# Abstract

Contents

[Abstract 1](#_Toc102824436)

[1.0 Introduction 3](#_Toc102824437)

[1.1 Aims 4](#_Toc102824438)

[1.2 Objectives 4](#_Toc102824439)

[2.0 Literature Review 6](#_Toc102824440)

[2.1 Mesh Generation 6](#_Toc102824441)

[2.2 Level of Detail 7](#_Toc102824442)

[2.3 Floating Point Errors 8](#_Toc102824443)

[2.4 Procedural Terrain Generation 8](#_Toc102824444)

[2.4.1 Noise 8](#_Toc102824445)

[2.4.2 Data Structures 9](#_Toc102824446)

[2.5 Atmospheric Rendering 9](#_Toc102824447)

[3.0 Design and Implementation 10](#_Toc102824448)

[3.1 Mesh Generation 10](#_Toc102824449)

[3.1.1 Sphere Algorithm 10](#_Toc102824450)

[3.1.2 11](#_Toc102824451)

[4.0 Testing 12](#_Toc102824452)

[5.0 Evaluation 12](#_Toc102824453)

[6.0 Conclusion 12](#_Toc102824454)

[7.0 Works Cited 13](#_Toc102824455)

# Introduction

This project sets out to research and implement a system that can procedurally generate realistically sized planetary bodies. The realistic scale is based on earth’s parameters, creating a planet which is 6,378 kilometres in radius (NASA 2022).

The implementation and research for this project follows 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 some aspects of procedural content generation. In addition, these examples each feature procedural planet generation. However, this feature is often relegated to a supplementary system in their gameplay, for example, in *Space Engineers* and *Elite Dangerous* (citation). Other games instead apply the concept to every aspect of the game, making it a part of the core gameplay loop, for example, in *No Mans 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 can be considered boring (Kumar et al. 2022). This is due to mainstream players preferring to have a diverse experience over an empty realistic one (citation). On the other hand, *No Man’s Sky* features highly detailed, varied planets which are kept to a small scale to suit this demographic (Hohl 2022). Consequently, this dissertation aims to resolve these issues, uniting the advantages of each approach to create an intuitive tool for developers to aid in the game creation process.

To do this, background reading has been conducted into each core component of the design. Additionally, existing implementations within established games have been evaluated to help define the feature set in the final package.

The implementation has been created within the Unity3D game engine. This is due to previous experience within Unity, as well as the engine’s existing compute shader system and its accessibility for the end user. Because 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.

To analyse the product, a testing setup will be created within Unity. This is 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 will be visually compared to others within the industry (for example, terrain diversity and realism), and quantitative factors such as memory usage and frame rate will also be measured.

In the following literature review, each of the core developmental components will be discussed in detail. This will include a critical evaluation of seminal and contemporary readings across the areas of mesh generation (Cajaraville 2019, Schneider 2006, Patel 2022), level of detail systems (Hoppe 2004, Savage 2017), floating point precision errors (O’Neil 2022, Montgomery 2008, Unity 2013), procedural terrain generation (Fischer 2020, Lagae 2010, Michelic 2019) and atmospheric generation (Elek 2009, Schafhitzel 2007, O’Neil 2005).

Building from the literature review, the design and implementation section will discuss how each of the core components have 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 which reviews the research and evaluates the final unity package and game executable. This section also outlines areas for refinement within the implementation, as well as highlighting possible future areas of study within the field of procedural planet generation. // past tense

## 1.1 Aims

The aim of the project is to create a unity package and executable that demonstrates a complete procedural planet generation system which is realistic in features and scale. The executable will allow 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 will all be created using the Unity3D game engine.

## 1.2 Objectives

The objectives of this project are as followed:

·         Generate a sphere with evenly distributed vertices

·         Create a level of detail system to simplify the mesh the further the player is from the planet

·         Design a system to remove any issues that could occur due to floating-point precision errors

·         Create a terrain generation algorithm that is customisable with parameters, that also generates diverse and interesting terrain

·         Implement realistic atmospheres for generated planets.

