks is expensive the number of chunks generated must be limited to maintain good performance. This means that only chunks that are around the player are generated as there is no point generating something player might not see. The only downside to this is that the player will have a limited view distance and the distance will have to be balanced to maintain a good framerate.

The chunk generation system generates chunks at different levels of detail around viewer depending upon distance to the planet. This gives a good combination of detail and performance.

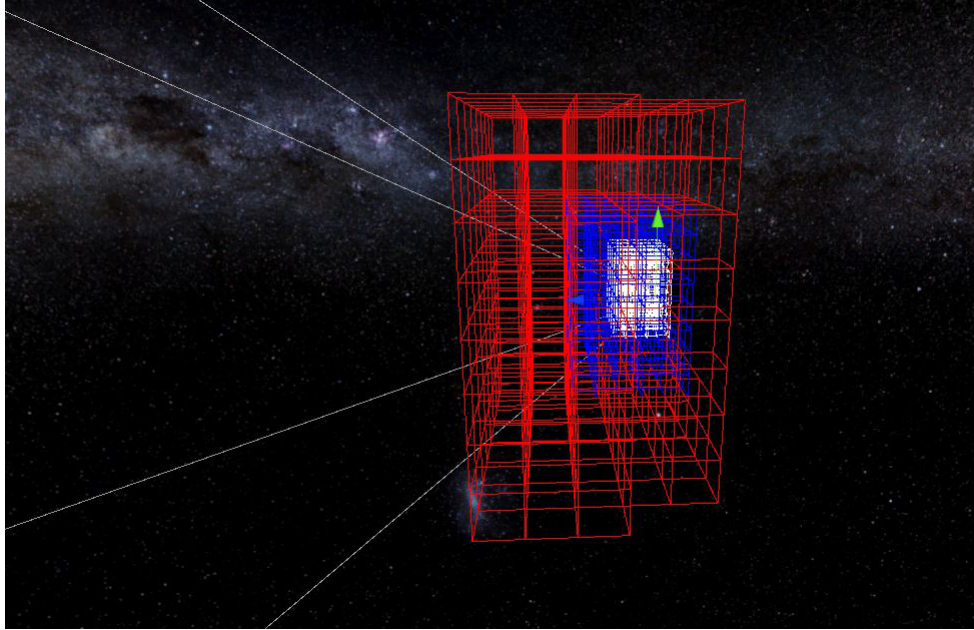
Unity scripts contain an update method that are called once per frame. In the update method the "InitVisibleChunks" method is called which is responsible to rendering the chunks surrounding the viewer.

This method works by going through all existing chunks and removing all chunks that are not within the camera frustum as none of these chunks will be visible so there is no need to keep them generated.

Next the chunks are generated. Chunks are only generated along the perimeter of a cube as the inside of the cube contains the chunks at a lower resolution. The "InitChunkRing" method is responsible to generating the chunks at a particular resolution. The method also takes in a "ChunkRing" object in its parameter. The class contains information on the previous chunk ring. This data is needed to be able to generate the chunks at the correct location.

```
public class ChunkRing
{
    public Vector3 leftMostChunk;
```

Figure 2: The chunk system rendering chunks at different resolutions



```
    public float resolution;  
    public int numberOfChunks;  
}
```

#### 4.3.1 Chunk Ring

The "initChunkRing" method works by first calculating the start position for this ring in chunk coordinates. This is calculated by taking the previous chunk position and then removing (one \* resolution) from this position. This then gives the bottom left most coordinate of the chunk ring. Using three nested for loops the method then calculates the current chunk position by adding creating a vector equal to the for loop values i.e. new Vector3(x,y,z) where x,y,z are the for loop values. The values of the new vector are also multiplied by the resolution to ensure the correct spacing. As only chunks on the perimeter of the imaginary cube should be generated the method checks if x,y,z are equal to zero or the width of this ring minus one. If any of them are equal to one of these values it means that this chunk is on the perimeter of the cube.

If it is on the perimeter we then have to check if a chunk currently exists in this position using a dictionary that stores all the chunks. If one does exist in this location we then also check if the resolution of that chunk is the same and if it is we can skip past this chunk as it is the same. If the resolution is different however we then need to make sure we remove all that chunk from the

dictionary as it is the wrong resolution.

If no chunk currently exists at the position we are creating the new one and the resolution is greater than the starting resolution aka the highest resolution we then need to make sure we check every chunk position inside this chunk as when it is being created it may be overlapping an old one so we need to remove all old chunks that is overlapping.

Finally, we check if the chunk is in the camera frustum before we create it and we also remove all the chunk positions that this chunk overlaps to the dictionary.

#### 4.3.2 Chunk coordinates

Chunks all have their origin at the bottom left of their corresponding mesh. To help make calculations easier a chunk coordinate system is used this is calculated by:

```
private Vector3 WorldPosToChunkPos(Vector3 worldPos)
{
    return worldPos / minimumBoundsSize;
}
```

MinimumBoundsSize is the width of the highest resolution chunks. This is because lower resolution chunks will need a larger world space size and will be a multiple of the highest resolution chunk.

### 4.4 The existing Chunks dictionary

All chunks generated are stored in a dictionary with the key being the chunks position and the value being the chunk object. If a chunk is a lower resolution than the highest resolution chunk then multiple positions are stored in the dictionary for that chunk. This allows overlaps to be quickly checked. The benefit of using a dictionary is the fast lookup speeds.

