SCHOOL OF COMPUTING

UNIVERSITY OF TEESSIDE

MIDDLESBROUGH

TS1 3BA

Procedural Generation of Planets in Real-time

BSc. Computer Games Programming

Jacob Keane

28th March 2014

Supervisor: Tyrone Davison

Second Reader: Wen Tang

SCHOOL OF COMPUTING

UNIVERSITY OF TEESSIDE

MIDDLESBROUGH

TS1 3BA

Procedural Generation of Planets in Real-time

BSc. Computer Games Programming

Jacob Keane

28th March 2014

Supervisor: Tyrone Davison

Second Reader: Wen Tang

ABSTRACT

This project follows the development of a procedural planet generation framework, designed for real-time applications in games & visualization.

The analysis involved research into representations of spherical geometry, subdivision, and level of detail.

Following this, an initial design was created detailing what stages the project would consist of, and how these would work together. These stages were then each researched, designed, implemented, and tested in a cyclic manner.

Testing was done in each of these stages, taking into account performance metrics and the visual quality of the results. At the end of the project, final overall tests, and user feedback on the product was done.

Most of the objectives of the project were met; however, the project was not quite in the state ready to be used in a game. This was due to a number of complex issues involved in having a vast universe in a game, so more work will be required before it becomes an easy to drop in framework/plugin for Unity.

1 CONTENTS

2 INTRODUCTION

2.1 Problem Definition

Content creation has always been an issue for games – Especially for larger worlds, more time is required to create the content, and a lot of storage is needed for it. As storage sizes have increased, game worlds have become increasingly large. However, for games such as space exploration games, the time and storage required to have an entire explorable universe, in which you could fly down to the surface of any planet, is simply unfeasible. However, this is where procedural generation comes in. If the surface and composition of a planet can be believably replicated using a combination of algorithms and replicable pseudo-randomness, then as long as these results can be processed and rendered in real-time, this becomes a non-issue, allowing for generation of an infinite amount of unique planets, with no storage requirements for them.

2.2 Justification of problem

To create a vast number of planets, an extremely large amount of time would be required to make them, and the storage required would be vast. For example, the Milky Way has approximately 100-400 billion planets. If we wanted just 1% of the lower bound of that, we would still need a billion planets. If we assume each planet has its own heightmap, a single 1024x1024 greyscale texture, roughly 953 Terabytes (~200,000 single layer DVDs) would be required to store it, and these planets would also lack sufficient resolution for you to be able to explore the surface of them.

To create a procedural planet generator is a considerable amount of work. This project aims to create a tool that provides planet generation, so game developers do not have to spend time perfecting planet generation, and can focus on gameplay.

2.3 Program requirements

To solve this problem, in this project I aim to develop a framework to be used by game developers that will provide the tools required to generate an entire universe for their game, with no storage requirements more than a simple seed number for the universe generated.

There are few requirements of the generation framework.

1. The planets need to be such quality that they are reasonably believable.
2. The generation needs to work in real-time. It is not acceptable for the game to pause and make the player wait as the planet generates a higher quality mesh for the cliff they are approaching.
3. The framework needs to be easy to use. It should be able to be integrated into a game without significant manual work required, and have easy to understand parameters & settings.
4. The framework needs to be easily expandable, should the user want to change how any part of it works, or add more features, it should be constructed in a way that makes this easy to do.

The following chapters explain the process used to create the project. It was split into several stages, each with its own research, design, implementation, testing, and evaluation.

3 METHODOLOGY

3.1 Design methodology

Unity was chosen as the engine of choice to design the framework for, as it is the common game engine of choice amongst game developers looking for rapid results – These are also the kind of games where procedural generation is most common. It's also a game engine designed with a large focus on dynamic content, as opposed to some other engines that are designed to work with mostly pre-created and pre-baked data, such as the Source engine and Unreal engines.

C# was the language of choice for this project. Out of the three languages Unity supports, UnityScript, C# and Boo. Out of these, C# was chosen for several reasons. Boo was ruled out almost immediately, due to it being a language that I am unfamiliar with, used by <5% of Unity users (Jashan, 2009), and offering no strong advantages.

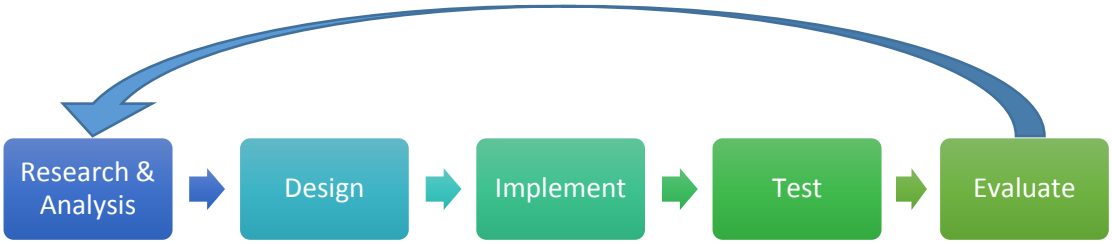
UnityScript's main advantage over C# is that the syntax is slightly simpler and easier to understand.

C# has several advantages over UnityScript. The syntax is more explicit, whereas in UnityScript there are numerous syntactical features that come with hidden performance costs, or can create other issues. (The use of #pragma strict helps to alleviate this issue somewhat). Visual Studio also has full C# support, whereas the support for UnityScript is limited.

I also have a lot more experience programming in C# than UnityScript, so the advantage of UnityScript being simpler becomes moot. The advantages of C# and my greater experience in it was the reason C# was chosen over UnityScript.

3.2 Implementation methodology

The framework was developed using an iterative development process, based upon the Agile software development methodology. The program was split up into its core components and stages, each of which was developed in a cyclic manner; they were individually researched, designed, implemented, tested, and evaluated, going back if needed, until suitable.



3.3 Testing methodology

There are three main factors the project needs to meet, aesthetics/ realism, performance and usability

3.3.1 Aesthetics / Realism

One of the main requirements for the planet generator is a certain level of realism. The planets need to look good enough to not feel out of place in a serious space exploration game.

3.3.2 Performance

Another core requirement is that the generation happens in real-time. This mean there should be no pauses, stuttering, or low frame-rates caused by the generation process. It should happen at a speed suitable for a game.

3.3.3 Usability

Since this is a framework for planet generation, it should be easy to use. Consideration should be taken to design it in a way that you do not have to know the in-and-outs of procedural generation methods to use it. It should be easy to integrate into a game, with minimal manual work required.

3.3.4 Expandability & Versatility

The parts of the framework should each be as versatile as possible. Where it is not suitable for it to be sufficiently versatile, it should be easily expandable / modifiable, for example, it is unreasonable for one surface sampler to be

suitable for all types of planets. However, if the system is designed so that the samplers are easily exchangeable, this not a problem.

