



Faculty of Science and Technology

BSc (Hons) Games Software Engineering

May 2022

Procedural Generation of Realistically Scaled Planets

by

Adam Hurst

DISSERTATION DECLARATION

This Dissertation/Project Report is submitted in partial fulfilment of the requirements for an honours degree at Bournemouth University. I declare that this Dissertation/Project Report is my own work and that it does not contravene any academic offence as specified in the University's regulations.

Retention

I agree that, should the University wish to retain it for reference purposes, a copy of my Dissertation/Project Report may be held by Bournemouth University normally for a period of 3 academic years. I understand that my Dissertation/Project Report may be destroyed once the retention period has expired. I am also aware that the University does not guarantee to retain this Dissertation/Project Report for any length of time (if at all) and that I have been advised to retain a copy for my future reference.

Confidentiality

I confirm that this Dissertation/Project Report does not contain information of a commercial or confidential nature or include personal information other than that which would normally be in the public domain unless the relevant permissions have been obtained. In particular, any information which identifies a particular individual's religious or political beliefs, information relating to their health, ethnicity, criminal history or personal life has been anonymised unless permission for its publication has been granted from the person to whom it relates.

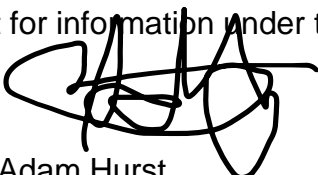
Copyright

The copyright for this dissertation remains with me.

Requests for Information

I agree that this Dissertation/Project Report may be made available as the result of a request for information under the Freedom of Information Act.

Signed:



Name: Adam Hurst

Date: 19/05/2022

Programme: BSc GD / BSc GSE (delete as appropriate)

Permission for use of produced work

We ask you to sign this form because Bournemouth University Higher Education Corporation (**BU**) may wish to use your Individual Project material (in its original or an amended form) for marketing purposes in one or more of the following ways:

- prospectuses and other BU promotional materials, including promotional videos
- as part of an advert or advertisement feature (which includes possible use on outdoor media such as buses and billboards)
- on the BU websites or intranet
- at open days
- in social media

If we used it, you would receive exposure for your work. You do not have to agree, and not doing so will have no effect on marking your Individual Project material. For ease this is simply called Material below.

If you are kind enough to agree to BU having rights of use and editing, this will not affect your ownership rights in the Material or your right to use it for your own professional or personal reasons. Unless it is impractical to do so – for example a montage on social media where space limits may apply, BU will ensure that it credits you for BU's use of the Material.

By signing this form:

- I grant a non-exclusive, world-wide, royalty-free licence to BU to use and sub-licence the Material, for marketing purposes; and I waive any non-property rights (including moral rights) now or in the future existing. This licence includes the right to edit and create derivative works from the Material, for example, a multi-student showreel.
- I agree this licence shall subsist permanently unless I seek to end it in line with the procedure below; but that where BU has already used, or is committed to using, the Material for a particular marketing campaign - for example, in a prospectus already printed, any ending of the licence shall relate only to new uses of the Material.
- **I understand BU will not ask me for any further approval or permission for specific uses of the Material for marketing purposes; but that I can withdraw my permission at any time, with the consequences for continued use set out above.**

If you wish to withdraw your permission for use of your information, please contact the Programme Support Officer by e-mail scitechcreativepso@bournemouth.ac.uk. Otherwise, we will keep your Material and associated personal information for as long as we have any plan to, or actively, use the Material for marketing purposes.

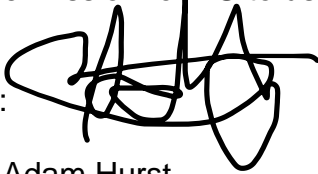
The legal basis for BU processing your personal information is consent, that is by signing this form you are giving permission to use your personal information provided in, and in relation to, the Material for the purposes set out above.

In processing your personal data, BU complies with the Data Protection Act 2018 and General Data Protection Regulation (Regulation (EU) 2016/679). You can find further information about BU's data protection and privacy approach here: <https://www.bournemouth.ac.uk/about/governance/access-information/data-protection-privacy>.

Further information about your rights under the data protection legislation can be found from our Data Protection Officer on dpo@bournemouth.ac.uk.

I give permission for BU to use my Material on the basis set out above:

Signed:

A handwritten signature in black ink, appearing to be 'Adam Hurst', written over a horizontal line.

Name: Adam Hurst

Date: 19/05/2020

Abstract

Procedural generation, the process of creating game content algorithmically, has been used in a plethora of modern games. Despite providing a valuable time and cost-saving alternative to manually designing each aspect of a given game world, it is still under-utilised. By designing a fully procedural approach to planetary generation, development teams can allocate further resources to develop core gameplay mechanics. This project seeks to implement this method by exploring various research sources, uncovering techniques, and then implementing them iteratively. The project consists of an executable to demonstrate the functionality of the Unity package, which has also been provided. This package has been designed with developers in mind, to aid them in implementing their own realistic and diverse planets within their projects.

Contents

Front Matter.....	0
Abstract	4
1.0 Introduction	6
1.1 Methodology	7
2.0 Literature Review	8
2.1 Mesh Generation	8
2.2 Level of Detail	10
2.3 Floating Point Errors	11
2.4 Procedural Terrain Generation.....	11
2.4.1 Noise	12
2.4.2 Data Structures.....	12
2.5 Atmospheric Rendering.....	13
3.0 Design and Implementation.....	14
3.1 System Design.....	14
3.2 Mesh Generation	15
3.3 Level of Detail	16
3.4 Floating Point Precision	18
3.5. Biome Generation	18
3.5.1 Temperature	20
3.5.2 Wind	21
3.5.3 Moisture	22
3.5.4 Biome Classification	22
3.5.5 Planetary Terrain Generation.....	23
3.6 Atmospheric Generation	24
3.6.1 Runtime Operations and Scriptable Objects	25
4.0 Evaluation	26
4.1 Performance Evaluation.....	26
4.1.1 Base Resolution.....	27
4.1.2 Radius	29
4.1.3 Biome Map Resolution.....	30
4.3 Planet Analysis	32
4.3.1 Terrain Analysis	32
4.3.2 Scale.....	35
5.0 Conclusion	36
5.1 Future Development.....	37
6.0 Works Cited.....	38

1.0 Introduction

Sculpting planets from scratch is an incredibly precise and time-consuming task. In recent years, developers have opted for a hybrid approach to creating planets. This utilised a procedurally generated mesh, which terrain can then be sculpted from. Whilst this approach has sped up the development process, there is still progress to be made.

This project has created a purely procedural approach for generating planets quickly and on a realistic scale. This scale was based on earth's parameters, creating a planet which is up to 6,378 kilometres in radius (NASA 2022). This program also features custom tools to be used by developers to create diverse planets at any scale. This tool is especially useful in saving time for smaller studios, especially those without pre-existing terrain systems.

The implementation and research behind this project followed the trend of procedurally generated content in games within the Sci-Fi genre. Games such as *No Man's Sky*, *Elite: Dangerous*, *Space Engineers* and *Astroneer* all feature aspects of procedural content generation. In addition, these examples each feature procedural planet generation. However, this is often relegated to a supplementary system in gameplay, for example, in *Space Engineers* and *Elite Dangerous* (SpaceGamerUK 2017). Other titles instead apply the concept to every aspect of the game, making it a part of the core gameplay loop, for example, in *No Man's Sky* (Lee 2015).

This variety of implementations highlights the strengths and weaknesses in each of the games previously listed. For example, *Elite: Dangerous* maintains realistically scaled planets but could be considered boring (Kumar et al. 2022). This is due to mainstream players preferring a diverse experience over an empty realistic one (Douglas 2016). On the other hand, *No Man's Sky* features highly detailed, varied planets which are kept to a small scale to suit the demographic of explorers and killers (Hohl 2022; Kumar et al. 2022). Consequently, this project aimed to unite the advantages of each approach. For example, to address the issues of monotonous design, a biome system was implemented to add diversity to the planet (Douglas 2016).

The project aimed to create a Unity package and executable that demonstrated a complete procedural planet generation system, and was realistic in features and scale. The executable allows the user to set parameters for the generation of the planet (such as size, colours, seed etc.) with a random planet setup for the user to explore and observe using a flying camera. This was created using the Unity3D game engine with the universal render pipeline.

To fulfil the aim of the project, several smaller objectives were created. These objectives comprised of:

- Generating a sphere with evenly distributed vertices
- Creating a level of detail (abbreviated as LOD) system to simplify the mesh the further the player is from the planet
- Designing a system to remove any issues that could occur due to floating-point precision errors
- Creating a terrain generation algorithm that is customisable with parameters, that also generates diverse and interesting terrain
- Implementing realistic atmospheres for generated planets.

Having specified the scope and rationale behind the project, the research stage began. The following literature review details the research conducted into each of the core developmental components. This includes a critical evaluation of readings across the areas of mesh generation (Patel 2022; Cajaraville 2019; Schneider 2006), LOD systems (Savage 2017; Hoppe 2004), floating point precision errors (O'Neil 2022; Unity 2013; Montgomery 2008), procedural terrain generation (Fischer 2020, Michelic 2019, Lagae 2010) and atmospheric generation (Elek 2009, Schafhitzel 2007, O'Neil 2005).

Building on the literature review, the design and implementation section will discuss how each of the core components has been developed. This involved an iterative approach, with the initial design and implementation being evaluated and reworked to address any bugs encountered. Diagrams, graphs, and images have been provided to showcase the systems' development.

Closing the dissertation is a conclusion that analyses and evaluates the final Unity package and game executable. This section also outlines areas for refinement within the implementation and details the knowledge and skills developed throughout the project.

1.1 Methodology

Background reading has been conducted into each of the core components of the design. Additionally, existing implementations within established games have been evaluated to help define the feature set included in the final package.

The implementation has been created within the Unity3D game engine, due to previous experience using Unity. This engine also features an existing compute shader system, as well as being accessible to the end user. As the project was created within Unity, all scripts were written in C#. This allowed for a structured, object-oriented approach. HLSL was also used to directly interface with the GPU for large increases in performance in parallel workflows.