#### 4.4.1 Removal of chunks

Care has to be taken when removing chunks. The gameobject itself has to be deleted from the scene but for a chunk with a resolution lower than the highest resolution chunk, all of the chunks positions have to be removed from the dictionary if not it can cause errors later on when creating new chunks due to already existing dictionary keys.

#### 4.4.2 Addition of chunks

When adding chunks if the resolution of the chunk is greater than the highest resolution chunk then a position needs to be added into the existing chunks dictionary for each chunk position that this chunk occupies. For example if a chunk is 2x the base resolution then it will take up twice as many chunk

positions in each dimension so eight positions will have to be entered. Then the chunk can be instantiated into the world.

## 4.5 Density Generation

The density values are generated on the GPU in a compute shader. It uses a simplex noise implementation for the GPU as well. Each chunk requests a buffer of density values from the "Density Generator" which is assigned before run time. The density generator sets up the buffers and variables needed for the compute shader and then dispatches the shader. One important variable that is set up is the number of thread groups. This decides how many thread groups should be created for the shader. Each thread group contains 512 threads as this is the limit for the hardware used to test the system when the thread group is run in a 3D configuration. This means  $X$  thread groups are needed where

$$X = \frac{width + 1}{8}$$

The density buffer is for each chunk is 17x17x17 as for  $x$  voxels you need  $(x+1)$  density values in each dimension for marching cubes to run.

The compute shader works by first calculating the density value for a sphere at the set planet centre and resolution. Another one of the parameters is the world position of the chunk. These world values are needed to calculate the correct values for each vertex of this chunk. The shader then creates a elevation variable which is used to offset the sphere terrain densities. A loop then evaluates the simplex noise function once for each octave and adds its value to the elevation. The noise function was edited to be in the 0,1 range rather than  $-1,1$ . The elevation value is then set so if it is below the minimum height parameter it is set to 0. This is done by

$$elevation = \max(0, elevation - minHeight)$$

. In this implementation negative density values are inside the isosurface whilst positive are outside the isosurface so we then negate the elevation from the sphere density. This then gives us a final density value for this vertex and this is added to the density buffer.

When writing to the density buffer the index must be calculated from thread ID. The thread ID is a 3 dimensional vector as this makes it easier to relate 3D world space density values to threads. As the buffer is a 1 dimensional structure we calculate the index for the buffer as:

```
int flattenedIndex(int x, int y, int z)
{
    return (x * (densityWidth) * (densityWidth))
    + (y * (densityWidth))
    + z;
}
```



## 4.6 Marching Cubes

The marching cubes algorithm is also run on a compute shader due to the increased performance benefits. As the algorithm goes through each cube and calculates the resultant mesh for that cube, it makes it very suitable for running in parallel on the GPU. To make indexing easier the compute shader's threads are also set to (8,8,8) in each direction giving a total of 512 threads per group. As chunk sizes are 16x16x16 voxels two threads group perfectly calculate one chunk however as the chunk size can be variable the shader starts off with a check to return if the thread id is greater than the chunk width to make sure unnecessary triangles are not calculated.

As stated in the background section of marching cubes the algorithm marches through each cube and decides which vertices to need to be rendered for that cube. This means that the compute shader then gets the 8 density values for that cube from the density buffer and puts them into a float array.

After this it goes through each cube corner and checks whether it is greater than the isolevel parameter. In practise the isolevel is set to 0 but can be changed to whatever suits the project. As also stated in the background section an eight bit index is created with one bit representing each corner of the cube and each bits value is dependant on whether the corner is in or outside the surface.

Using the cube index, Bourke 1984's triangulation table is used to get which edges have a vertex laying on them for the current index. The edges ordered so that the vertices calculated from the triangulation table have the correct winding order and therefore the polygon's normals face the correct direction.

The vertices for each edge then need to be obtained so that it's possible to linearly interpolate between the two to get the correct position for the output vertex for that edge. Bourke 1984 also provides an edge table however a simpler implementation is used which is provided by Lague 2019. The edge table requires many if statements to work and although the shader compiler should be able to optimise these if statements to avoid branching which can massively decrease performance of the shader, the overall implementation is more complicated. Lague 2019's method instead uses two tables which can one of the edge 12 cube edges in as an index. The first table gives the index of one vertex whilst the second table gives the index of the second vertex of that edge. There is then a third table which contains the vertex positions in object space. This is indexed into using the vertex index. Now that the two vertices for each edge has been obtained it is then possible to linearly interpolate between the density values for these vertices to obtain the vertex position closest to a density value of 0 on that edge.

Three edges vertices are calculated at a time and a triangle struct is created and the three vertices are assigned to the triangle struct. This is then added to the triangle buffer. The reason a triangle buffer is used rather than a vertex buffer is because when an append buffer is used when three vertices are written by a

thread to the buffer it is possible for another thread to also write to the buffer in between vertices. This then messes up the vertex order so it is not possible to construct the mesh so by writing each triangle at once instead it doesn't matter which order the triangles are written to the buffer. One alternative to this is to use a structured buffer and write the vertex into the correct index for that cube however when constructing the mesh you then have to filter through vertices with with a default value.

Another issue that appeared when implementing this on the compute shader was that originally the indexing was wrong due to forgetting that there are width + 1 density values in each direction. This was easy to fix though by just remembering to add one to the width when calculating the flattened index.

#### 4.6.1 Resolution

Due to needing a level of detail system for rendering the meshes must also be rendered at different resolutions. This is done by keeping the chunks number of blocks constant and instead changing the width of each block. This means higher resolution chunks are smaller in world size. This is done by parsing the compute shader the resolution for that chunk and when interpolating the vertices they are multiplied by the resolution. This means that a chunk with a higher resolution is actually less detailed. This can quite easily be fixed by instead dividing by the resolution.