4 RESEARCH & ANALYSIS

4.1 Spore

The videogame Spore features an entire galaxy, each planet of which you can visit, even at surface detail. To do this, they used procedural planet generation.

They used a combination of techniques. For their planet mesh, they used a Quadrilateralized Spherical Cube, textured using cubemaps. They used a brush-based system, which raise/lower the terrain height, and procedurally placed these in order to create planet-like features.

For texturing the planet, they used detail and colour maps derived from the height map, and their system also worked to generate real-time LODs. However, the brush system they used, combined with the small size of their planets resulted in unrealistic and ‘cartoon’ looking planets.

(MAXIS, ELECTRONIC ARTS, 2007)

4.2 Procedural planets into detail - Twan De Graaf - 2012

In this paper, QuadSpheres along with Cubemaps were used again. However, there is no real-time generation, the planets are procedurally generated in an offline process, in separate software to the game engine, then imported into it. This means they have to be stored, and thus would not be suitable for a massive galaxy. However, the results, albeit pre-processed, are excellent and if they are feasible in real time, they are a good reference.

(GRAAF, Twan De, 2013)

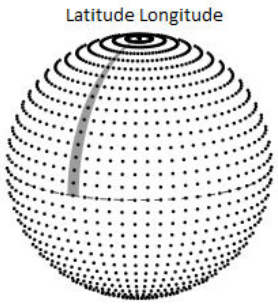
5 ITERATION 1 – MESH GENERATION

5.1 Research

The mesh could be generated in a variety of different ways. The planets will be mostly spherical, but there are numerous different ways of generating spheres. Whichever method is chosen, there are several important properties:

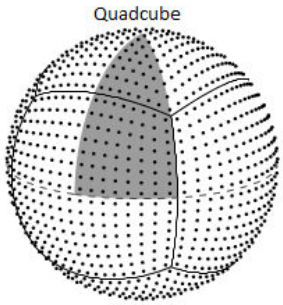
1. The amount of distortion on the sphere. This can be explained by the variation in surface area of a triangle of the mesh, across the sphere. The less variation, the better, as there are various issue that can arise.
2. Complexity – An overly complicated generation method can lead to many issues, especially generating things such as surface normal, tangents, and UV co-ordinates. A more complicated method would also take more time to implement and have more problems that are possible.
3. Subdivision – Some methods are more suited to easy subdivision than others are. Whichever method is chosen, it needs to be able to easily be subdivided, and vary in quality, so level of detail can be implemented, and the parts of the planet closer to the camera can be rendered at a higher quality.

5.1.1 Longitude-latitude sphere



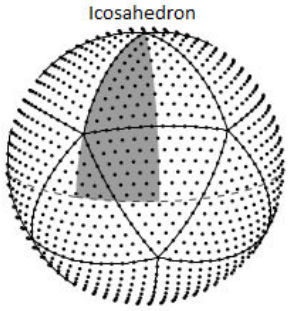
Longitude-latitude is the standard co-ordinate system used for spherical co-ordinates. However, if these are used to generate a sphere, you end up with several issues. A standard longitude-latitude sphere has sphere distortion issues as you approach the poles that cause problems when trying to UV map and generate textures for them. It is also hard to subdivide easily.

5.1.2 Quadrilateralized Spherical Cube (QuadCube)



This method has several advantages. Whilst it still has some distortion, it is minor, as it is spread across the eight corners of the cube, as opposed to severe distortion at two poles that the longitude latitude sphere suffers from. It's fairly simple to implement, and using a cube as a base makes chunking for level of detail easier as each cube faces is just a square, and this simplifies a lot of issue. For example, each square could easily be recursively subdivided in a quad tree for level of detail.

5.1.3 Geodesic grid / Icosahedron



An alternative method for sphere generation is using a geodesic grid. This is a complicated method in comparison to the other options, but it has an extremely low amount of distortion issues. It however, is hard to UV-map, and subdivision would be more complicated than the quadcube, but still not too complicated. There is also a large amount of possible seams.

5.2 Design

Out of these methods, Quadrilateralized spherical cube was chosen, due to its low distortion and simplicity.

5.3 Implementation

A CubeGen component was created, which generates one side of the cube, based on a side enum to specify which side of the cube should be generated. This was initially written to create a plane at a given resolution, and then was modified to include the QuadCube distortion. It will then orient the distorted surface correctly for the given side, applies the height offsetting, and return the mesh.

5.4 Testing

At this stage, the mesh generator generates an acceptable sphere mesh. However, there is a slight seam in the normals (calculated by unity).

5.5 Evaluation

The mesh generation is good enough to move onto the texture generation, to ensure it works together properly. The normal seam is a minor issue, and this can be fixed in a later iteration.

6 ITERATION 2 – PLANET SAMPLING

6.1 Design:

The formation and representation of a planet's surface is extremely complicated. A system that is simple, yet versatile enough for most use-cases was needed to represent these vast, varying surfaces. Out of the many possible factors that could be used to represent what a given point on the planet's surface could look like, I needed to pick a few variables that could be used to represent most of these.

A large amount of variables possible for the surface were considered, including:

- Height
- Temperature
- Moisture
- Water coverage
- Snow coverage
- Vegetation
- Land type

It was decided that most of these variables could be derived from three of the basic parameters; Height, Temperature and Moisture. Temperature can decide factors like Snow coverage, Height can be used to determine what areas are under water, beaches, normal land or mountains, and moisture can be used to decide whether a point should have grass coverage, dirt, desert, etc.

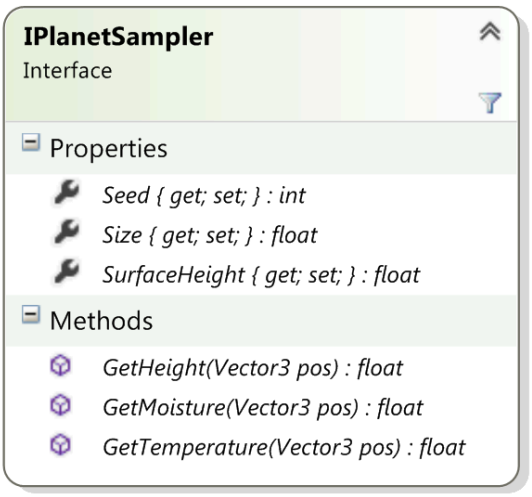


Figure 1 - Interface for the planet sampler

6.2 Implementation:

A generic interface for planet samplers was created, along with several test samplers that implemented it. To create these samplers, a C# port of LibNoise, an open-source coherent noise generation library was used.

Several test samplers were implemented, using Perlin noise, Ridged multifractal noise, and combinations of these, to generate the values for the height, temperature, and moisture levels.

For each of the samplers, as well as implementing the interface for the sampler, they also inherit MonoBehaviour. This allows them to be added to the planet as a component. By using `GetComponent(typeof(IPlanetSampler))` in the CubeGen to find the sampler, this allows us to get a reference to any component implementing the IPlanetSampler interface, and thus easily swap out and change sampler settings from inside the unity editor.