The creation of each component followed a specific methodology. This involved taking various techniques from the literature review and testing them, before running test implementations and evaluating the outcome. This iterative process was repeated until the optimal method was found.

To analyse the product, a testing setup was created within Unity. This was done through Unity's included profiling suite for testing memory usage, and to allow for easy adjustment of the configuration sliders. Once set up, the generated planets were visually compared to others within the industry (for example, terrain diversity and realism), and quantitative factors such as memory usage and frame rate were also measured and analysed.

2.0 Literature Review

The following section details the research conducted into each of the core implemented modules. The central component of the project was mesh generation, as each of the other components would alter the base mesh. For example, terrain generation manipulated the vertices of the mesh, giving the planet realistic terrain. Whilst scaling the planet, floating point errors may emerge, which will cause visual artefacts which will need to be fixed.

2.1 Mesh Generation

Spherical mesh generation was crucial to creating an accurate representation of a planet. This is because most planets are typically a spherical shape, due to gravitational forces pulling material to the centre of the planet (Sears 2022). During this research, several effective techniques were found for generating a sphere mesh.

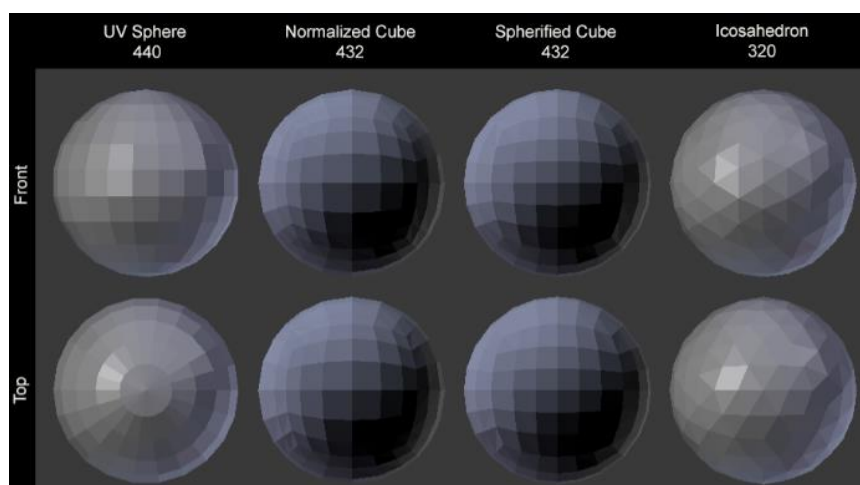


Figure 0: Graph demonstrating examples of the different mesh generation methods (Cajaraville 2019)

These techniques included: UV spheres, normalised cubes, spherified cubes and icosahedron, which can be seen in Figure 0 (Cajaraville 2019).

These algorithms can have their effectiveness evaluated based on their:

- computational efficiency
- distribution of vertices
- accuracy of the generated vertices to the unit sphere,
- control over the mesh's resolution.

The effectiveness has been evaluated, and provided in the table below (Figure 1).

Algorithm	Computational Efficiency (1-5)	Distribution (1-5)	Accuracy (1-5)	Control (1-5)
UV Sphere	5	1	1	2
Icosahedron	4	2	2	1
Normalised Cube	3	3	4	4
Spherified Cube	2	4	4	4
Fibonacci Sphere	1	5	5	5

Figure 1: Table demonstrating relative ratings of the different mesh generation methods (Cajaraville 2019; Patel 2022)

A benefit of both the cube algorithms is the ease when implementing a Quadtree or geometric clipmap, which can be used as a LOD system for changing the mesh's complexity (Schneider 2006).

One additional method is the Fibonacci sphere (Patel 2022). This algorithm allows for evenly distributed vertices when compared with previously described methods. As remarked by Keinert et al, the Fibonacci sphere is a "well-known approach to generate a very uniform sampling of the sphere" (2015, 7). Unfortunately, due to the non-linear generation of the vertices, triangulating these points proved computationally difficult (Lague 2020). Another downside of this approach would be the difficulty of implementing a LOD system, caused directly by the generation method of the vertices.

One promising technique is called the marching cubes algorithm. This is a method of converting voxel based values to a polygonal mesh. Voxels are defined as "a value on a regular grid in three-dimensional space" (Mega Voxels 2019). The algorithm works by using a set of 8 voxels to form a cube, then generating a triangle based on these 8 values (Sin and Ng 2018).

This technique is typically used on flat terrain, however a paper written by Sin and Ng demonstrates a method to transform the voxels into the unit sphere, allowing for the creation of spherical objects (2018). Unfortunately, the algorithm is known to be significantly slower than the other techniques described, due to the original algorithm having to traverse all the voxel data to generate the mesh (Newman and Yi 2006). Although efforts have been made to speed up and improve this algorithm, a more traditional approach would work best for something of the scale intended for this project.

2.2 Level of Detail

If a LOD system is not implemented, rendering highly detailed planets would require the generation and rendering of billions of vertices every frame. This limitation is important, as the max mesh size in Unity (using a 32-bit index buffer) is 4 billion vertices (armDeveloper 2022). Additionally, the increase in mesh complexity would cause performance issues with a mesh of this size. (Ruben 2020)

One technique for implementing a LOD system is a data structure called a quadtree. Raphael Finkel, the creator of the quadtree, defines them as, “a data structure appropriate for storing information to be retrieved on composite keys” (Finkel et al. 1974). As such, this approach would ideally be complemented with the use of heightmaps of varying detail, as the quadtree is designed to pull from an existing data set. Unfortunately, due to the intended scale of the project, it would be infeasible to generate the required terrain data before runtime. Additionally, this reliance on a pre-existing data set would mean that the implementation could not have a purely procedural approach, which is the desired outcome of this project. Having understood these limitations, using a quadtree was deemed unsuitable for this implementation.

Geometric Clipmaps are also used to implement LOD systems. This system “caches the terrain in a set of nested regular grids centred about the viewer”, and functions similarly to the algorithm implemented within texture clipmapping (Hoppe 2004). According to Savage, this algorithm works by tessellating the terrain mesh at a higher resolution in the centre of the mesh and decreasing the resolution further from the camera (2017). Savage’s blog also discusses further methods of expanding this technique. These techniques include using Geomorphing to transition between LODs smoothly and adding terrain skirts to more traditional plane-based terrain approaches (Savage 2017). Due to the algorithm's simplicity, this method was attempted during the implementation.

2.3 Floating Point Errors

Floating point errors are defined as precision issues that occur when trying to represent an infinite series of numbers within a finite set of bits (Montgomery 2008). As floating-point numbers (also known as floats) only have 6 digits of precision, making a 1:1 scale planet can cause complications (O'Neil 2022). Symptoms of this inaccuracy can be seen in a talk at 'Unite 2013' about the game *Kerbal Space Program* (abbreviated as KSP). This talk demonstrates a "Jitter" that occurs, where the gameobject vibrates, which worsens the further out they bring the test spaceship (Unity 2013).

This conference describes a solution to this issue, which moves the camera and game objects into different spaces depending on the current planetary scale (Unity 2013). This approach, although effective, would not be ideal for this project. This is because the discrete spaces described by the KSP developers do not allow for a completely seamless transition between these spaces.

Another method for dealing with these errors was using doubles in place of floats. This worked because a double features twice the precision of a floating-point value (Shankar 2021). One trade-off of this approach is the increased memory bandwidth required.

One final approach that aids in alleviating floating point errors is the scaling of planets depending on their distance to the camera (O'Neil 2022). This method would allow the planet to remain within the area of precision for floats, whilst still giving the illusion of being much further away.

2.4 Procedural Terrain Generation

The creation of realistic and diverse terrain was crucial to the project's success. Procedural generation has been defined by Shaker et al. as "the algorithmic creation of game content with limited or indirect user input" (2016). When applied to terrain, this refers to the techniques used to generate locations for the vertices of a mesh (Rose and Bakaoukas 2016).

Procedural generation of terrain is important, especially in this project. It allows for the generation of infinite maps without taking up disk space, by being generated at runtime. Furthermore, the short generation time allows development teams more time to focus on gameplay mechanics. When using procedural terrain generation, key considerations include noise, heightmaps and data structures (such as voxels and arrays).

2.4.1 Noise

One of the more popular techniques within the field of procedural generation is the use of noise functions (Scratchapixel 2022). Noise is defined as, “the random number generator of computer graphics” (Lagae et al. 2010). Of these noise functions (such as Perlin, simplex and anisotropic), each has its own characteristics such as coherency and distribution. These noise functions are then used to generate a float, which is mapped to an elevation value that is used when generating the terrain. However, this method does have its drawbacks. Fischer et al. remind that noise is a difficult tool to implement well, leading to difficulties when creating “genuinely realistic looking terrain” (2020). Furthermore, without using multiple layers of noise (fractal noise), the terrain generated would be very repetitive. This can be seen in the product of Michelic’s work (2019).

However, combining fractal noise with basic weather simulation, as used in autobioomes, can create both realistic and diverse terrains (Fischer et al. 2020). This approach uses an initial layer of fractal noise, before applying weather simulation to this data. This weather simulation includes the calculation of temperature, wind and precipitation. This then breaks the mesh up into different biomes. After this step is complete, the algorithm then applies more fractal noise to the terrain, specific to the generated biomes. The high-quality output of this method made it suitable for this implementation.

2.4.2 Data Structures

When noise is used in conjunction with flat terrain, the data is typically stored within a two-dimensional heightfield. According to Becher et al, this is, “the most common data structure used for the storing and rendering of terrain” (2017). The data stored within these heightmaps are altitudes, which are used by a mesh builder/terrain generator to construct or manipulate the mesh to represent different heights. Although this method is prevalent across many games and papers, heightmaps cannot represent multiple level terrain, such as cliffs and overhangs. This is due to the data being stored in a two-dimensional format (Becher et al. 2017).