### 4.7 Shader

The shader is created using Unity's shader graph which is a visual graph tool for making shader's. This allows you to focus on the functionality of the shader rather than the syntax.

Figure 3 and figure 4 show the completed shader. It gets the vertex in world coordinates and then calculates the distance of that vertex from the centre of the planet. After this it divides the distance by the max height and then samples the gradient texture using the height of the vertex which is now in the 0-1 range. The colour from the gradient texture is then set as the output colour.

Figure 5 shows the result of shader. A variety of terrain colours can be seen on the planet such as blue for the water and green for the grass.

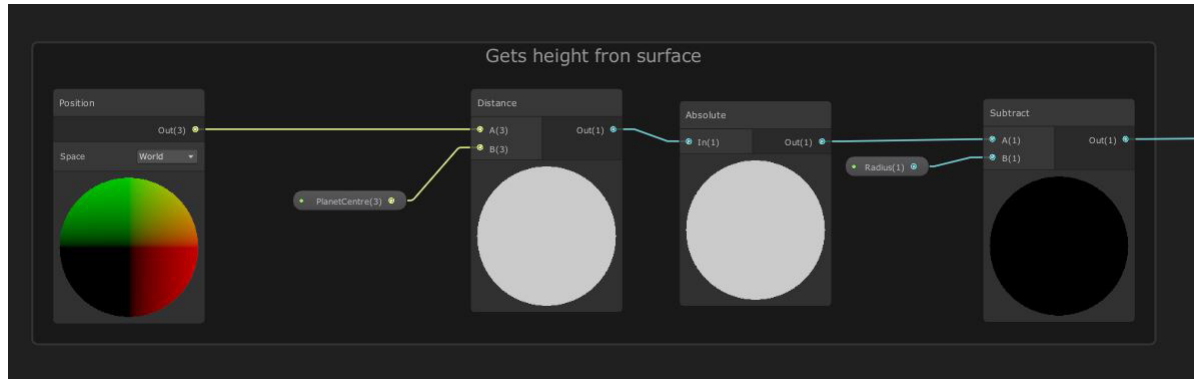


Figure 3: The first section of the shader which gets the distance from the centre of the planet