# 2.0 Literature Review

## 2.1 Mesh Generation

The most basic component of the system that will be implemented as part of this dissertation is the creation and generation of a sphere mesh. This is crucial as all planets are typically a spherical shape, due to gravitational forces pulling material to the centre of the planet (Sears 2022). For this element of the project, there exists a wide variety of techniques and algorithms to make this initial sphere. Such techniques include: UV spheres, normalized cubes, spherified cubes and icosahedron (Cajaraville 2019). These algorithms can have their effectiveness evaluated based on their: computational efficiency, distribution of vertices, and how close the generated vertices are to the unit sphere. A benefit of the both the cube algorithms, is the ease to implement a Quadtree, which can be used as a level of detail system for changing the mesh’s complexity (Schneider 2006).

One additional method is the Fibonacci sphere (Patel 2022). This algorithm allows for more evenly distributed vertices compared to the previously described methods and, as remarked by Keinert et al, 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 would prove computationally difficult (Lague 2020).

Another downside of this approach would be the difficulty for implementing a level of detail systems, caused directly by the generation method of the vertices. One promising technique is called the marching cubes algorithm. The method uses voxels, which is defined as “a value on a regular grid in three-dimensional space”(Anon. 2019). The algorithm works using a set of 8 voxels to form a cube, then generating a triangle based off 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 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

The rendering of a highly detailed planets would require the generation and rendering of billions of vertices every frame if a level of detail system is not implemented. This is additionally important, as the max mesh size in unity (using a 32 bit index buffer) is 4 billon vertices (armDeveloper 2022). One technique for implementing a level of detail 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” (quadtree citation). This is perfect for use in storing predefined heightmap of varying levels of detail. Unfortunately, to implement this algorithm is computationally complex, and difficult to implement with a procedural generation technique (quadtree citation).

Geometric Clip maps are an additional technique to implement a level of detail system. This is a LOD system that, “caches the terrain in a set of nested regular grids centered about the viewer” and is similar to the algorithm implemented with texture clipmapping (Hoppe 2004). The ideas in this paper are then further discussed and implemented by Mike Savage. This blog also discusses further methods of expanding this technique, such as using Geomorphing to transition between level of details more smoothly, as well as how to add features such as terrain skirts to more traditional plane based terrain approaches (Savage 2017). Due to this algorithm relative simplicity, and the fact it is designed to be used with terrain visualization, this is what will be featured in the final product (Savage 2017).

## 2.3 Floating Point Errors

Floating point errors can be defined as the issues that can occur when trying to represent an infinite series of numbers, within a finite set of bits (Montgomery 2008). As floats only have 6 digits of precisions, 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 concerning the game Kerbal Space Program. This talk demonstrates a “Jitter” that occurs, which is a vibrating of the game object, that worsens the further out they bring the test spaceship (Unity 2013). In order to amend theses issues, the Kerbal Space Program developers then describe a solution that moves the player camera and game objects into different game spaces, depending on the current scale that is being dealt with (Unity 2013). Additional methods for dealing with these errors include: using doubles in place of floats, manipulating the view matrix and scaling the planets depending on their distance to the camera (*Unite 2013 - Building a new universe in Kerbal Space Program* 2013; O’Neil 2022). As such, a combination of these methods will be implemented.

## 2.4 Procedural Terrain Generation

### 2.4.1 Noise

In Computer graphics, there are many methods for procedural content generation. One of the more popular techniques within this field is the use of noise functions (Reference needed). 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 function has their own characteristics, such as coherency and distribution. This noise is then used to generate a floating-point value, to be mapped mapped to a relevant evaluation for a terrain mesh to use, but it does have its draw backs. As described by Fischer et al. noise is, “inherently unintuitive way to adjust noise parameters and consequently, the difficulty to create genuinely realistic looking terrain” (2020). Additionally, without using multiple layers of noise, also called fractal noise, the terrain that would be generated would very unrealistic and somewhat repetitive, as shown by the end product of Michelic’s work (2019). However, combining fractal noise with basic weather simulation, as used in autobiomes, can create both realistic and diverse terrains (Fischer et al. 2020). This technique uses an initial layer of fractal noise, then applies some weather simulation to this data to break it up into different biomes. After this step is complete, the algorithm then applies more fractal noise to the terrain, which is specific to the generated biomes. Due to the high-quality output of this method, this is the way in which the terrain will be generated as part of the implementation.