6.2.1 Sampler implementations

6.2.1.1 PlanetSamplerPerlin

Uses a pair of perlin noise generators with different seeds. The first is used for sampling the height, and the second for sampling the moisture. The temperature is created using a custom formula.

The equatorial distance is taken, and multiplied by a ‘sun power’ variable. This affects how much a part of the planet is heated up, based on its distance from the equator. This simulates the equator being hotter, and the poles being colder, due to the angle from the sun. The height of the surface is then taken into account, and points above a certain height become progressively colder. Then, a small random variance is added, taken from the second perlin noise function.

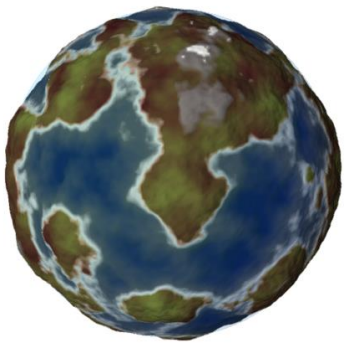


Figure 2 - Perlin planet sampler in use

6.2.1.2 PlanetSamplerRidgedMultifractal

This generated works much the same as the perlin equivalent, except it uses ridged multifractal noise instead of perlin noise. The results looked interesting, but the planets were not realistic

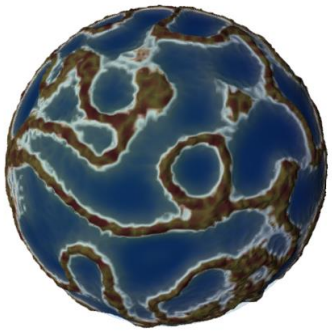


Figure 3 - Ridged Multifractal planet sampler in use

6.2.1.3 PlanetSamplerComplex

This was a test of combining different generators together. The results did not look as good from a distance, but the detail when closer, was more interesting than plain perlin.

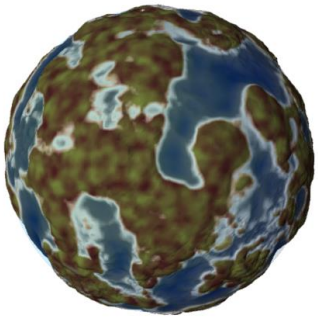
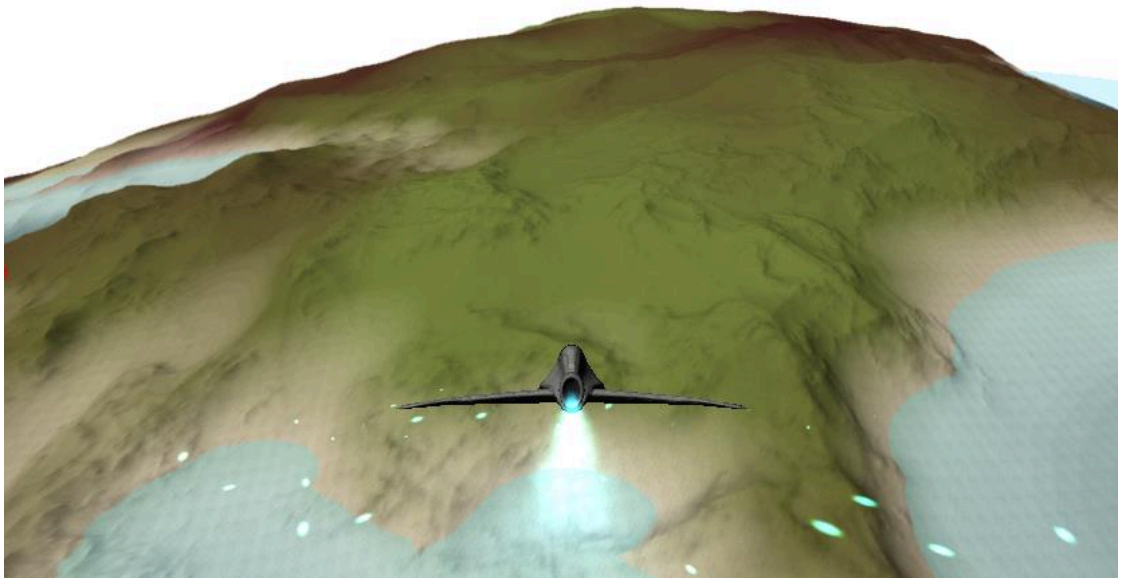


Figure 4 - Complex planet sampler in use



6.2.1.4 PlanetSamplerComplex2

A much more complicated method of combining the generators was used here. This time, Billow noise and RidgedMultifractal noise were again, but this time, a scale bias for each was used, and a selector based on perlin noise.

The selector was used to specify which generator to use, when the perlin selector has a high enough value, the selector would use the ridged multifractal added to the billow, and when not, it would just use the billow.

The aim of this was to use the multifractal noise to generate mountains, and not have the multifractal noise affect the normal, billow ground.

6.3 Testing:

The sampler performance was tested by implementing it in the mesh generator, and profiling the generation speed. It was found that the mesh could be regenerated at roughly 60 fps without problem, when sampled at 128x128 resolution. Since unneeded premature optimization is considered bad practice, it was decided the performance at this point was acceptable.

6.4 Evaluation:

By using a generic interface, combined with MonoBehaviour to use them as a component, this allows the user to easily change the sampler used without having to touch the code. This is excellent for manual changes to a given planet, and debugging inside unity. The interface is versatile enough to suit the majority of use cases.

7 ITERATION 3 - TEXTURING

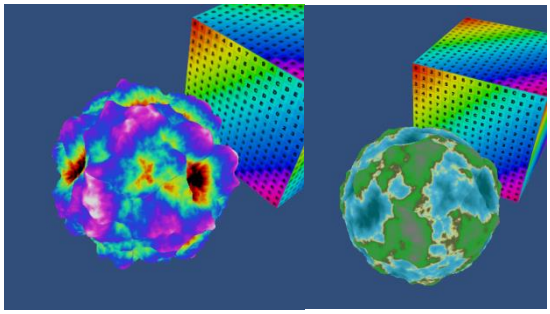
7.1 Revision 1 - Failed experiments with cube-mapping

To texture the base planet, generating the texture for each side as a cubemap, then using the normalized local fragment position to lookup in the cubemap to get the texture co-ordinates was experimented with. The idea behind this system was to avoid having to generate & store UV co-ordinates for the planet at L0 detail. However, I had severe issues with the image distortion causing the texture to not match up with the mesh correctly (Most likely due to the cube->sphere distortion process) – I could not identify the cause, and proceeded to rewrite the texturing code to use classical UV co-ordinates generated, as these would be needed for the subdivision process anyway.