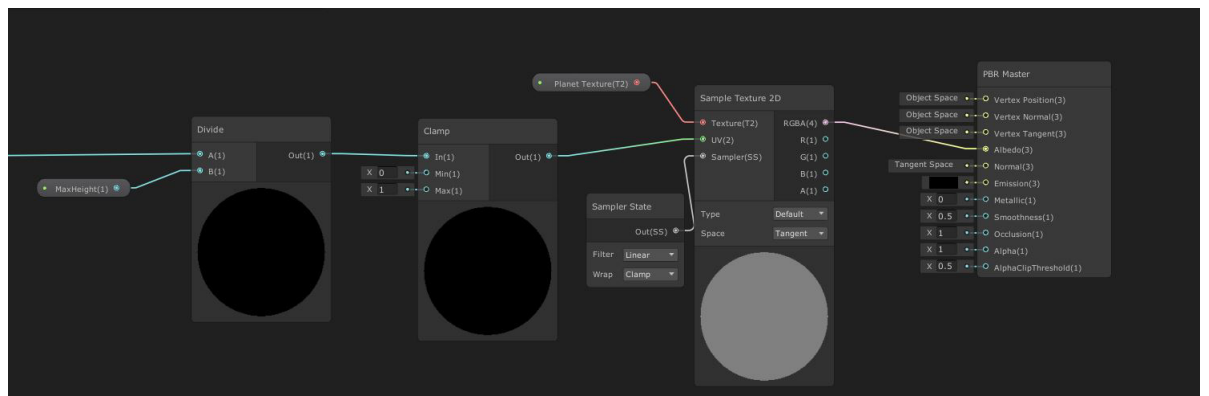


Figure 4: The second section of the shader which colours the vertex

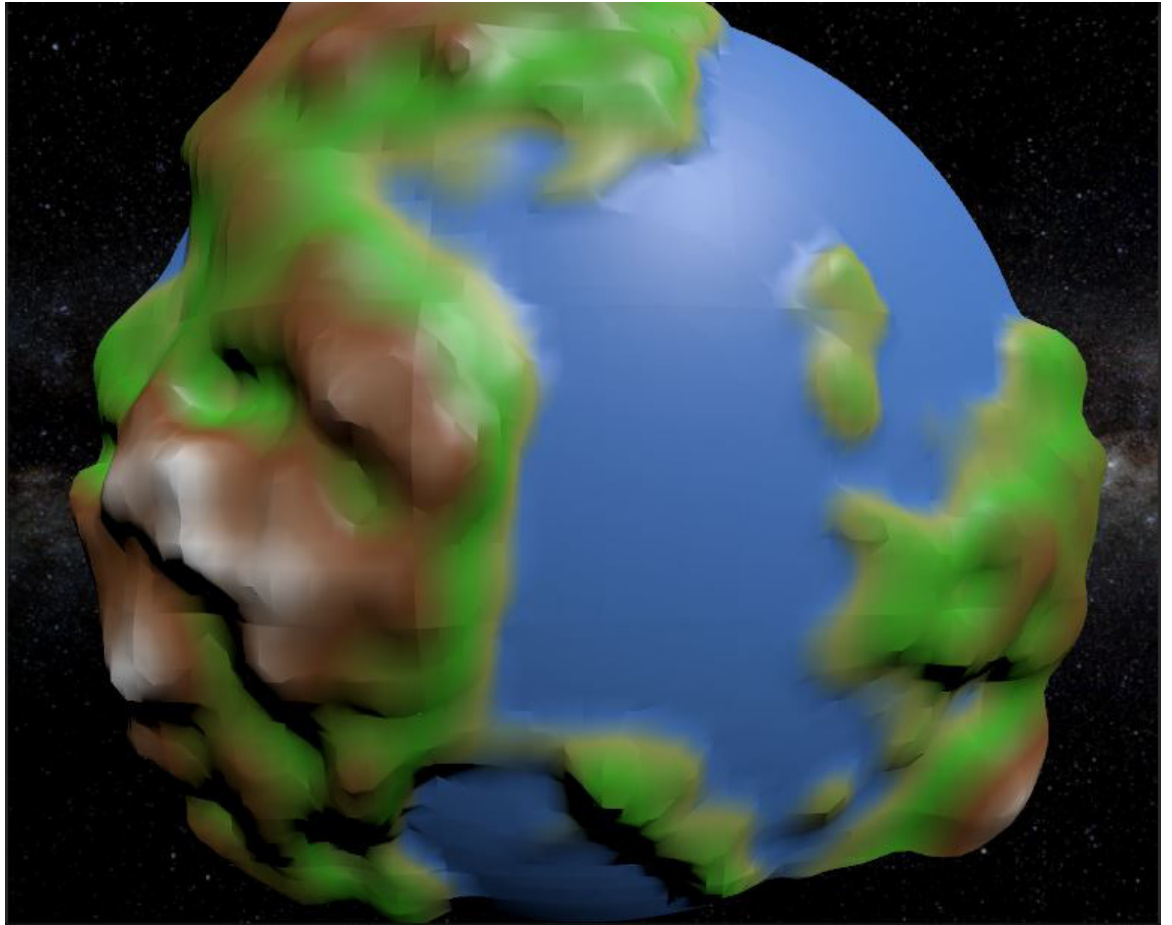


Figure 5: A low resolution planet with shading

## 5 Results and Evaluation

This section discusses what tests have been carried out and why. It also shows how successful the project has been at hitting its requirements and shows what aspects were not accomplished.

### 5.1 Testing methodology

All testing takes place on the same planet that is generated from the same seed. The planet is 50,000 metres in radius and uses 7 octaves of noise. This ensures the testing is fair as the geometry of a planet can vary quite wildly which could lead to very different results.

Due to time constraints the tests are run in the Unity editor these can lead to slightly lower framerates than if they were run in a built version of the project. This is because the editor has some overhead costs. The testing also has some overhead such as writing the data to file which can potentially hold up the application. To try and counter any variance each test was repeated five times and averaged.

### 5.2 Viewer moving performance

This sub-section is evaluating the performance of the overall system when the viewer is moving. It is also evaluating the effect the level of detail system has on the framerate.

#### 5.2.1 The test

This was tested by placing the camera 5,000 units away from the surface of planet and have the camera rotate around the planet at a constant speed. As the camera moved it also rotated to point towards the centre of the planet. This ensured the planet was always in view of the camera. Each test varied the resolution, which ranged from 8-128. The higher the resolution value the less detailed the terrain was due to how resolution is handled internally. Due to the resolution changing the world size of each chunk, the view distance also changed. By having a higher resolution value there was also a greater view distance but by keeping the camera close to the terrain a similar area of the planet was rendered for each test.

For the tests where the level of detail system was on the chunk rings consisted of 4 chunks of the base resolution, 2 chunks of  $2 * baseresolution$  and 1 chunk of  $4 * baseresolution$ .

For some of the smaller tests such as resolution 8 and 16 sometimes no geometry was rendered due to being too far from the surface of the planet but the chunks surrounding the camera were still generated.

Resolution	8	16	32	64	128
Min FPS	10	10	10	10	10
Average FPS	10.04	10.27	11.47	14.94	17.36
Max FPS	13.26	20.15	21.26	22.70	24.81
Average Frame Time (s)	0.09959	0.09806	0.0900	0.07089	0.06028

Table 1: A table showing the framerate and frame times for different starting resolutions without level of detail.

Resolution	8	16	32	64	128
Min FPS	10	10	10	10	10
Average FPS	11.06	11.16	12.01	20.89	31.84
Max FPS	42.85	43.18	30.11	36.57	43.23
Average Frame Time (s)	0.09664	0.09651	0.08713	0.05725	0.03506

Table 2: A table showing the framerate and frame times for different starting resolutions with level of detail running.

