// TODO

* ~~finish / bin LOD~~
* ~~Finish atmospheres~~
* Testing
* Write evaluation
* Write conclusion
* Finish diss first draft
* Make graphs and tables and diagrams
  + captions
* 2nd draft
* FINAL DISS DONE
* FINAL PRODUCT DONE
* SUBMITTED

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 to a realistic scale. This size is 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 which can 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 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 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 games 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 aims to unite the advantages of each approach. For example, to address the issues of monotonous design, various biome systems have been implemented to add diversity to the planet (Douglas 2016).

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.

To fulfil the aim of the project, several smaller objectives were created. The objective 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 will 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 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 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.

**1.3 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 in the final package.

The implementation has been created within the Unity3D game engine, due to previous experience within Unity. This engine also features an existing compute shader system, as well as being accessible for 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 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 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. These techniques included: UV spheres, normalised cubes, spherified cubes and icosahedron (Cajaraville 2019).

<Insert stolen picture from presentation here>

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.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 |

A benefit of both the cube algorithms is the ease when implementing a Quadtree, 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 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).

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 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 tradeoff 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 comparatively 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.

**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. reminds 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 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 approach uses an initial layer of fractal noise, before applying weather simulation to this data. This 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 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 2D format (Becher et al. 2017).

Another method that would allow for the implementation of more advanced terrain features, are 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 better suited this implementation, generating the appropriate terrain at runtime to cooperate with LOD systems.

**2.5 Atmospheric Rendering**

Atmospheres help to create realistic and immersive planetary environments. Schafhitzel 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.

**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 Hurst\_PlanetaryCameraController 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.

<Basic FlowChart/Layout of all the classes that will be needed>

**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 1).

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 (figure 2), 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.

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.

<FaceGeneration Diagram here>

**3.3 Level Of Detail**

In 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, with the only difference being past LOD 2, where the outline of geometry is removed to limit the total vertex count.

<LOD DIAGRAM HERE>

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

The Hurst\_PlanetaryCameraController script 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 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 would 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.

<biomeGenerationWorkflow diagram here>

**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 were 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 together to create a temperature value between 0 and 1.

<biome temp diagram here>

**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 for performance reasons. However, this method was still complex to implement. 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. The addition of a small random deviation allowed for a realistic simulation. This would repeat a user-defined number of times until the wind had spread across the majority of the cube.

<Wind Propagation diagram here>

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 Perlin 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.

**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.

**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 biomes 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 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.

<terrain generation workflow diagram here>

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.

**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.

The fragment shader and the scattering pre-computation compute shader were taken directly from Lague’s project, which used Raleigh 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 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.

**3.6.1 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 Hurst\_PlanetaryCameraController would call the UpdateAtmosphere function in Hurst\_PlanetGenerator. This would then trigger the update within AtmosphereSettings, updating the fragment shader.

**4.0 Evaluation**

**4.1 Performance Evaluation**

**4.2 Mesh Quality**

**4.3 Output analysis**

**5.0 Conclusion**

* Mostly successful with goals
  + Have been able to generate random procedural planets
  + Output is reliable and

This section is to cover your achievements/findings/outputs in a concise and reflective way. You should be self-reflective looking at how successful you have been, what you did well and what could have been done better. You should also highlight your own personal learning achievements in terms of developing new skills and knowledge. Finally, you should have a brief discussion on how the project could be improved upon further.

A picture containing outdoor, person

Description automatically generated