7.2 Revision 2 - Solution: UV generation (the normal way)

I modified the mesh generation process to generate UV co-ordinates for each mesh, going from (0,0) on one corner to (1,1) on the other.

To texture each mesh, originally generating a large texture was being generated, sampling the height for each pixel, looking up that height in a gradient, until each pixel was set to generate the texture, and applying that texture to the mesh, using the pixel colour as the diffuse surface colour.



Whilst this looked okay, it was not suitable for a realistic looking planet, and high-resolution textures would have to be generated for each mesh.

Because of this, it was decided to change the texturing system to allow for use of actual textures for the mesh, instead of just a colour per pixel. A texture splat system was created, in which there was two RGBA base splatmap textures, and each colour channel was used as a lookup value for how much of a texture to apply, and combining the UV coordinates with these values, multiple textures could be splatted onto the mesh.

However, there was an issue where it was interpolating too many values for each vertex, and this caused the pixel shader to run out of instructions.

The system was deciding what texture to use CPU side, and writing these values into two RGBA textures, where each colour channel was how much of a given texture to use.

7.3 Revision 3

The system was modified only use one RGBA texture, pass in the sampler data to the splatmap (Height, moisture and temperature went into the red, green and blue channels, alpha was left unused for future use), and the logic of which texture to use was moved GPU side, into the fragment shader, where these values where analysed, and the blending values were decided based upon those.

8 ITERATION 4 – SUBDIVISION AND LEVEL OF DETAIL

8.1 Design:

When the player gets closer to the planet, the planet should increase in quality, up to ‘surface quality’ level of detail.

There are various ways this subdivision can be achieved, but since a QuadSphere was chosen as the basis for the mesh generation, each mesh at the base level is a square, so this can easily be subdivided in one of many ways. It was decided to use a quadtree to subdivide it into, which would just involve splitting the square into quarters recursively.

8.2 Implementation:

To do this, recursive subdivision of each cube face was implemented. Each mesh can be split up into a QuadTree. This will evenly split the mesh into four quarters, each of which could then be further subdivided if needed.

The parent would request the four children to be generated, and once all four had finished generating, disable the mesh renderer on itself, and enable its children. This allowed the meshes to be swapped out with no visual artefacts other than the ‘pop-in’ caused by the sudden change in quality

Each child would use a sub-rectangle of its parent rectangle, which affected the values sampled for position, etc. causing the smaller region to be generated at the same quality as the parent, resulting in four meshes, the same resolution, but quarter of the size each.

8.3 Testing:

When the subdivision level reached 9-10 recursive subdivisions, floating point precision became an issue – the distances between vertexes became so small that the normal calculations would begin to produce invalid results, frequently producing ‘black’ surfaces that were wrongly lit.

The solution to this issue was to multiply in the planet scale during the vertex generation process, instead of scaling the resulting planet after. This allowed the normal calculations to be performed on much larger values, meaning the floating-point precision was not a problem until many more subdivisions.

9 ITERATION 5 – MULTITHREADING

9.1 Changes made:

At this point in the project, the subdivision worked, and the planet would get higher in quality as you got closer. However, this process was all running on the main thread, causing the application to halt as the meshes and textures were being generated. This was clearly not acceptable for real-time applications, so severe code refactoring was done, allowing it to be multithreaded.

However, there are some severe issues writing multithreaded code in Unity. The Unity API is not thread-safe, and thus most functions from it are blocked if they are not called from the main thread. This meant that it was not possible to create a new Mesh or Texture2D from my side threads. To get around this issue, the data that would be contained inside them was generated on a separate thread, then once that was done generating, wrapped it up inside these containers on the main thread.

The mesh generator was modified to return arrays of positions, normal, UV co-ordinates, etc. instead of returning a mesh.

The texture generator was modified to return an array of Color32, instead of a Texture2D.

9.2 Testing:

After offloading the generation onto separate threads, the performance was greatly improved. The simulation no longer froze briefly when the mesh/texture generation was happening, but there was still some slight jitter in framerate when the meshes were swapped out.

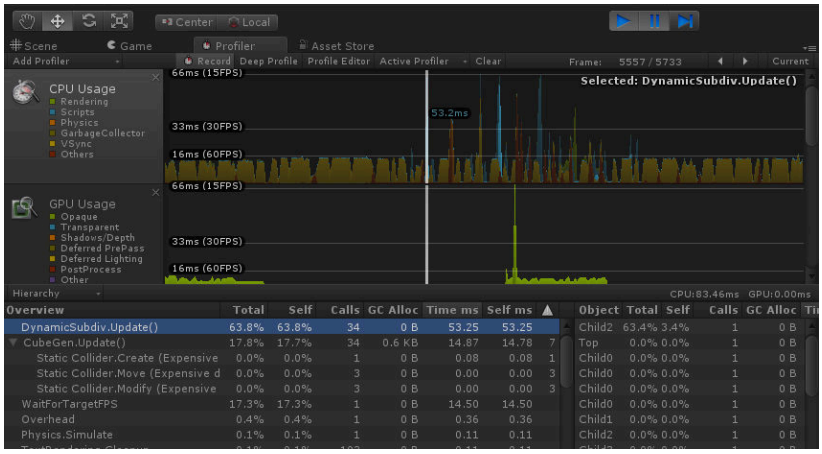


Figure 5 - Unity profiler showing jitter in framerate

However, upon building and running the executable, this jitter was greatly reduced, to the point where it was almost un-noticeable. However, it could still do with further improvement.

10 ITERATION 6 - FIXES

A number of bugs in the project needed resolving. One of the most severe bugs was the seams between meshes. These were caused by two separate issues.

1. No continuity of surface normal from one mesh to another
2. Differing level of detail on two neighbouring meshes could cause mesh ‘holes’ that could be seen through.

10.1 No continuity of surface normals

10.1.1 Design

To calculate the normal for a given vertex, the positions of the vertex above and to the right of it are used. However, for the edge of the mesh, these positions were not available, due to them being off the edge of the mesh.

To fix these, two extra strips of vertex positions were calculated during the mesh generation, one across the bottom edge, and one across the right edge. These are only used during the normal calculation process, but they allow the far edge to be calculated correctly, resolving the normal seams.