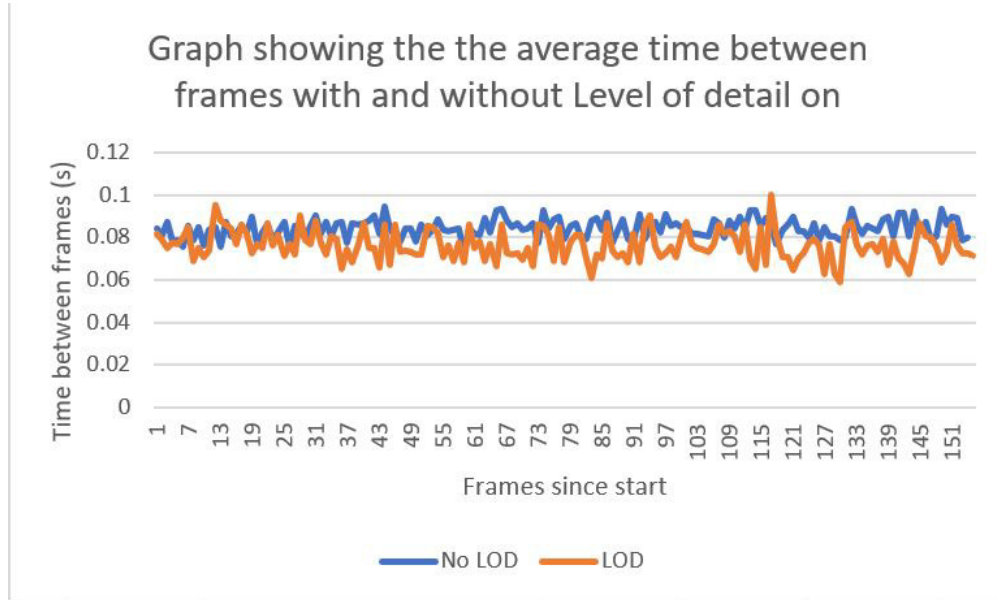


Figure 6: A graph showing the time between frames averaged over the 5 preset resolution with and without level of detail

### 5.2.2 The results

Table 1 shows the results with the level of detail system turned off. This means all chunks for a particular test had the same resolution. The results from this table show that the minimum framerate for all tests were 10, this is due to the time it would take to generate the all the chunks. When generating them the worst case would be 0.1s in between frames and would take 10 frames to generate these chunks and therefore giving a minimum frame time is 10. The way to increase this would be to speed up how long it takes to generate all the chunks such as not passing data back to the GPU.

These results also show as the resolution increases so does the average framerate, this is most likely due to not have to as many chunks to generate due to their increased world size. There saves time transferring data between the CPU and GPU and also by having a higher resolution less triangles will be on screen meaning the GPU has do less work. The same also applies to the max framerate. The average FPS for resolutions 8 and 16 are quite similar and its only when the resolution starts to increase even more that a noticeable difference starts to appear. The average framerates are still very low across all the tests but this could be due partly to the overhead from the editor but also shows that overall the system needs more optimising.

The average frame time also decreases as the resolution increases which is what we would expect to see. To render at 60 frames per second, which is a value typically used as a goal in most commercial games each frame must take no longer than 0.016667 seconds and from the results we can see we are still at times nearly 6x slower than that which is far below the set out requirements. (Schliesser 2018)

Table 2 shows the same tests repeated but this time with the level of detail system turned on. It shows the same patterns as table 1 however the average FPS are higher for each resolution and the average frame times are lower as the resolutions increase compared to when level of detail is turned off. This means that by using the level of detail system we gain a much higher average framerate and also in turn gain a much further view distance. The disadvantage of this is that terrain that is further away will look less detailed but as it is further from the viewer it is hard to notice. If it does become a problem there are ways to hide some of the differences by using fog to blur the less detailed terrain.

To further support the increase in performance from the level of detail system figure 6 shows the averaged times between frames over all the tests. As can be seen from the graph the tests with the level of detail system on have a lower time between frames which indicates a higher framerate. The peaks on the graph show where large amounts of chunks had to be generated which increase the time between frames.

### 5.2.3 summary of the results

These results show that although the system doesn't meet all the requirements laid out in the design section such averaging 60 frames per second it is still possible to move the viewer around the world and maintain at least 30 frames per second with the level of detail system on. This allows the viewer to see large amounts of the planet and to even view the planet out into space.

The disadvantages of this are that the resolutions needed to have at least 30 frames per second as very high this means the terrain is not very detailed and doesn't look particularly good. Therefore, the system needs to be optimised more or changed to allow a more detailed terrain at a higher framerate.

In these tests the number of chunks rings were fixed and when the level of detail system was on, only combination of resolution multipliers were tested. One thing that could help performance is having less detailed chunks around the player as there are currently 4 rings of detailed chunks. This could be decreased. Further testing would be needed to see the impact this would have and whether this would mean that the system could achieve higher framerates with more detail.

## 5.3 Planetary Detail

In this section I will be evaluating the results of the terrain and how detailed the planets are. I will also highlight any artifacts that are generated.

As seen in figure 5 an entire planet can be seen at a fairly low resolution. The terrain is quite basic and doesn't feature some of the more advanced requirements such as caves, overhangs and cliffs however due to time constraints it was more important to focus on the rendering speed rather than the quality. Basic mountains and ocean can be seen and the colours of the terrain try to represent the type of terrain that it is. For example green for grass, blue for ocean etc. I believe this gives a good representation of a planet in a non-photorealistic representation. This could be improved later by adding textures and increasing the variety of terrain features.

One area that could be improved the most is oceans, why now they are smooth, flat surfaces which only contain a colour that make it hard to notice that they are oceans. Instead they could be replaced with a custom water shader which would allow waves and some distortion, this better represent an ocean.

From looking at the planet at a distance, the planet some terrain features can seem overly large however this can be tackled by editing the parameters which affect the noise strength and the number of octaves. The use of different resolutions can also vastly affect how the planet looks. Some of this could be solved by using an atmospheric shader which would hide some of the lower resolution terrain until the viewer was closer to the surface.

Although planets can seem good at various angles and distances there are also some rendering issues. The sub-sections below will detail the what the issues



are and how they have arisen.

