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 brought together procedural mesh generation, 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 to generating planets quickly and to a realistic scale. This realistic scale is based on earth’s parameters, creating a planet which is up to 6,378 kilometres in radius (NASA 2022). This program also features new tools which can be used by developers to create truly 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 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 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 to have 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 this demographic (Hohl 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 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 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 analyses 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. //

**1.3 Methodology**

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 and 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.

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 features detailed research into each of the core components of the project. The central component of the project is mesh generation. All the other components, excluding atmospheric generation, depend on this first step. This is because the mesh is manipulated by the other components. For example, terrain generation manipulates the vertices of the mesh, changing it into something that resembles realistic terrain.

The LOD system is then used to heighten the detail of the terrain when the player camera is close to the mesh. This is done by increasing the vertex count. Floating point errors must then be taken into account as, at this scale, visual issues such as unusual edge cases and jittering can occur.

The only standalone feature was atmospheric rendering, which added another dimension of realism to the generated planet.

**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, normalised 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 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 billion 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 centred 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 levels of detail 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's relative simplicity, and the fact it is designed to be used with terrain visualisation, 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 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 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 these 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**

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 (citation).

Procedural generation of terrain is important, especially for this project. It allows for the generation of infinite maps whilst not taking up any disk space as it is being generated at runtime. Furthermore, this comparatively short generation time allows development teams more time to focus on gameplay mechanics. Within this field, key areas include: noise, heightmaps and data structures.

**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 its own characteristics, such as coherency and distribution. This noise is then used to generate a floating-point value, to be mapped to a relevant evaluation for a terrain mesh to use, but it does have its drawbacks. 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 be 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 by 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 is 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 sources, 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 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 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.

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

**3.2 Level Of Detail**

* Using an rough approach similar to geometric clipmapping
* Generates rings of triangulated squares at increasing resolutions
* Ignores further away vertices and does not generate them
* As stated in prev section, the only part of the planet that is generated is the relative face that faces the planet

**3.3 Floating Point Precision**

Once the previous features had been implemented, issues began to arise as further features were developed. When working to a realistic scale, the geometry of the planet would begin to disappear and the camera would begin to shake. These symptoms highlighted the issue of 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.

**3.4 Terrain Generation**

Key areas within this field include: noise, heightmaps and data structures. When working in a spherical world space, existing algorithms need to be adapted from their usual 2D heightmap approach. This is so that it functions in the spherical world made by this implementation. Another key element is to make sure that the terrain generation algorithm is able to work efficiently during runtime, due to the scale of the planets and the size of the mesh this could be an issue. Additionally the LOD system requires the use of an algorithmic approach to terrain generation, so that the terrain can be dynamically generated during runtime.

3.4.1 Biome Generation

* Create a biome map initially before rendering of planet to save on runtime computation
* Stored in a list of ints, the values refer to the index of the passed in biome scriptable objects
  + These hold all of the adjustable values discussed in a second
* Process
  + Generate an initial Map of temperature values
    - Generates an initial planet based off the set noise parameters
    - Calculate based off of the distance to the poles and the relative altitude
    - Weights can be adjusted by one of the biome parameters to favour height or distance to the pole more
  + Generate a Wind Map
    - Generate points on a unfolded cube for the wind to originate from
    - Iterativley check the direction of the wind, then propagate the wind in that direction
    - Do this sequentially for each wind source
  + Generate Moisture Map
    - Assume High Altitude, cold areas are the origin of water sources
    - Diffuse moisture based on the wind pattern and the temperature of that location
      * Colder means further diffusion and more moisture, warmer means less moisture and less diffusion
    - Create the Biome map
      * Take the biomes and evaluate each point of the moisture and temperature map to assign a biome to each point

3.4.2 Planetary Terrain Generation

* Part of the mesh generation shader
* Initial layer
  + Generates basic noise from set scale and offset parameters
  + Takes vertices, multiplies by mapped amplitude from noise function
* Apply biome specific noise
  + Reads from biome map and applies additional noise to each of the biomes, based off of the biome parameters
  + Checks then done to fix transitions between biomes
    - Parameter for distance between vertices
    - Bring them closer together if too far apart
* PLanet done :P

**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.

Adam

* Lit review:
* 2.1 You say "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." ok, have you evaluated them based on these properties? Can you put this in a table to present it? I would also suggest a diagram comparing each (I think you showed us one in the presentation?)
* I suggest you clarify how marching cubes relates to this context (it's a technique for conversion from voxel to polygonal formats)
* I suggest you clarify how quadtrees could be used. Again, they're a useful tool which could be relevant, you just need to explain this more up-front I think. Mentioning limitations of graphics hardware might help. You go on to the right conclusion, that it needs to be procedurally generated as you can't store all that data, but the journey to this point needs clarification.
* Geometric clip maps could be clarified
* 2.3 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 - what are these methods, their advantages / disadvantages / limitations / etc? A table could help?
* Generally: try to clearly and concisely describe a technique, then critically analyse it and preferably compare several.
* Design and Implementation
* Don't forget to label your diagrams / Figures - Fig 1 is missing its caption. Include a brief description with this too. Ideally the reader should be able to skim through your dissertation and get a good idea of what's going on by looking at the pictures and reading their captions! 😄

A picture containing outdoor, person

Description automatically generated