10.2 LOD Mesh holes

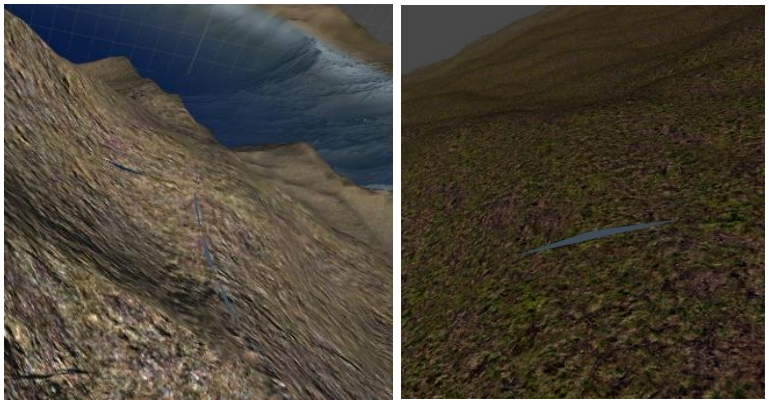


Figure 6 - Mesh hole is visible near the center of the image

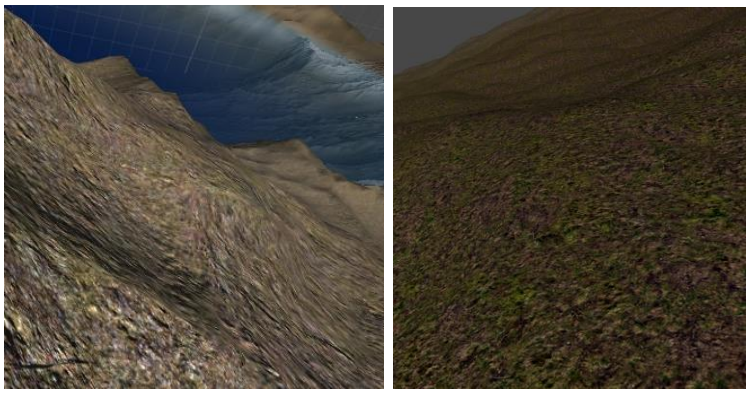


Figure 7 - Mesh hole is now fixed with a mesh skirt

To solve the mesh holes, several methods were considered:

1. Generate the edge vertices of the mesh to match its neighbour. This idea was rejected after initial consideration. It would require either all the neighbouring meshes to be regenerated whenever a mesh changed LOD, or making the new mesh match its neighbours, which would result in the edges outer edges of the quadtree to always be at LOD0 quality, which would not be acceptable. Due to all the extra regeneration required, this idea was rejected.
2. Move the edge vertices of the higher resolution neighbour to match its neighbours, using shaders. This was a viable option, but a large amount of extra work would be required, and a large amount of extra data would have to be passed into the vertex shaders to know which vertexes to move, the LOD of neighbouring meshes, etc.
3. Generate a ‘skirt’ around the mesh, that would fill any holes that may happen to be there, whether it was needed or not. This solution had a slight overhead, which was a slight increase in the size of each mesh, and the amount being rendered.

The method chosen generating a mesh ‘skirt’. This was an extra strip of quads on each side of the mesh, that go straight down from the edge, but use the same normal values etc as the flat mesh. Whilst these normals were technically incorrect for the angle the face was at, this means this vertical edge would be lit the same as the surface, and with appropriate UV co-ords calculated, the result is when you view mesh through where the gap would be, t would look a lot more accurate, due to mesh being filled with an appropriately lit and textured mesh. This was in no way a perfect solution, and has the overhead of always having extra mesh details, but it was a massive improvement over having holes there.

10.3 Rendering issues

When dealing with the immense distances of planetary sizes a number of issues with limited floating point precision occur. One of these is the limited accuracy of the depth buffer.

The camera needs to be able to render anything from nearby objects (Within a few cm of the camera, when you're flying near the surface of the planet), to thousands of units away (When viewing the entire planet)

However, the depth buffer needs to store the depth of everything from the nearest point to the furthest. When dealing with this entire range, the limited precision of the depth buffer causes rendering glitches such as Z-fighting, where two different meshes can pop in front/behind of each other each frame.

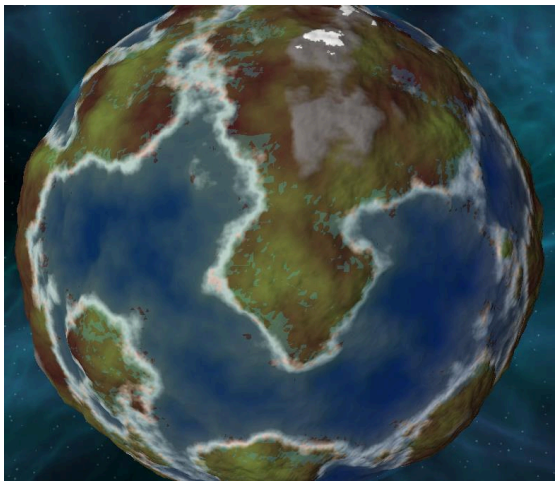


Figure 8 - Planet rendered using pre-set near & far planes (0.1 near, 45000 far) Note the rendering artefacts near the edges of the land where the water mesh appears in front.

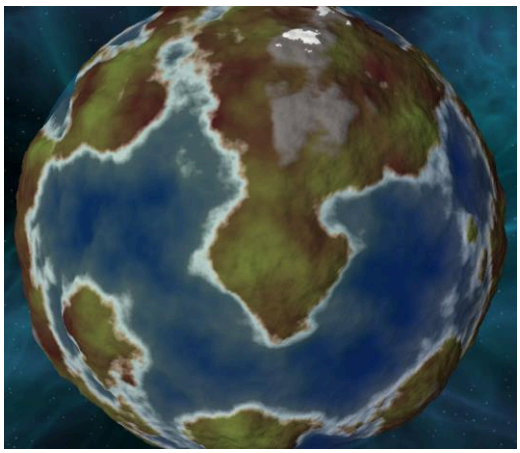


Figure 9 - Planet rendered using dynamically calculated near & far planes (In this case, 140.1253 and 45402.79) Note the lack of artefacts

To solve this issue, I wrote a script to measure the current distance the camera was away from the planet, and an approximation of the surface (Since there can be considerable variation in surface height), that would set the near and far planes based on that. When you are far away from the planet, the near plane is set as a few hundred units, and the far plane far enough to render the entire planet. As you approach the planet, the near plane is pulled closer to the camera gradually, and the far plane is pulled nearer as well. This allows the precision to always be where it is needed, and solved the rendering issues.

However, this solution is only suitable for a single planet being rendered, as the distances are all calculated relative to that. For multiple models / planets, a rendering system that uses multiple camera for rendering things at different distances would be required.

11 ITERATION 7 - OPTIMIZATION

Optimization

In this phase, the focus was on improving the performance of the framework. There were some severe stutter issues, and whilst the average FPS was near high, there would be an occasional frame every few seconds taking upwards of 300ms. This jitter in frame time was unacceptable.

To resolve these issues, performance profiling, and an evaluation the code layout / overall process was performed. When a mesh was subdivided, it would, if there was not too many threads running currently, create a MeshGenerator and TextureGenerator on a new thread, run these until they’re finished, then collect the results and dispose of the Threads, MeshGenerator and TextureGenerator. This caused severe delays during the allocation of Threads, memory for the MeshGen/TextureGen, and garbage collection on them. To solve this issue, I created a pool of MeshGenerators and TextureGenerators at start-up, with the arrays already allocated, and used ThreadPool.QueueUserWorkItem instead of creating a new Thread, to alleviate the overhead of creating a new thread each time. These changes significantly reduced the frame time spikes.