### 5.3.1 Gaps

Due to the level of detail system, when neighbouring chunks are at different resolutions there can be gaps between the meshes. This is because the different resolution chunks have a different number of vertices at the border and therefore when the noise is sampled and applied to the vertices they no longer match up due to the sampling rate being different.

In figure 7 two neighbouring chunks of different resolutions can be seen. With the chunk on the left being at a higher resolution and therefore having more sample points over a smaller area. Due to this the meshes will not be able to fully close and a modified version of marching cubes will be needed to fix this. An alternative would be to try and stitch the gaps together by iterating over by edges. Lengyl 2010 proposes a modified marching cubes solution that contains many more cases and uses data from neighbouring chunks to fill the gaps.

### 5.3.2 Skinny triangles

As shown in figure 8 marching cubes can produce lots of small skinny triangles when triangulating an isosurface. Although this shouldn't effect it the visual quality it can impact performance quite heavily due to the unnecessary extra triangles which could instead be represent by one of the surrounding larger triangles. This puts a greater strain on the GPU and can lead to lower framerates and in turn lower render distances.

### 5.3.3 Seams

There are noticeable seams between chunks. As seen in figure 9a and figure 9b the seams are less noticeable at higher resolutions they are still visible. This means they when the viewer is far from the planet they are more likely to notice the seams then when they are on the planet. However, as the higher resolution chunks are smaller in world space as the viewer moves they will move past more chunks and therefore cross more seams and even if they are less noticeable when static they could become more prevalent due to the constant motion of these hard edges.

They are due to the normals between neighbouring meshes being different due to each chunk not using data from the neighbouring chunks to generate the normals at boundary's. This is something that could be fixed in the future without too much effort by calculating the neighbouring vertices in the marching cubes compute shader.

### 5.3.4 Noise errors

The implementation of GPU simplex noise used in this project sometimes causes small indentations in the terrain. These artifacts are not visible when tested

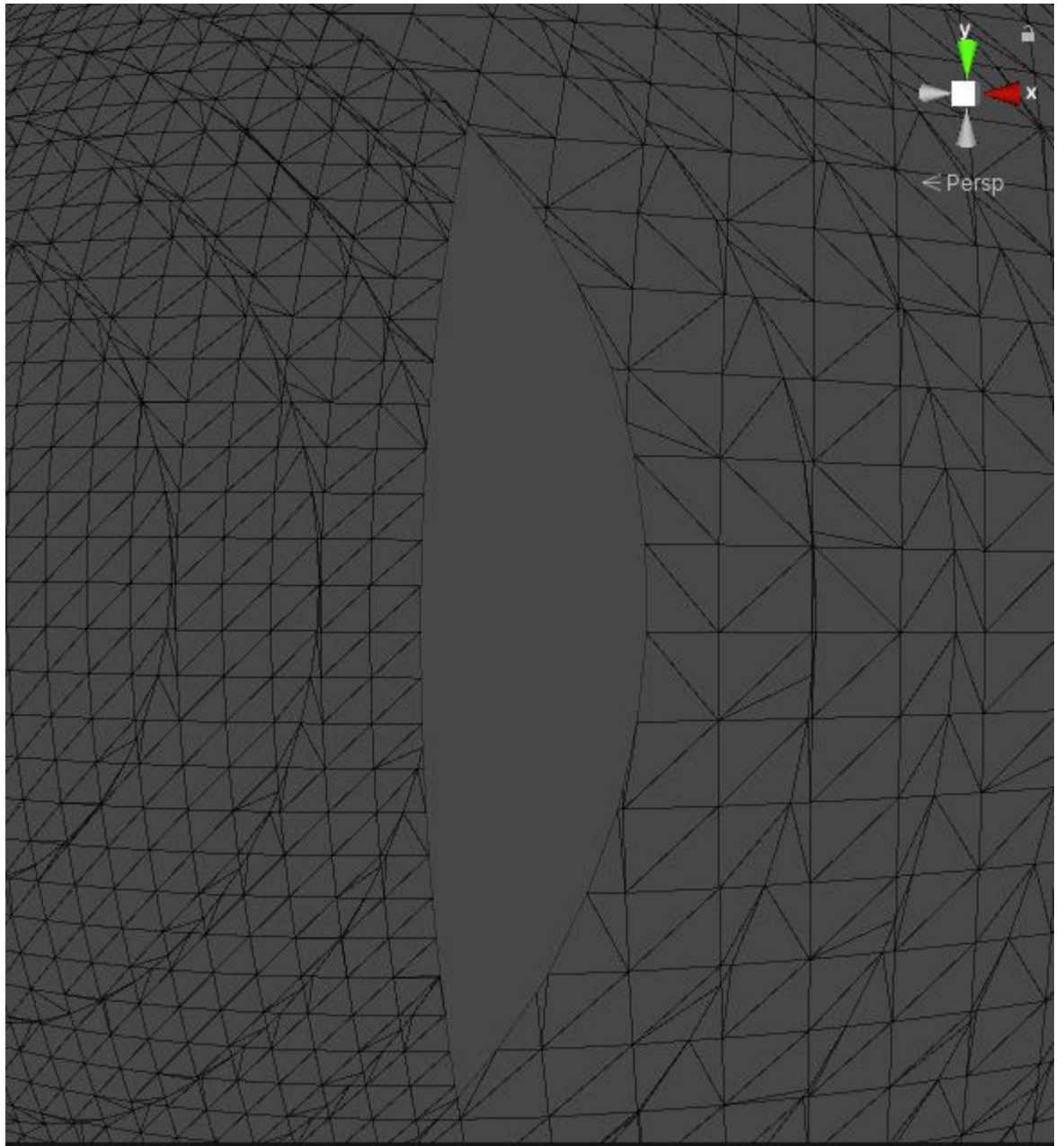


Figure 7: A wireframe image of a gap in the planet.