Another method that would allow for the implementation of more advanced terrain features, is voxels. As previously discussed, voxels can be used to store volumetric 3D data. However, both these methods required a data set, which was infeasible due to the scale of the project. Instead, a more algorithmic approach is better suited for this implementation, generating the appropriate terrain at runtime to cooperate with LOD systems. This would involve calculating the vertex’s exact position on the planet when it needed to be rendered.

2.5 Atmospheric Rendering

Atmospheres help to create realistic and immersive planetary environments. Schafnitzel et al. (2007) and Elek (2009) feature similar techniques that solve the problems of efficient atmospheric rendering. The core functionality of this algorithm is derived from pre-calculating the light scattering integral and storing all of this data in a lookup texture or table. This will then be used by a GPU shader as a post-processing effect or as part of the fragment shader (Elek 2009). The scattering integral can be computed using two different techniques, Rayleigh and Mei (O'Neil 2005). Rayleigh scattering is the scattering of smaller particles within the atmosphere, whereas Mie is relevant to the much larger airborne particles within the atmosphere. Whilst a combination of these techniques would be ideal, either one of these methods individually would work for the implementation.

3.0 Design and Implementation

This section will explore the design and iterative process of implementing: sphere mesh generation, LOD systems, biome generation and atmospheric rendering. The strengths and weaknesses of each approach will be discussed, accompanied by a rationale for their inclusion in the final implementation.

3.1 System Design

The philosophy for the implementation of this project was to create a system that is customisable and easy for developers to implement. As such, the program would be broken down into as few scripts as possible to make it easier for the end user to set up. The main script generating the planet, `Hurst_PlanetGenerator`, encapsulates the majority of the code for the project. No other scripts will be called within this class, with the exception of compute shaders. This allowed for increased performance and has not been designed to be edited by the end user. The camera script, called `CameraScript`, is designed to be accessible but also cannot be edited by the user. There was a focus on using Unity scriptable objects to hold the majority of the configuration parameters. These objects stored data for the biomes and planets' atmosphere.

3.2 Mesh Generation

Comparing the sudo-code for normalised and spherified cubes highlighted the efficiency of the normalised cube algorithm. This is due to the spherified cube featuring square root mathematics, which is computationally slow (Cabot 2020).

Research was conducted into whether this algorithm had been previously implemented into a planetary generation system. A video series created by Sebastian Lague (2020) featured the inclusion of a normalised cube. Lague's code became the base of the sphere generation algorithm for this project. Sudo-Code for this algorithm can be found below (figure 2).

```

For f faces in cube:
    AxisA = GetUpOfNormal(Normal)
    AxisB = Cross(Normal, AxisA)
    For (y < MeshResolution)
        For (x < MeshResolution)
            Iterator = x + y * MeshResolution
            Percent = (x, y) / (MeshResolution - 1)
            PointOnUnitCube = Normal + (Percent.x - 0.5) * 2 * AxisA +
                                (Percent.y - 0.5) * 2 * AxisB
            PointOnUnitSphere = Normalize(PointOnUnitCube) * Radius
            Vertices[Iterator] = PointOnUnitSphere
            If x != MeshResolution - 1 And y != MeshResolution - 1
                Triangles[NumberOfTriangles] = Iterator
                Triangles[NumberOfTriangles + 1] = Iterator + MeshResolution + 1
                Triangles[NumberOfTriangles + 2] = Iterator + MeshResolution
                Triangles[NumberOfTriangles + 3] = Iterator
                Triangles[NumberOfTriangles + 4] = Iterator + 1
                Triangles[NumberOfTriangles + 5] = Iterator + MeshResolution + 1
                NumberOfTriangles += 6

```

Figure 2: Sudo-Code outlining how to generate the vertices of a normalised cube

This algorithm generated the points of a unit cube using a percentage value calculated from the resolution of the mesh, and the relative up and right axes of the faces. These vertices are then normalised to create a sphere and multiplied by the radius to make the sphere the user specified size.

The initial results of the sphere algorithm functioned well, creating an accurate base sphere for the planet. However, using this method at higher mesh resolutions caused

the program to slow to an unplayable speed. To resolve this issue, the mesh would have to be simplified, for example, by not generating non-visible faces.

Originally, this adjustment used the forward vector of the camera to render faces that had surface normals facing in the opposite direction. This method was promising; however, it caused some faces of the normalised cube not to render. This was due to the use of an arbitrary threshold value within the program which only provided a rough approximation of whether the face was visible. Additionally, using the forward vector of the camera proved to be an ineffective way of representing the visible portion of the planet.

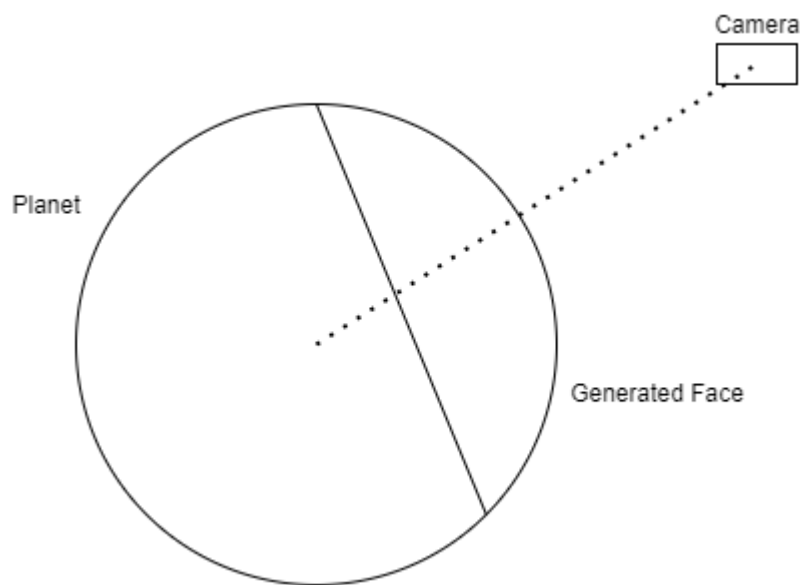


Figure 3: Diagram showing how the face normal is calculated based off the camera's location

The final implementation dynamically changed the normals fed into the algorithm. This approach does not use the cardinal directions as inputs (up, down, left, right, forward, back). Instead, a normal vector is calculated based on the camera's position relative to the surface of the planet (Figure 3). This allows for a face to be generated angled towards the camera, regardless of the player's location in world space. Once this initial face was generated, four other connecting faces were generated at a lower resolution, to maintain the illusion of a complete sphere.

3.3 Level of Detail

For order for high detail geometry to be properly rendered, a LOD system was required to allow for the mesh of the planet to modify its resolution. However, during the system's implementation, several issues occurred. This led to its omission from the final program.

The first attempt at implementation featured geometric clipmapping. Although this method preferred to pull from a pre-existing data set, it was possible to alter the algorithm to work with the project's procedural approach. Mike Savage's full implementation (2017), based on research by Hoppe (2004), provided a reference for how to implement a geometric clipmap. However, attempts to integrate this approach within Unity proved to be too difficult and time-consuming.

Despite this setback, a simplified version of this approach was attempted for the final implementation. At the lowest LOD, this system would generate the mesh normally to a user specified resolution. As the camera approached the planet, the LOD settings would then increase, resulting in a section of the face being rendered at double the resolution of the initial basic mesh. The pattern of the mesh was calculated by generating an outline of the planet's face, before rendering the mesh at a higher resolution inside the border. This process would then repeat for all following LODs generating more borders the higher the LOD setting was. The only difference occurred past LOD 2, where the outer outline of geometry is removed to limit the total vertex count (Figure 4).

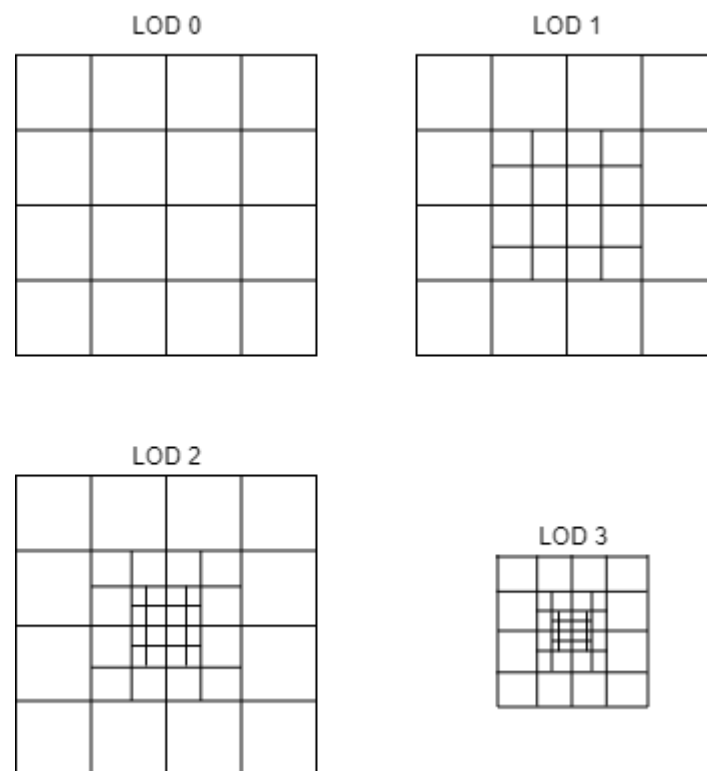


Figure 4: Diagram showing how the LOD system splits up the mesh when the LOD value increases

Whilst integrating this approach with the mesh generation compute shader, some issues arose. These issues included incorrectly triangulating the vertices and having

problems calculating the correct offsets for the borders. Unfortunately, the LOD system had to be omitted from the final implementation due to these issues.

3.4 Floating Point Precision

When working on a large scale, the geometry of the planet would begin to disappear, and the camera would shake. These symptoms highlighted the issues caused by floating point precision errors.

To remedy these errors, several additions were implemented, including a new data structure (DVec3) and a new script (Hurst_PlanetaryCameraController). DVec3 was designed to hold the position of gameobjects within a planetary scale. By utilising doubles, which double the precision of floating point numbers, objects were able to travel a greater distance. DVec3 also featured several helper functions, such as a distance calculation, to help integrate it with the existing codebase.