12 ITERATION 8 – FEATURES

12.1 Water

One of the most recognisable features required for an earth-like planet is water. About 71% percent of the earth's surface is covered in water, so without this, it would be hard for a planet to be considered earth-like.

12.1.1 Design

The surface height of a planet does not stop at the surface of the water, and in reality, there surface goes below the water. Because of this, it was decided to make the water a separate layer, and have terrain below the water.

12.1.2 Implementation

I decided to generate a second sphere, and use this mesh as the water surface. It was rendered along with the planet

12.1.3 Testing

The water mesh made the planet look a lot better from a distance, but the area below the water looked wrong with the simple water shader used. To fix this, I added a few extra instructions to interpolate the diffuse colour of the surface towards a dark water colour, to simulate the light scattering of dense water. This made it look a lot more visually appealing.

12.1.4 Evaluation

The system implemented has a number of flaws – The water mesh used was fairly high resolution, when it didn't really need to be high resolution throughout. At a far distance, a fairly low resolution would have sufficed, and when closer to the surface, only the nearby parts of the water would have to be high resolution. To solve this issue, a method similar to how the planet mesh is generated could be used, and have the water be sub-divided and level-of-detailed, but this was not too much of an issue in testing. The extra rendering required for the sphere mesh was not too much, and at the time, a simple shader was used on it, although this should be changed to a more realistic shader, as an ocean surface is complex. Water is definitely an area that needs a lot more attention.

12.2 Surface level details part 1 - Grass

One of the recognisable features of an earth like planet is the grass coverage on the ground. From a perspective near to the ground, this will have a large visual impact on how believable the terrain is.

Due to using a custom terrain system, using Unity's build in grass system for it was not possible, so a custom system must be created.

12.2.1 Research

I did some research into methods used to render grass in games. In GPU Gems, Chapter 7 (NVIDIA, 2007), they offer a method for rendering a large amount of grass. A 'star shaped' mesh of quads can be used, each textured with a grass texture. With a large number of these, closely placed together, the impression of thick, full grass is given.

12.2.2 Design

The grass system was mostly just created as a test. However, it still went through several revisions:

12.2.3 Revision 1:

12.2.3.1 Implementation

Each time the CubeGen subdivides, the Populate function of the GrassGenerator is called. This checks if the CubeGen's current subdivision is low enough to generate grass yet. If it is, it will perform similar math to that which the MeshGenerator and TextureGenerator use to sample the points (This should probably be refactored into some generic code). It will do this for a given resolution inside the specified subdivision level for the CubeGen. Once each of these points is calculated, they are stored in array of Vector3s. Graphics.DrawMesh is then used to draw the given grass mesh at each one of the points.

12.2.3.2 Testing

Upon testing this method, there were severe frame rate issues. The unity stats window showed an extremely large amount of draw calls. Upon doing some research into Graphics.DrawMesh, it appears that you cannot do any kind of draw call batching when using this method.

12.2.4 Revision 2:

12.2.4.1 Implementation:

Most of the implementation stayed the same in the second revision. However, changes were made to how the grass was drawn. Instead of using Graphics.DrawMesh, a new GameObject was created for each grass mesh, and positioned at the generated positions.

12.2.4.2 Testing

Performance increased greatly after this change. The draws calls dropped significantly and the batched draw calls increased significantly. However, there was almost a second pause as 1024 gameobjects were all being created in one frame.

12.2.5 Revision 3:

12.2.5.1 Implementation:

A simple modification was made to spread out the generation across multiple frames. I changed the function to be a unity co-routine; by changing the void function to an `IEnumerator`, and used `yield return null` inside for the loop, when the was called again, it would continue from the last point it got to, thus spawning them in 32 at a time, row by row.

Figure 10 - Old fill order (one row at a time)

However, visually, having the grass appear one row at a time was not too appealing, so another modification was made to change the order they were instantiated. Instead of doing a row at a time, it placed an evenly distributed set of the meshes each frame.

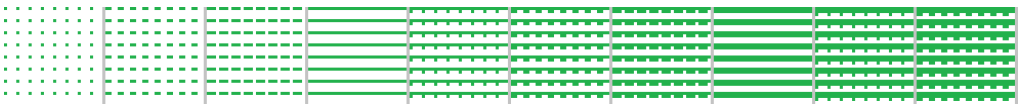


Figure 11 - New fill order

12.2.5.2 Testing:

With these changes, the grass would draw, and be instantiated with acceptable performance. There was a slight degradation of performance compared to no grass, but this was due to there being more drawn. The grass added a significant amount of polygons to be drawn, and this performance hit was expected.

12.2.6 Evaluation

There are several flaws with this design. The fact the GrassGenerator is connected to the level of subdivisions, and not the size of the subdivision is a problem. This means for a much larger planet, the grass generation would start earlier, and later (or never) for a smaller one. Since this number was currently set manually for a planet, this was not too much of a problem, but when set dynamically, a better system should be used. This would need more improvement later, but this was just a simple test to see how practical grass rendering was at surface level.

It also may be a better implementation to create one mesh for the grass, containing all of the smaller grass meshes as one large mesh. Performance comparisons with the current method would have to be done to determine the better method.

Another issue is the grid-like arrangement of the grass is apparent. This may just need a better grass model to resolved, or the density of the grass may need adjusting. If these do not solve the issue, a slight random variance in the grass position may help solve this issue, although care would have to be taken not to place the grass so that it was not attached to the ground.

The instantiation pattern is also still not optimal. Whilst the positions are created spread evenly, the order they are placed in is still not even, and goes from the top left, to the right, down a row, then repeats, until the end is reached. A more optimal even distribution could be implemented, but the math for this would be complicated.

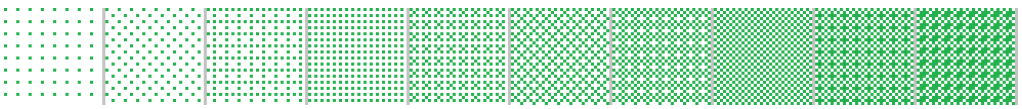


Figure 12 - Example of a more optimal instantiation order

12.3 Surface level details part 2 – Trees, rocks, plants and other details

12.3.1 Design:

The aim for this part was to create a system for placing assets on the terrain that was as versatile as possible. Upon analysing what the requirements for this would be, I made several decisions:

- Due only being able to have one component of a given type on a gameobject, one component had to handle any assets that would be placed.
- For any asset to be placed, a set of placement requirements must be met
- Once these requirements are met, there may be a variety of different assets that could be placed. For example, a tree placer may place one of several tree models.

From these, I produced a list of requirements for the populator:

- The populator can have 1-n things that can be populated, each with its own set of prerequisites.