### 2.4.2 Data Structures

When noise is used in conjunction with flat terrain, the data is then typically stored within a two dimensional heightfield which is, “the most common data structure used for storing and rendering of terrain” (Becher et al. 2017). The data stored within these heightmaps are altitudes, used a mesh builder to construct or manipulate a mesh to represent these different heights. Although this method is prevalent across many games and papers, heightmaps suffer from not being able to represent multiple level terrain, such as cliffs and overhangs, due to the 2D nature of how the data stored (Becher et al. 2017). Another additional method that could be used to store the data, that would allow for these more advanced terrain features, are voxels. As previously discussed, voxels can be used to store volumetric 3D data, and essentially works as a three-dimensional heightmap. However due to both these methods requiring a data set, the method that will be implemented as part of this project will be a more algorithmic approach, generating the appropriate terrain at runtime to allow for cooperation with level of detail techniques.

## 2.5 Atmospheric Rendering

To create a more realistic and immersive planetary environment, atmospheres would be a great addition to the framework that is being built. Many source, such as Elek and Schafhitzel et al. all feature a similar technique that solves the problem of efficient atmospheric rendering (2009; 2007). This method works by creating an effect. 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, to be then 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.

# 3.0 Design and Implementation

This section will explore the design and iterative process of implementing: sphere mesh generation, level of detail systems, terrain 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 design philosophy for the implementation of this project was to create a system that is customisable, and easy for the user to implement. As such, it would be broken down into as few scripts as possible making 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 for increased performance which are not designed to be editable for the end user. The other script that is user accessible is the camera script, called Hurst\_PlanetaryCameraController. There will be a focus on using unity scriptable objects for holding the majority of the configuration parameters. These objects store data for the biomes, atmosphere and the planet.

* Basic Flow Chart/Layout of all the classes that will be needed

## 3.1 Mesh Generation

### 3.1.1 Sphere Algorithm

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 1).

This algorithm generates 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.

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

The initial results of the sphere algorithm functioned well (figure 2), creating a good 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, 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.

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. This allows for a face to be generated angled towards the camera, regardless of the player's location in world space. // Add closing section

**3.5 Atmospheric Generation**

Atmospheric generation was the final element to be implemented as it did not directly interface with the terrain generation process. This section contains implementation details about the different shaders and scripts that make up this component of the project.

Comparing the papers explaining atmospheric rendering, written by Schafhitzel et al. (2009) and Elek (2007), both featured identical implementations. However, 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.

The fragment shader and the scattering pre-computation compute shader were taken directly from Lague’s project. 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 Ienumator 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.

**3.5.1 Runtime Operations and Scriptable Objects**

One feature of Lague’s implementation that required updating were 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 Hurst\_PlanetaryCameraController would call the UpdateAtmosphere function in Hurst\_PlanetGenerator. This would then trigger the update within AtmosphereSettings, updating the fragment shader.

Figure 1: Basic Normalized Cube Algorithm

# 4.0 Testing

# 5.0 Evaluation

# 6.0 Conclusion

# 7.0 Works Cited

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

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].

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/@oscarsc/four-ways-to-create-a-mesh-for-a-sphere-d7956b825db4 [Accessed 2 Feb 2022].

*Coding Adventure: Procedural Moons and Planets*, 2020. Available from: https://www.youtube.com/watch?v=lctXaT9pxA0 [Accessed 10 Feb 2022].

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

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.

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., 2004. Geometry Clipmaps, 8.

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].

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

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].

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].

O’Neil, S., 2005. *Chapter 16. Accurate Atmospheric Scattering* [online]. NVIDIA Developer. Available from: https://developer.nvidia.com/gpugems/gpugems2/part-ii-shading-lighting-and-shadows/chapter-16-accurate-atmospheric-scattering [Accessed 6 Feb 2022].

Patel, A., 2022. *Delaunay+Voronoi on a sphere* [online]. Available from: https://www.redblobgames.com/x/1842-delaunay-voronoi-sphere/ [Accessed 4 Feb 2022].

Savage, M., 2017. *Geometry clipmaps: simple terrain rendering with level of detail - mikejsavage.co.uk* [online]. mikejsavage.co.uk. Available from: https://mikejsavage.co.uk/blog/geometry-clipmaps.html [Accessed 12 Apr 2022].

Schneider, J., 2006. GPU-Friendly High-Quality Terrain Rendering. *Journal of WSCG*, 8.

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].

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].

*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].