The CameraScript functioned to keep the camera within a bounding box of 10,000 by 10,000. This ensured the planet moved around the camera, relative to the camera's position within the bounding box. Through testing, this proved to alleviate all visual artefacts that were previously observed.

3.5. Biome Generation

For the generation of diverse and interesting planets, a combination of terrain and biome generation was necessary. Biome generation emerged as a core component within the project, acting as a way to break the planet up into diverse sections.

This then informed the terrain generation system, which was crucial in adding altitude to the generated vertices of the mesh. This allowed for the terrain of the biomes to exist within the game world.

The iteration of the biomes component was inspired by the work of Fischer et al. (2020). The stages of implementation included:

- Generating the temperature data,
- Generating the wind data,
- Generating the moisture data,
- Specifying the biomes.

As running these elements were predicted to be computationally expensive, it was decided that the biomes would be precomputed on start and stored within an array. This limited the computations per frame and increased the framerate of the final product.

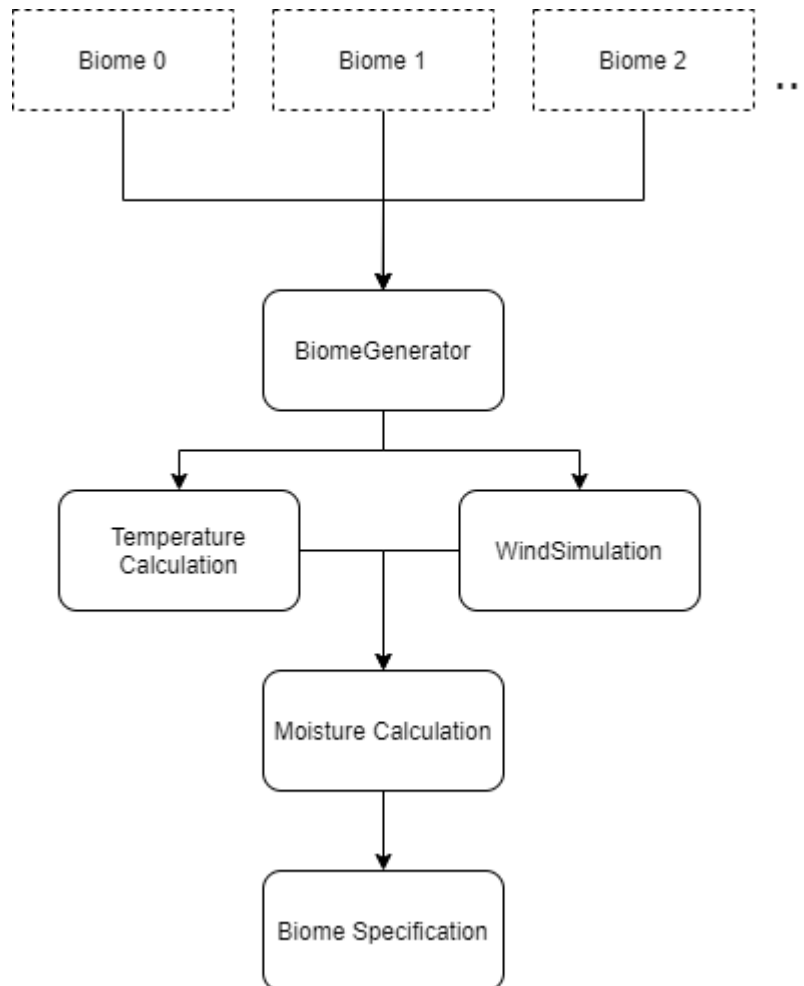


Figure 5: Diagram showing the workflow of the biome generation system

3.5.1 Temperature

Fischer et al. (2020) offered a logical calculation for generating the temperature values. In their paper, temperatures are calculated per vertex of the mesh, dependent on their altitude and their distance from the poles of the planet (Fischer et al. 2020).

Within this project, temperature calculations were implemented by first generating vertices of the planet at a user specified biome resolution. Each of these vertices was used to calculate an initial temperature value relative to the predefined poles ($\{0,1,0\}$, $\{0,-1,0\}$). Another temperature value was then calculated by mapping the vertex's magnitude between the radius of the planet and its maximum possible height, generating a 0 to 1 value. The two values were then multiplied by a user defined weight and combined to create a temperature value between 0 and 1 (Figure 6).

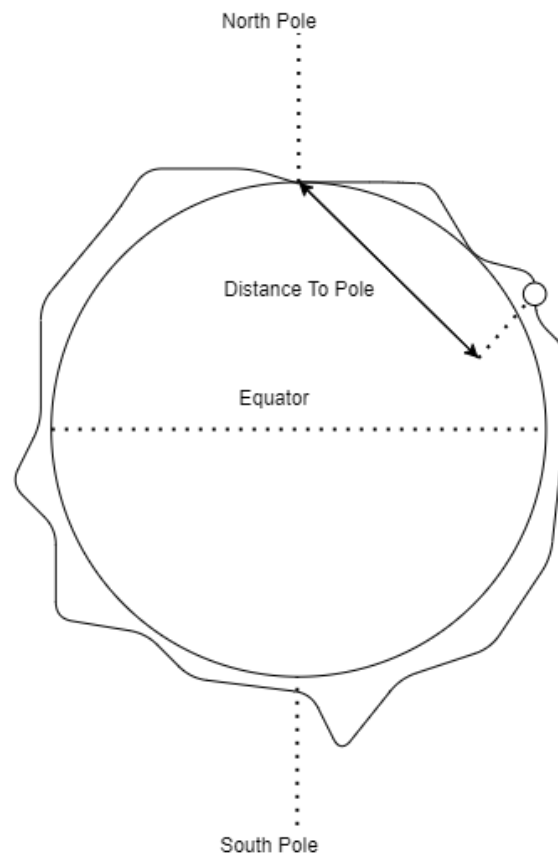


Figure 6: Diagram showing the how the temperature it calculated, using the distance to the pole and its relative altitude

3.5.2 Wind

Implementing wind was the most complex part of the biome generation process. Fischer et al. (2020) suggested using an approach as described by Jos Stam (2003), whilst removing some of the intricacy, such as diffusion, for performance reasons. However, this method was still complex to implement as it involved advanced fluid dynamic concepts. Instead, a simpler approach was designed and utilised.

This technique worked by creating an array to store all of the wind values. This was combined with a mapping function, allowing for a cube representing the planet to be held within this array. Several wind origins were then randomly generated, dictating where the wind algorithm would start from. The wind generator then iterated through the available wind nodes, propagating them across the cube in the direction which the wind nodes were facing (Figure 7). The addition of a small random deviation allowed for the simulation to simulate real world air disturbances. This would repeat a user-defined number of times until the wind had spread across the majority of the cube.

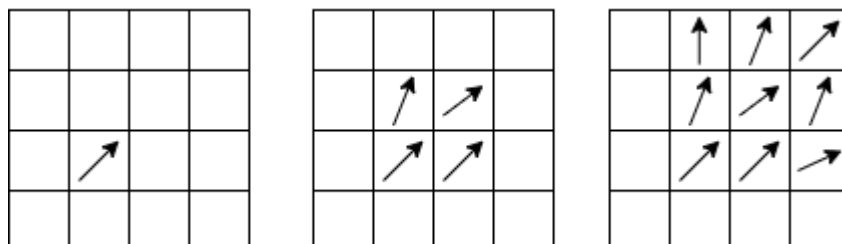


Figure 7: Diagram showing the how the wind vectors are propagated across the wind map, over each iteration

Unfortunately, issues with reliability and the vast range of the wind values meant another solution was required. The final implementation was a simple noise function. This used the array from the previous attempt but utilised the Simplex noise function to act as the wind. This created a random value that was coherent whilst also being a close approximation of the data previously gathered. Although not realistic, this method allows for gradual changes in biomes, similar to how the wind simulation would have.

3.5.3 Moisture

Initially, generating the moisture values of the biome map utilised a similar approach to that of the wind. Fischer et al. described propagating and diffusing the moisture based on the wind generated (2020). However, due to the problems encountered whilst programming the wind algorithm, this was not possible.

Instead, the final implementation calculated the moisture value from an average of the temperature and wind values. As a result, if the wind speed was high and the temperature was low, the moisture level was high and vice versa.

3.5.4 Biome Classification

Once all of the components of the biome had been calculated, the final biome map could be generated. Fischer et al. specified that moisture and temperature values should be used within this biome definition process (2020). As such, this approach was utilised within this implementation.

To allow for the creation of user defined biomes, a scriptable object was created. This object featured the parameters for the biome's classification (such as moisture and temperature range) and customisation parameters (such as the biome's relative altitude and the scale of its noise).

These objects would then be passed into the biome classification function. This function used the temperature and moisture ranges specified by this scriptable object and compared them to the previously calculated moisture and temperature values. This calculated biome was then passed onto the biome map, to be used with the terrain generation component.

3.5.5 Planetary Terrain Generation

Once the biome map had been generated, the planet's terrain could be implemented. The terrain was added using the previously discussed mesh generation shader with some modifications.