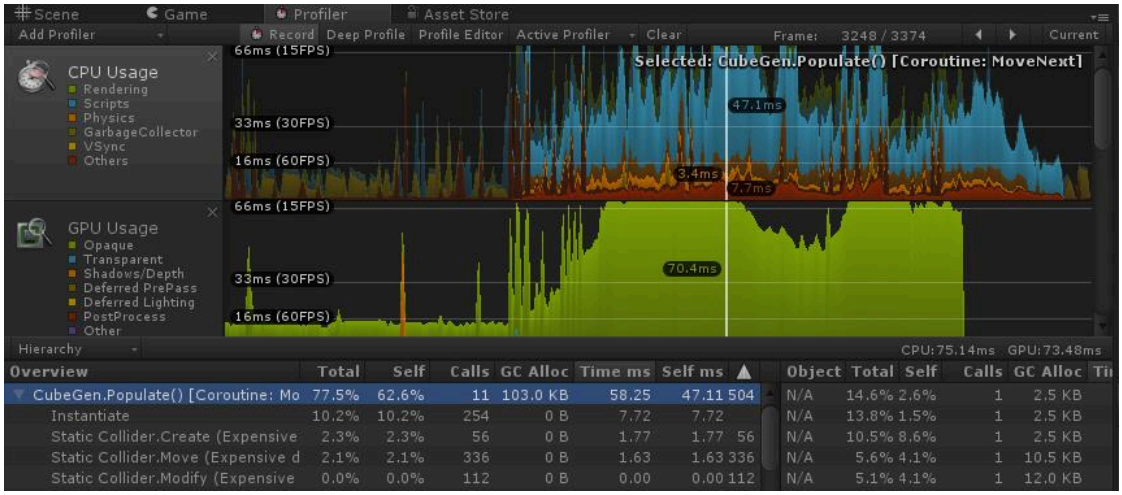
And the populee (Thing that is populated):

- A range for each of the major parameters (Height, moisture, and temperature) that they can spawn within.
- The chance that they can spawn
- A falloff value for the ranges, so they don't have a sudden cut off
- 1-n prefabs that could be spawned, to be selected from randomly

12.3.2 Implementation:

The SuperPopulator class was created. This worked in a manner similar to GrassPopulator, however, it contained a list of Populees. Once a point on the surface is calculated, this list is iterated through and the Check method is called for each populee, with the given position. If it all the requirements passed for a mesh to be placed, it will return a mesh at random to be instantiated.

12.3.3 Testing:



The performance hit from the asset placement was severe – dropping the framerate below acceptable levels.

13 FINAL TESTING

1. The planets need to be such quality that they are reasonably believable.
2. The generation needs to work in real-time. It is not acceptable for the game to pause and make the player wait as the planet generates a higher quality mesh for the cliff they are approaching
3. The framework needs to be easy to use. It should be able to be integrated into a game without too much manual work required, and have easy to understand parameters & settings.
4. The framework needs to be easily expandable, should the users want to change how any part of it works, or add more features, it should be constructed in a way that makes this easy to do.

13.1 User Testing:

Whilst some of the testing can be done objectively, some of the criteria are subjective. Because of this, testing was done on users to gather their opinions. Here are the questions asked:

Please provide details on your machine specifications:

If possible, run the program at 1920x1080, fullscreen, Fantastic quality. If not, please provide the settings you are running it at:

Please answer the following questions on a scale of 1 to 10, where 1 lowest, and 10 is highest.

How well did you feel the program ran?

In a space exploration game, how well do you feel these planets would be suited?

How severe was the jitter in framerate, if there was any?

Did you experience any framerate drops? If so, how much of an issue do you feel these were?

Numeric answer questions:

What was the average framerate viewing the planet from the default distance?

What was the average framerate approaching the planet?

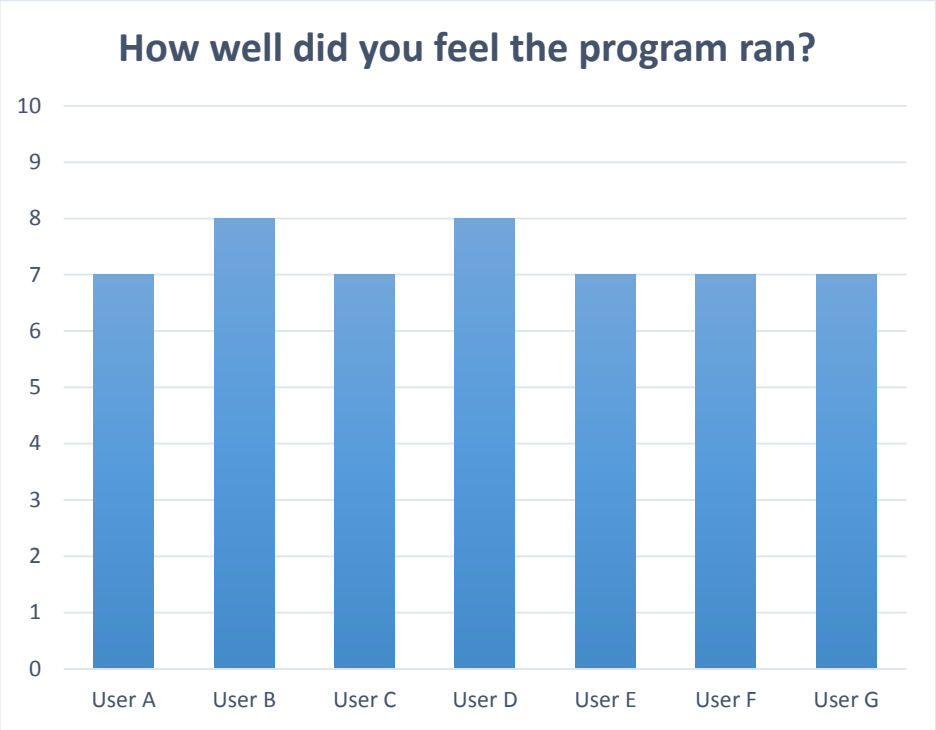
What was the average framerate viewing the planet from near the surface?

Detailed feedback questions:

What issues, if any did you have?