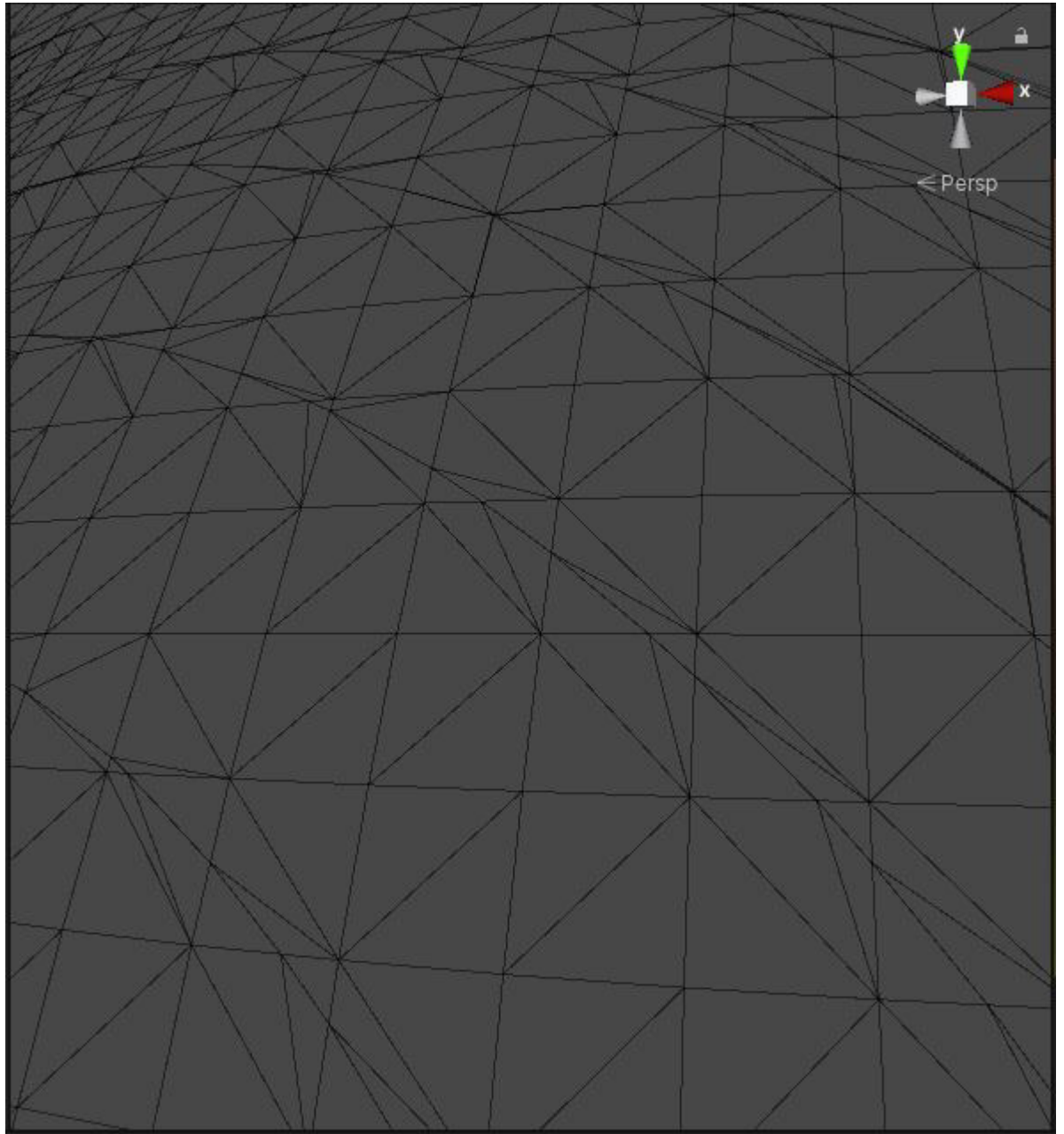
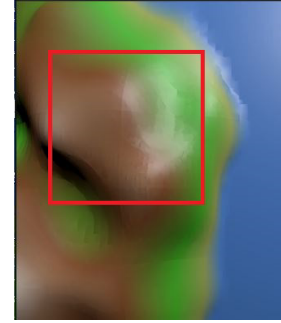


Figure 8: A wireframe rendering of part the planet detailing lots of small triangles



(a) Low resolution seams.



(b) High resolution seams.

Figure 9: Seams at different resolutions at the same location on the same planet.

with a CPU implementation of simplex noise which indicates it is due to a bug in the GPU version. The GPU implementation is simplified in some ways to increase performance on the GPU so this could be due to a performance trade off.

## 6 Future Work

From the results it can be seen that although at least 30 frames per second can be obtained using the level of detail system and the a low level of quality there is much than could be improved. In particular the framerate and the quality of the rendering need to be worked on.

To do this, the performance loss of transferring data between the CPU and GPU needs to be eliminated by keeping all data on the GPU. This would increase performance massively. In addition, chunks are currently generated at runtime even if they have previously been visited which is a waste of CPU and GPU time. Instead once a chunk is generated it should be compressed and saved to disk and then when revisited it can be read from disk again. This should give better performance than generating it, especially if the terrain generation gets more complicated (Kapoulkine 2017). This would also allow the terrain to be edited by the player at run time and then saved and reloaded.