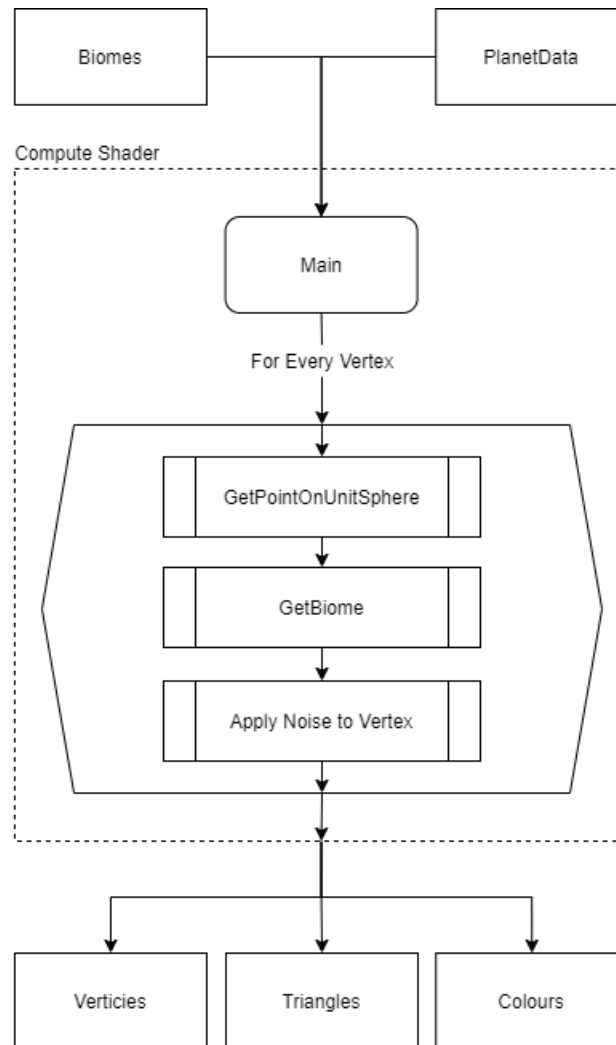


Figure 8: Diagram showing the workflow for the mesh generation shader, with the added terrain components

Firstly, biome parameters were added. This allowed the compute shader to have awareness of each of the user defined biomes. With this new data, two new helper functions were required, alongside the compute shader. As the biome data was stored in a single dimensional array, a way of mapping the biome data to a 3D location was required. This was done by normalising the vertex, and calculating which face of the unit cube it was in. Once calculated, the vertex coordinates would then be used to calculate a position within the array of biomes.

After the biome had been retrieved, the altitude of the vertex was set. This used the openSimplex noise function to generate a value from 0 to 1. The resulting value was then multiplied by the maximum terrain height and the biome's relative altitude value. This new altitude value is then applied to the vertex of the pre-calculated unit sphere to generate the final vertex. This process would then be repeated for all the vertices, generating the final mesh to be rendered.

3.6 Atmospheric Generation

Atmospheric generation was the final element to be implemented. This section contains details about the different shaders and scripts that created this component of the project.

Elek (2007) and Schafhitzel et al. (2009) both utilised identical approaches to implementing atmospheric rendering. Whilst this method appeared popular, the complex mathematical concepts required seeking out an existing implementation. The example, provided by Lague (2020), featured a full implementation which expanded my understanding of the subject. Lague's work became the core of this component, due to his approaches' efficiency and visual quality.

The fragment shader and the scattering pre-computation compute shader were taken directly from Lague's project, which used Rayleigh scattering (2020). This was due to their known functionality and observed efficiency during runtime. However, some issues arose when porting his code to this project. This is due to Unity version incompatibilities, and this project's use of Unity's Universal Render Pipeline (abbreviated as URP).

Many attempts were made to fix these issues. These included translating the fragment shader to a compute shader and moving the majority of the computation to an lenumator function on the CPU. The final implementation required rewriting the two shaders, testing each line of the shaders and tweaking them to work better with the URP.

3.6.1 Runtime Operations and Scriptable Objects

One feature of Lague's implementation that required updating was his scriptable objects. These objects held the data which set the parameters within the fragment shader.

Lague's implementation lacked support for the atmosphere effect moving with the mesh of the planet. To overcome this, additional functionality to the `SetProperties` method was made. An additional `Vector3` parameter and call to the fragment shader allowed for the planet position to be passed into the fragment shader. This resulted in the successful movement of the planet's atmosphere where necessary.

To properly integrate the atmosphere code with the `Hurst_PlanetGenerator` script, further functions were needed to update the `AtmosphereSettings` scriptable object. To do this, the `CameraScript` would call the `UpdateAtmosphere` function in `Hurst_PlanetGenerator`. This would then trigger the update within `AtmosphereSettings`, updating the fragment shader.

4.0 Evaluation

In the following section, the final Unity package will be evaluated in several ways. This includes:

- A performance evaluation (Framerate, memory usage etc.)
- An evaluation of the mesh's quality (vertex distribution, mesh seams)
- An overall analysis of the generated planet (Visual details, diversity etc.)

All of the tests were conducted on the hardware and settings detailed below (Figure 9).

Hardware Specification	
Part	Spec
CPU - Ryzen 2700x	8 cores, 16 threads, 4.2 ghz
RAM	Ddr4 3000mhz, 64gb
GPU - Gtx 1080ti	11 gb gddr5
Testing Settings	
Resolution	1273 X 952
Temperature weight	1.0

Figure 9: Table outlining the specifications of the hardware used for testing, as well as the setting set in Unity

4.1 Performance Evaluation

The project's performance is an important metric to measure. If the program performed poorly, it would not be a helpful tool for the end user. This is because, if it performed inadequately, so would the end user's program. To evaluate the program's performance, three main aspects were investigated. These areas were:

- Base resolution
- Radius
- BiomeMap resolution

These areas were specifically selected due to a prediction that they would be the factors which most impacted the program's performance. This was due to either the large size of the data being processed, or the effect the data had on the mesh generation shader.

All tests were conducted using the built in profiler within the Unity editor. This allowed for the generation of quantitative data, such as frame rate, memory usage and computation time.

4.1.1 Base Resolution

The base resolution directly impacted the number of vertices generated within the mesh. As a result, this factor would most negatively impact performance. In order to evaluate the base resolution, two tests were conducted.

The first test conducted observed how the base resolution affected the frame rate. This metric was crucial for the end user, as a poor framerate would lead to a poor gameplay experience. Additionally, without the inclusion of a working LOD system, it was crucial to analyse the detail of the mesh whilst remaining in a playable state. As such, the frame rate between resolutions from 30 to 150 squared vertices were tested. This tested whether the program met a playable frame rate of 24 FPS (Logical Increments 2021).

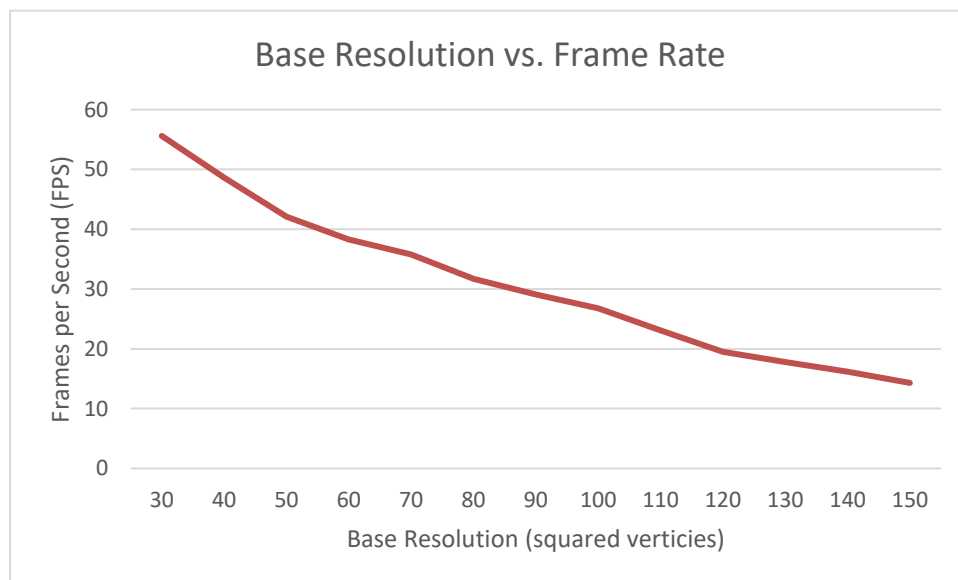


Figure 10: Graph outlining the relationship between the frame rate of the gameplay, and the resolution of the mesh

As shown in the graph above (Figure 10), the falloff of the frame rate to resolution followed a pseudo linear trend. Unfortunately, when the mesh exceeded a resolution of 100 squared vertices, the game's framerate dropped below the playable threshold. This further highlighted the necessity of a LOD system, which is later discussed in

the output analysis, as the project would have benefitted from being able to generate a higher resolution to allow for a higher detailed planet.

To clarify these findings, the applications memory usage was measured during runtime to identify a possible correlating linear pattern. This was also crucial to identifying the efficiency of the program, and whether it would be able to run on lower end hardware. As can be seen in the graph below (figure 11), the memory usage did follow a linear increase.

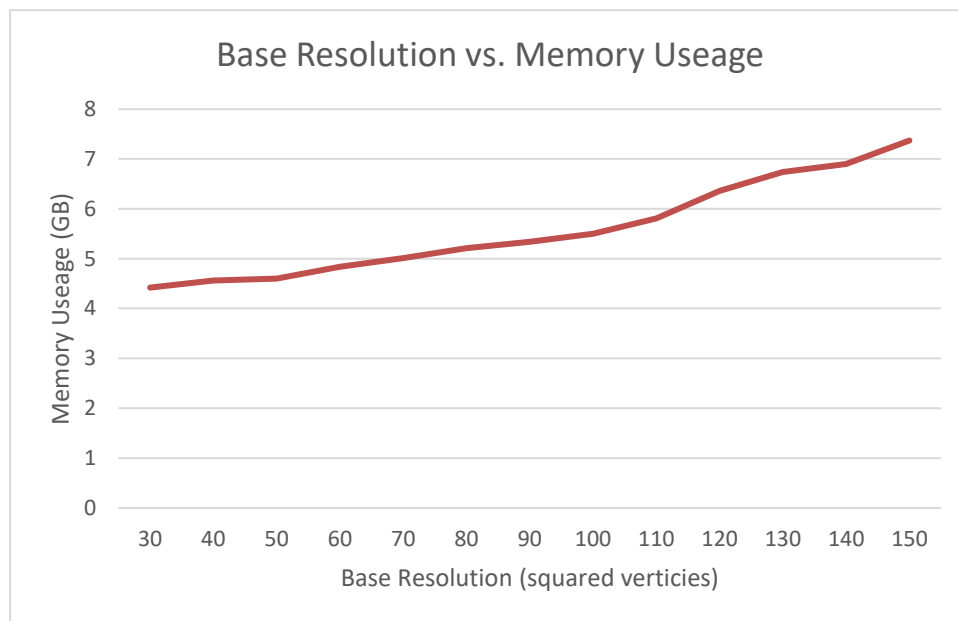


Figure 11: Graph outlining the relationship between the memory usage of the application, and the resolution of the mesh

Once again, this reiterates the importance of a functioning LOD system. The average PC contains 8 GB of RAM, so increasing the resolution of the entire mesh beyond the testing parameters would result in a non-functional program for the majority of computers (Witman 2022).

4.1.2 Radius

It was crucial to investigate the radius value, as Unity needed to function regardless of the mesh's potentially large scale. The test that was conducted measured how the frame rate was affected as the radius value was changed. As observed below, there was no observable difference when the radius value increased. The change within the graph is simply a difference of a few frames per second, and as such, is considered within the margin of error for this test.

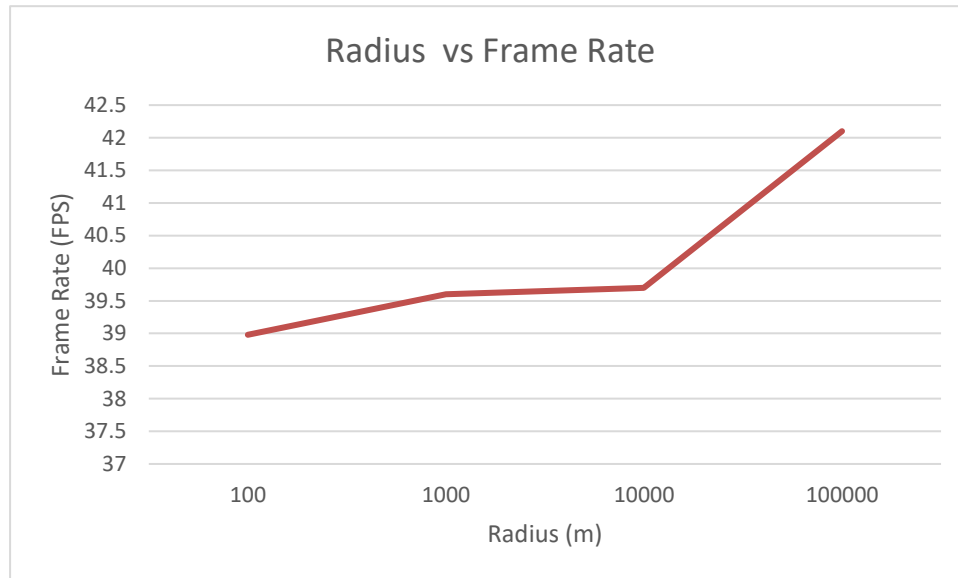


Figure 12: Graph outlining the relationship between the frame rate of the game, and the radius of the planet

4.1.3 Biome Map Resolution

The biome map generation is the only component of the project which doesn't execute every frame. This was a decision taken during the implementation stage, as it was predicted that calculating the data every frame would lead to poor performance. As such, a test was devised to evaluate the outcome of this decision. This would also highlight the relationship between the biome map resolution and the time taken for the biomes to generate. Biome map resolution values of between 10 and 150 pixels were tested multiple times. This allowed for an average to be calculated, providing a more accurate representation of this relationship.

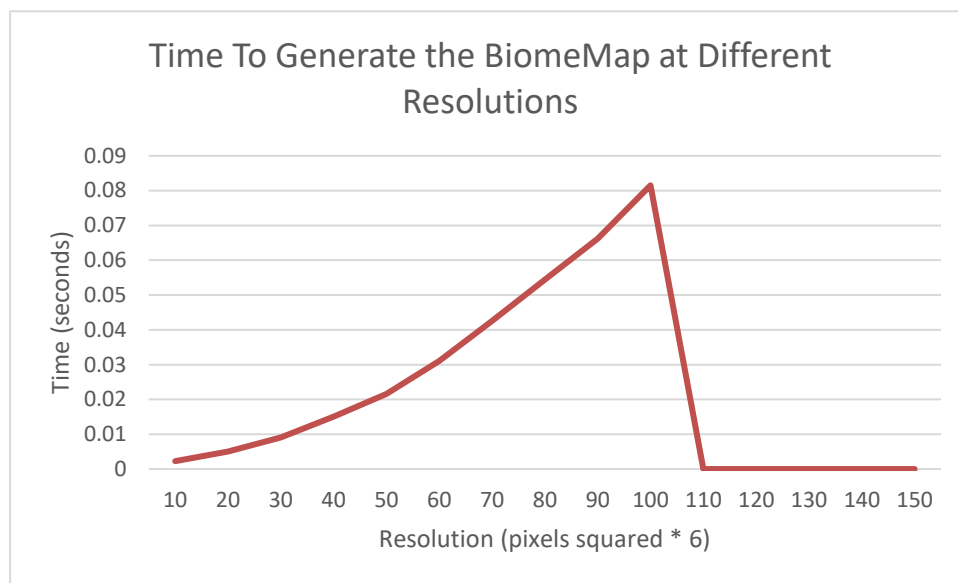


Figure 12: Graph outlining the relationship between the resolution of the biome map, and the time it takes to generate

As shown in the graph above, an exponential relationship can be seen between the resolution of the biome map and the time taken for the biome to generate. This has likely been caused by the many for loops which make up the algorithm, as well as the larger data set requiring more time to iterate through. Unfortunately, when the resolution reached a value of over 100, the system crashed and exited with a memory violation error. This can be observed in the above graph as the plateau. This was likely caused by a logic error in the getBiome function. As resolutions higher than 100 pixels had not been tested, a fix had not been pre-emptively implemented.

As the system was designed to create an evenly distributed sphere, it was important to test its accuracy. To evaluate the mesh, two crucial areas needed to be investigated. Firstly, the edges of the terrain faces were scrutinised, as any gaps would lead to visual artefacts. Secondly, it was important to analyse how the mesh deformed when terrain was implemented.

The first concern emerged during the earliest iterations of the implementation. Often, large gaps would feature in the mesh alongside jumps in terrain height. However, by the final iteration, this issue had been completely resolved. As shown in both figure 13 and figure 14, the transition between the terrain and sphere faces were seamless in the final implementation. Even where the errors used to specifically occur along big changes in terrain altitude, the faces now lined up with this new method.

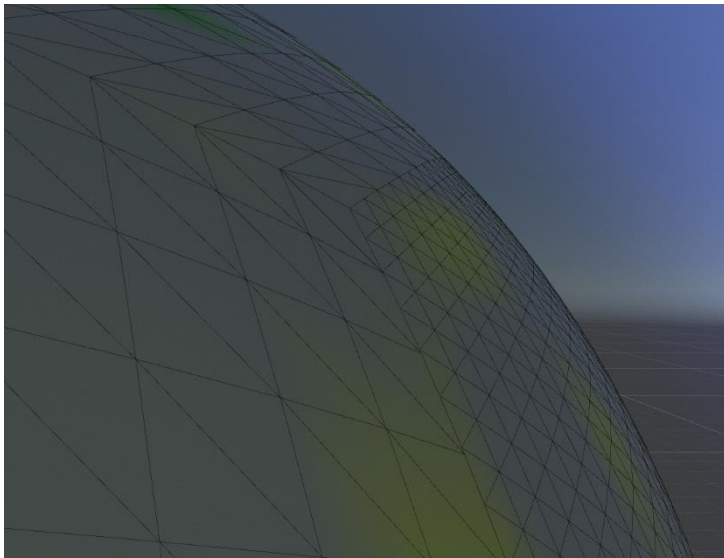


Figure 13: Screenshot of the generated sphere mesh, demonstrating the seamless transition between the different resolution faces

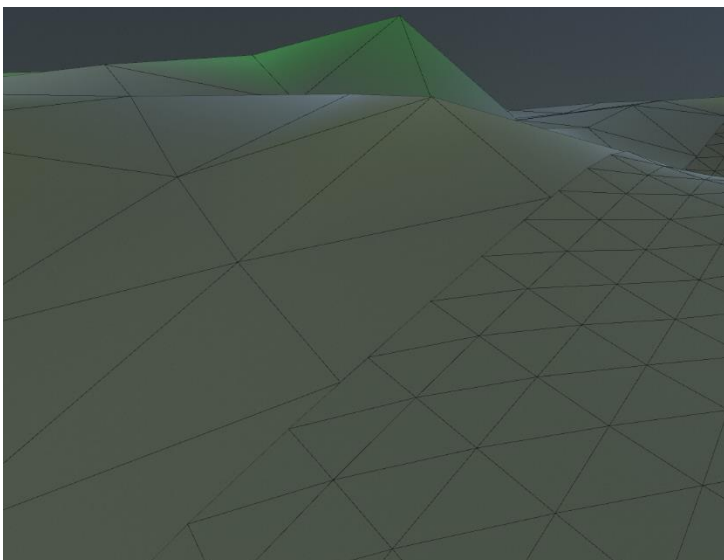


Figure 14: Screenshot of the generated planetary terrain mesh, demonstrating the seamless transition between the different resolution faces

4.3 Planet Analysis

The planet's diversity was also dictated by the terrain and biome diversity which had been implemented. As a result, it was important to analyse the visuals generated by these algorithms by comparing them to the existing implementations within the industry.

4.3.1 Terrain Analysis

To analyse the terrain, it was important to observe it both within a close proximity, and at a distance. The biomes were also evaluated based on their effectiveness in adding variety to the overall planet's terrain. Firstly, it was important to observe the terrain at varying resolutions. While the detail of the planet would be created from a higher resolution mesh, it was also important to measure the quality of the low-resolution version. This was because, this is how the planet will be perceived from a distance.

As seen in figure 15, the higher the mesh resolution, the more terrain features are added. One example includes the canyon and peaks which develop out of the planet's rough geometry. Although this terrain may seem rudimentary at first, when compared to the terrain used within *Elite Dangerous*, similarities can be drawn. As seen in figure 16 the overall curvature and shape of the terrain is largely similar to this implementation. Some of the peaks particularly resemble those in figure 15. The key difference between the two implementations is *Elite Dangerous*' additional noise layers, textures, and noise maps. Each of these elements work together to add more detail at higher resolutions.