Finally the quality of the terrain generation could be improved massively, currently the terrain is slightly dull and could feature many new types of terrain and also include features such as rivers and lakes. One bug that must be fixed is the gaps in the terrain as this immediately ruins the immersion and is an eye sore.

## 7 Conclusions

In conclusion the aim of the project was to create a system that could generate voxel planets in real time and still maintain an average framerate of 60 frames per second. In addition to this, the terrain also needed to contain interesting terrain features that can only arise from voxel terrain such as overhangs, caves, and cliffs.

The system designed utilised the marching cubes algorithm for isosurface extraction and used simplex noise to generate the terrain features. A level of detail system was created to help increase the performance of the system and increase the view distance.

The results of the system were that by using the level of detail system and a low level of detail an average framerate of at least 30 fps could be achieved. They also showed that the level of detail system gave better performance than just using a constant resolution. However, in order to get at least 30 fps the level of detail must be so low that it impacts the quality of the terrain which makes the planets less interesting to look at and blurry.

There were also some artifacts during the generation of the planets. These include seams at the borders of chunks all at the same resolution due to the normal vector calculations not taking into account the neighbouring meshes. In addition, when neighbouring meshes have different resolutions gaps are formed between the chunks and lots of skinny triangles are formed during generation which is a detriment to performance however, it can be solved by using dual contouring.

Finally, the terrain of the planet was not very detailed and did not contain many interesting terrain features, however the shading helped to bring the planet to life and make it more visually appealing to look at. The terrain generation could have done with a big improvement and would have drastically changed the look of the planets and made them more impressive.

## 8 Reflection on Learning

During the creation of this system I have learned a wide variety of skills and have found the application of isosurface extraction techniques to be very interesting. I would have liked to have spent more time on the generation of the terrain to create more interesting features however it was vital to have a solid rendering system implemented first. Without this, it would have been hard to view the generated meshes without long generation times. In reflection, when researching different techniques and trying to implement them I shouldn't have spent so long on particular features because sometimes I would finish implementing a feature only to realise its performance was too poor and that there was a better option. This means I should have done more research on different techniques or been willing to reconsider what else was out there if implementation was taking a long time.

I should have also spent more time testing the system in a variety of ways because when it came to time to collect results a variety of bugs appeared which made it impossible to collect any reliable data. If more testing was completed then I could have caught these bugs early on and wouldn't have had to rush to fix them.

Overall, I have found this topic fascinating to learn about and it has reinforced my passion for computer graphics and also games. It has shown that developing computer graphics applications are incredibly rewarding due to being able to visually see the results.

## References

- Bevins, Jason (2005). *What is coherent noise?* URL: <http://libnoise.sourceforge.net/coherentnoise/index.html/>. [accessed: 06/05/2020].
- Boris (2018). *Dual Contouring Tutorial*. URL: <https://www.boristhebrave.com/2018/04/15/dual-contouring-tutorial/>. [accessed: 11/05/2020].
- Bourke, Paul (1984). *Polygonising a scalar field*. URL: <http://paulbourke.net/geometry/polygonise/>. [accessed: 05/05/2020].
- DanielG.Aliaga (2010). *Level of Detail: A Brief Overview*. *Isosurface* (n.d.). URL: <https://svi.nl/Isosurface/>. [accessed: 07/05/2020].
- Kapoulkine, Arseny (2017). *Voxel Terrain: Storage?* URL: <https://blog.roblox.com/2017/04/voxel-terrain-storage/>. [accessed: 14/05/2020].
- Lague, Sebastian (2019). *Coding Adventure: Marching Cubes*. URL: <https://youtu.be/M3iI2l0ltbE/>. [accessed: 08/05/2020].
- Lengyl, Eric Stephen (2010). “Voxel-Based Terrain for Real-Time Virtual Simulations”. DOI: <http://transvoxel.org/Lengyel-VoxelTerrain.pdf>. [accessed: 13/05/2020].
- Michel, Christoph (2016). *Understanding front faces - winding order and normals*. URL: <https://cmichel.io/understanding-front-faces-winding-order-and-normals>. [accessed: 12/05/2020].
- Patel, Amit (n.d.). *Noise Functions and Map Generation*. URL: <https://www.redblobgames.com/articles/noise/introduction.html>. [accessed: 21/04/2020].
- Schliesser, Tim (2018). *How Many FPS do you need?* URL: <https://www.techspot.com/article/1668-how-many-fps-do-you-need/>. [accessed: 14/05/2020].
- Scott Schaefer, Joe Warren (1987). *Dual Contouring: The Secret Sauce*. DOI: <https://people.eecs.berkeley.edu/~jrs/meshpapers/SchaeferWarren2.pdf>.
- Scratchapixel (n.d.). *Value Noise and Procedural Patterns: Part 1*. URL: <https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/procedural-patterns-noise-part-1>. [accessed: 06/05/2020].
- William E. Lorensen, Harvey E. Cline (1987). “Marching Cubes: A High Resolution 3D Surface Construction Algorithm”. In: *Computer Graphics* 21.4, pp. 163–169. DOI: <https://doi.org/10.1145/37401.37422>.
- Zackary P. T. Sin, Peter H. F. Ng (n.d.). *Planetary Marching Cubes: A Marching Cubes Algorithm for Spherical Space*. DOI: <https://dl.acm.org/doi/pdf/10.1145/3301506.3301522?download=true>.