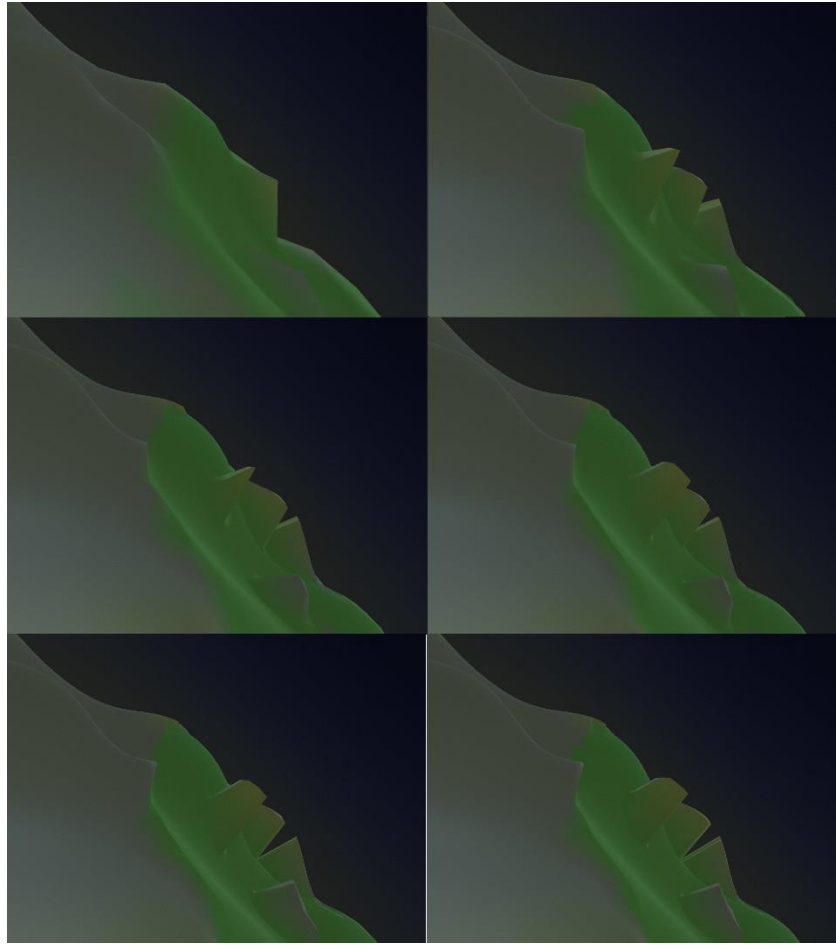


Figure 15: Screenshots demonstrating how increasing the resolution setting, refines the terrain and creates more terrain features. Top-left is the lowest resolution, and bottom-right is the highest resolution.



Figure 16: Screenshot showing terrain example from *Elite Dangerous*, for the purpose of comparing against this project's results (Milne 2019)

The biome system added variety and diversity to the generated terrain. As highlighted in figure 17, the more biomes added to the planet, the more details began to emerge. As each biome was added, additional terrain features and colours appeared. This refinement can be clearly observed in the last image of figure 17, where a grey hue emerged, symbolising the surfacing of the new mountain biome.

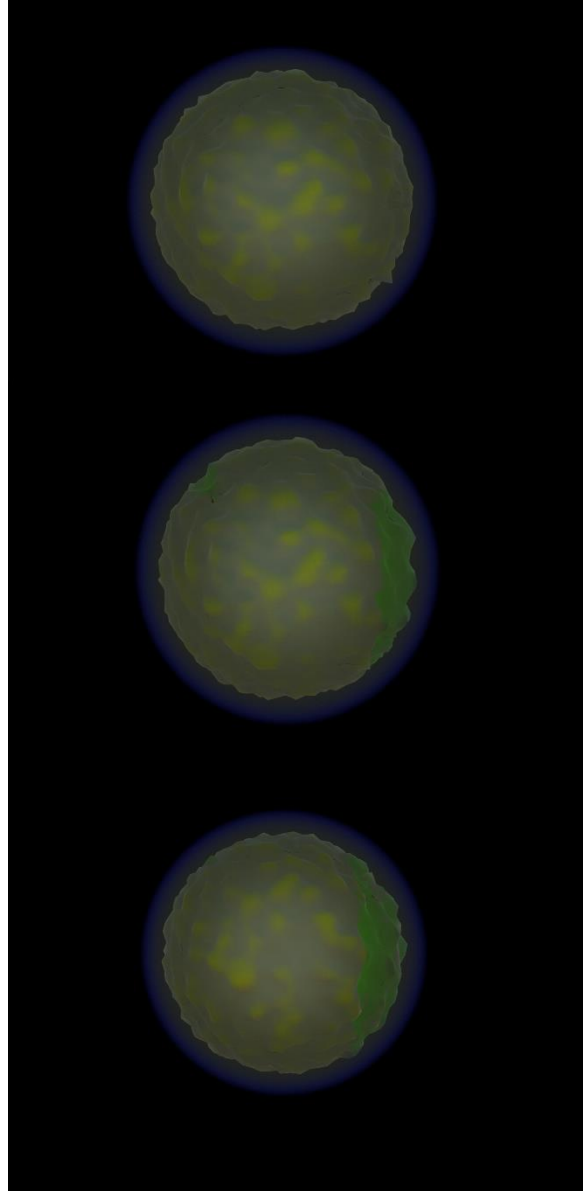


Figure 17: Screenshots showing (from top to bottom) how the addition of biomes influences how the terrain looks and generates

However, this implementation did have drawbacks. During testing, the algorithm could not reliably produce a planet with more than three discrete biomes. This did somewhat limit the planet's variety. It is, however, theorised that this system required very specific, non-overlapping ranges for the biome parameters. This would then allow for a diverse dispersal of biomes. Additionally, some inaccuracies when generating the biome map were encountered. These issues led to clear seams between the biomes on the edges of the unit cube faces.

4.3.2 Scale

Due to the large scales involved in the project, it was important to investigate how the generated planets looked and functioned whilst being scaled up. This was crucial to the output of the project, as a core aim of the implementation was being able to create planets to a realistic scale. As such, several comparisons were created across several different radii. This allowed for the relationship between planet size

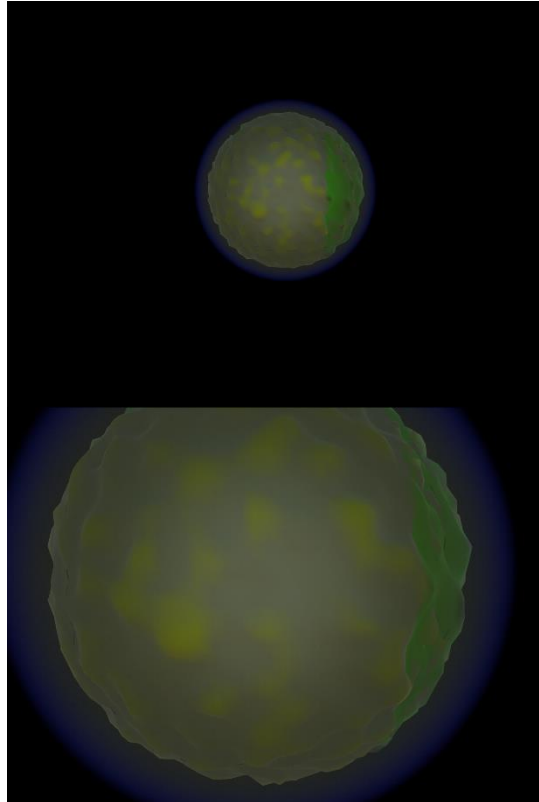


Figure 18: Screenshots showing (from top to bottom) how the increase in radius influences how the terrain looks and generates and terrain generation to be evaluated. As seen in figure 18, the radius of the planet did not affect the way that the terrain and planet were generated. The smaller scale model featured the same terrain features present on the earth scale model.

5.0 Conclusion

Overall, the final implementation was partially successful in achieving the original aim. Whilst succeeding in creating a procedural planet generation system, the full scale and realism could not be realised. Primarily, this was caused by issues implementing the LOD system. Further details on the project's strengths and limitations have been detailed below. This section also details areas for the project's future improvement.

As the project was mostly successful, many high-quality features have been included within the final implementation. Most importantly, this project was able to successfully generate a planetary body to a user specified size. The addition of scriptable objects ensured that interacting with the project's architecture remained easy for the end user, whilst allowing for the objects to be used across multiple planets and games, using this system. This modularity was enhanced by its easily mouldable design. As almost all of the parameters, including the mesh generation shader, were exposed within the `Hurst_PlanetGenerator` script, the end user could easily write shaders tailored to their specific needs.

Another major strength for the project's realism was the inclusion of its atmosphere system. This system worked flawlessly, allowing for a more immersive game world for the planets and the player. Additionally, the implementation of the biome system worked efficiently and allowed for increased variety within the generated planet.

Working on the project was both challenging and rewarding, allowing for the development of new skills in procedural content generation, biome generation, mesh generations and the use of HLSL to create compute shaders. Developing these skills allowed for the generation of the final executable and the final Unity package.

The final executable allows the player to explore a procedurally generated planet and works to demonstrate the technology created for the project. The package has been designed to be easily dropped into a pre-existing Unity project that uses the universal render pipeline and used with little setup. The package also features various objects to configure the planet without using code, making it easy for any end users to interact with.

5.1 Future Development

Despite the many successes, the implementation could have been improved in several areas.

Firstly, as touched on previously, a working LOD system would have benefited the final implementation. Although it was included within the initial scope of the project, several issues were encountered during the implementation of the feature. This resulted in the feature being entirely omitted. This system would have allowed for much more realistic terrain detail, as the resolution of the mesh would have dynamically increased based on the camera's relative position. This addition would also have allowed for the planet's realistic scale to have been properly implemented, as it would have been possible to cull any unnecessary geometry.

Furthermore, the terrain and biome systems would have benefitted from further refinement. Implementing a more realistic approach to biome generation would have been beneficial. This would have included accurate wind and precipitation simulations which would have allowed for the biomes to have been realistically modelled. Further simulations such as continent simulations and asteroid collisions would have also refined the immersion of this system. Another small but crucial amendment which would have improved the overall terrain quality would have been the addition of multiple layers of noise. This would allow for the generation of highly detailed meshes, heightened by a working LOD system. Overall, these improvements would have resulted in more texture within the terrain, further aiding player immersion.

As well as developing these existing components, additional systems could be implemented in the future to further develop the project. One addition would have included breaking the terrain into chunks. Whilst this would have meant moving away from a purely procedural approach, this would have, in theory, increased performance drastically while allowing for the addition of more advanced terrain features. One such inclusion could have been a dynamic rendering approach. This approach would have involved using the usual rendering techniques from a distance, switching to marching cubes as the player approaches the surface. This inclusion would open the possibilities for adding overhangs, and the ability for the player to directly manipulate the planet's terrain.

6.0 Works Cited

armDeveloper, 2022. Real-time 3D Art Best Practices: Geometry [online]. armDeveloper. Available from: <https://developer.arm.com/documentation/102448/0100/Triangle-and-polygon-usage> [Accessed 12 Apr 2022].

Becher, M., Krone, M., Reina, G. and Ertl, T., 2017. Feature-based volumetric terrain generation. In: Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games [online]. Presented at the I3D '17: Symposium on Interactive 3D Graphics and Games, San Francisco California: ACM, 1–9. Available from: <https://dl.acm.org/doi/10.1145/3023368.3023383> [Accessed 20 Apr 2022].

Bonet, R., 2020. Are You Pushing Too Many Vertices to Your GPU? Careful There... [online]. GameDev.net. Available from: <https://gamedev.net/tutorials/programming/general-and-gameplay-programming/are-you-pushing-too-many-vertices-to-your-gpu-careful-there-r5411> [Accessed 19 May 2022].

Building Worlds in No Man's Sky Using Math(s), 2017. Available from: <https://www.youtube.com/watch?v=C9RyEiEzMiU> [Accessed 19 Mar 2022].

Cabot, S., 2020. Exploring the Myth: Calculating Square Root is Expensive [online]. DEV Community. Available from: <https://dev.to/iamscottcab/exploring-the-myth-calculating-square-root-is-expensive-44ka> [Accessed 7 May 2022].

Cajaraville, O. S., 2019. Four Ways to Create a Mesh for a Sphere. Medium [online]. Available from: <https://medium.com/@oscarcs/four-ways-to-create-a-mesh-for-a-sphere-d7956b825db4> [Accessed 2 Feb 2022].

Coding Adventure: Atmosphere, 2020. Available from: <https://www.youtube.com/watch?v=DxfEbulyFcY> [Accessed 13 May 2022].

Coding Adventure: Procedural Moons and Planets, 2020. Available from: <https://www.youtube.com/watch?v=IctXaT9pxA0> [Accessed 10 Feb 2022].

Elek, O., 2009. Rendering Parametrizable Planetary Atmospheres with Multiple Scattering in Real-Time, 8.

Finkel, R. and Bentley, J. L., 1974. Quad Trees A Data Structure for Retrieval on Composite Keys. [online], 4. Available from: https://www.researchgate.net/publication/220197855_Quad_Trees_A_Data_Structure_for_Retrieval_on_Composite_Keys [Accessed 19 Apr 2022].

Fischer, R., Dittmann, P., Weller, R. and Zachmann, G., 2020. AutoBiomes: procedural generation of multi-biome landscapes. The Visual Computer, 36 (10–12), 2263–2272.

Heaven, D., 2016. When infinity gets boring: What went wrong with No Man's Sky [online]. New Scientist. Available from: <https://www.newscientist.com/article/2104873-when-infinity-gets-boring-what-went-wrong-with-no-mans-sky/> [Accessed 13 May 2022].

Hohl, 2022. How Big Are The Planets In No Man's Sky? [online]. AllGamers. Available from: <https://ag.hyperxgaming.com/article/1805/how-big-are-the-planets-in-no-mans-sky> [Accessed 25 Apr 2022].

Hoppe, H., 1998. Smooth view-dependent level-of-detail control and its application to terrain rendering. In: Proceedings Visualization '98 (Cat. No.98CB36276). Presented at the Proceedings Visualization '98 (Cat. No.98CB36276), 35–42.

Hoppe, H., 2004. Geometry Clipmaps, 8.

Keinert, B., Innmann, M., Sanger, M. and Stamminger, M., 2015. Spherical fibonacci mapping. ACM Transactions on Graphics, 34 (6), 1–7.

Kumar, J. M., Herger, M. and Dam, R. F., 2022. Bartle's Player Types for Gamification [online]. The Interaction Design Foundation. Available from: <https://www.interaction-design.org/literature/article/bartle-s-player-types-for-gamification> [Accessed 25 Apr 2022].

Lagae, A., Lefebvre, S., Cook, R., DeRose, T., Drettakis, G., Ebert, D. s., Lewis, J. p., Perlin, K. and Zwicker, M., 2010. A Survey of Procedural Noise Functions. Computer Graphics Forum, 29 (8), 2579–2600.

Lee, J., 2015. No Man's Sky and the Future of Procedural Games [online]. MUO. Available from: <https://www.makeuseof.com/tag/no-mans-sky-future-procedural-games/> [Accessed 25 Apr 2022].

Logical Increments, 2021. Information About Frame Rate [online]. Logical Increments. Available from: <https://logicalincrements.com/articles/framerate> [Accessed 19 May 2022].

Mega Voxels, 2019. What is a voxel? [online]. Available from: <https://www.megavoxels.com/2019/08/what-is-voxel.html> [Accessed 4 Mar 2022].

Michelic, F., 2019. Real-Time Rendering of Procedurally Generated Planets, 8.

Milne, C., 2019. At the end of a five-month Elite Dangerous expedition, I looked into the Abyss. Rock, Paper, Shotgun [online]. Available from: <https://www.rockpapershotgun.com/elite-dangerous-expedition-diary-part-3> [Accessed 19 May 2022].

Montgomery, N., 2008. Floating Point error – what, why and how to!! In: [online]. Presented at the PHUSE 2008, PHUSE, 20. Available from: <https://www.lexjansen.com/phuse/2008/cs/CS08.pdf> [Accessed 25 Apr 2022].

- NASA, 2022. Imagine the Universe! [online]. Available from: https://imagine.gsfc.nasa.gov/features/cosmic/earth_info.html [Accessed 24 Apr 2022].
- Newman, T. S. and Yi, H., 2006. A survey of the marching cubes algorithm. *Computers & Graphics*, 30 (5), 854–879.
- O’Neil, 2022. A Real-Time Procedural Universe, Part Three: Matters of Scale [online]. Available from: https://www.gamasutra.com/view/feature/131393/a_realtime_procedural_universe_.php [Accessed 4 Feb 2022].
- Patel, A., 2022a. Delaunay+Voronoi on a sphere [online]. Available from: <https://www.redblobgames.com/x/1842-delaunay-voronoi-sphere/> [Accessed 4 Feb 2022].
- Patel, A., 2022b. Procedural map generation on a sphere [online]. Available from: <https://www.redblobgames.com/x/1843-planet-generation/> [Accessed 4 Feb 2022].
- Rose, T. and Bakaoukas, A., 2016. Algorithms and Approaches for Procedural Terrain Generation - A Brief Review of Current Techniques. In: . 1–2.
- Savage, M., 2017. Geometry clipmaps: simple terrain rendering with level of detail - mikesavage.co.uk [online]. mikesavage.co.uk. Available from: <https://mikesavage.co.uk/blog/geometry-clipmaps.html> [Accessed 12 Apr 2022].
- Schafhitzel, T., Falk, M. and Ertl, T., 2007. Real-Time Rendering of Planets with Atmospheres, 8.
- Schneider, J., 2006. GPU-Friendly High-Quality Terrain Rendering. *Journal of WSCG*, 8.
- Scratchapixel, 2022. Value Noise and Procedural Patterns: Part 1 [online]. Scratchapixel. Available from: <https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/procedural-patterns-noise-part-1> [Accessed 18 May 2022].
- Sears, D., 2022. Why are planets round? [online]. *Scientific American*. Available from: <https://www.scientificamerican.com/article/why-are-planets-round/> [Accessed 2 Mar 2022].
- Shankar, R., 2021. Difference Between float vs double Data Types [Updated] [online]. Hackr.io. Available from: <https://hackr.io/blog/float-vs-double> [Accessed 18 May 2022].
- Sin, Z. P. T. and Ng, P. H. F., 2018. Planetary Marching Cubes: A Marching Cubes Algorithm for Spherical Space. In: *Proceedings of the 2018 the 2nd International Conference on Video and Image Processing* [online]. Presented at the ICVIP 2018: 2018 the 2nd International Conference on Video and Image Processing, Hong Kong Hong Kong: ACM, 89–94. Available from: <https://dl.acm.org/doi/10.1145/3301506.3301522> [Accessed 21 Feb 2022].

SpaceGamerUK, 2017. Frontier Is Missing the Point With Elite: Dangerous 'The Commanders' Addition [online]. Game Skinny. Available from: <https://www.gameskinny.com/nprjv/frontier-is-missing-the-point-with-elite-dangerous-the-commanders-addition> [Accessed 18 May 2022].

Stam, J., 2003. Real-Time Fluid Dynamics for Games, 17.

Unite 2013 - Building a new universe in Kerbal Space Program, 2013. Available from: <https://www.youtube.com/watch?v=mXTxQko-JH0> [Accessed 10 Feb 2022].

Vitacion, R. J. and Liu, L., 2019. Procedural Generation of 3D Planetary-Scale Terrains. In: 2019 IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT). Presented at the 2019 IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT), 70–77.

Witman, E., 2022. How much RAM do you need? How to tell and when you should upgrade your computer storage [online]. Business Insider. Available from: <https://www.businessinsider.com/how-much-ram-do-i-need> [Accessed 19 May 2022].