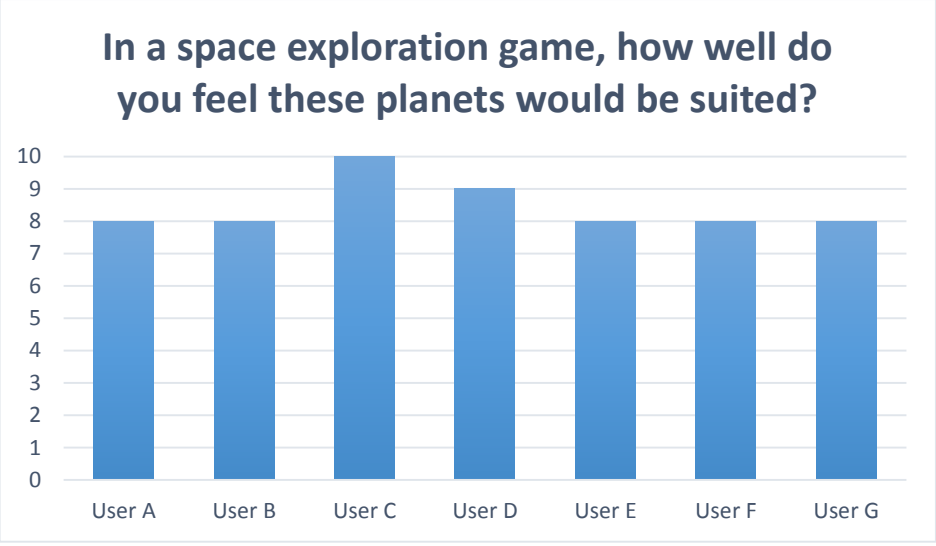
If you had to pick one thing to be improved, what would it be?

13.1.1 Testing results summarized:



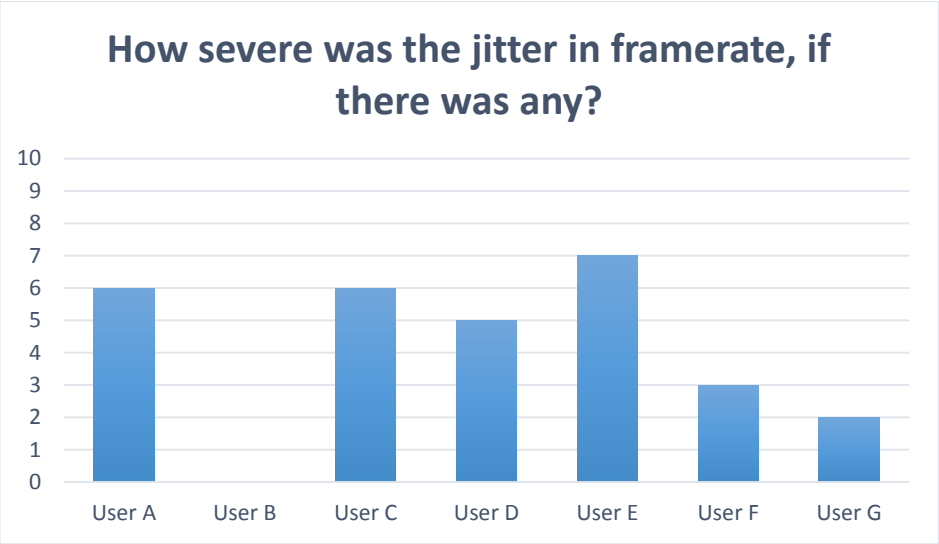
Mean: 7.3

Nobody felt the performance was unacceptable, although there were no perfect scores. There is clearly more room for improvement here, but the current performance is acceptable.



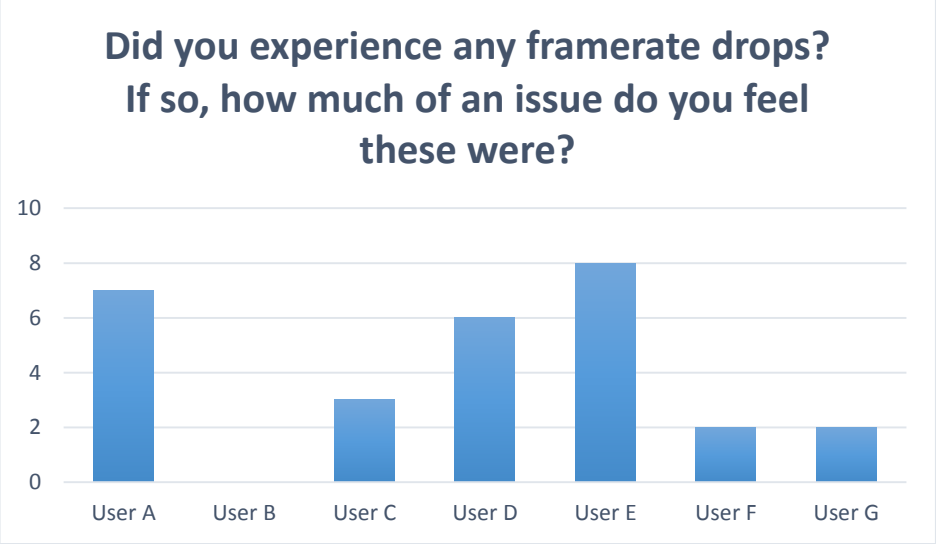
Mean: 8.4

The all positive scores here seem to be encouraging. There is still room for improvement, but the planets seem to be acceptable.



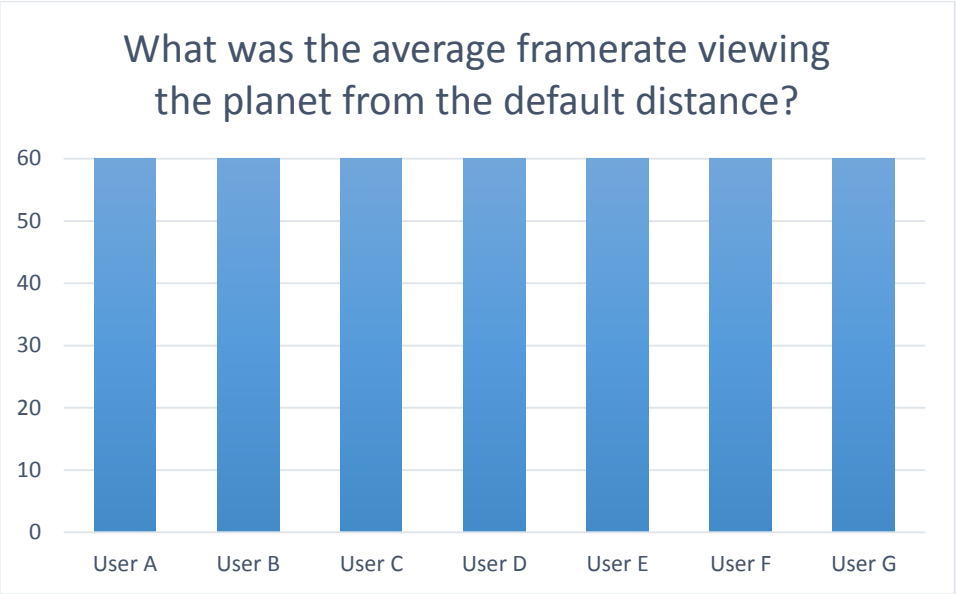
Mean: 4.1

The jitter is a clear issue for some users. It appears this is more of an issue on lower spec machines, which is likely due to the configuration of number of threads & texture size.

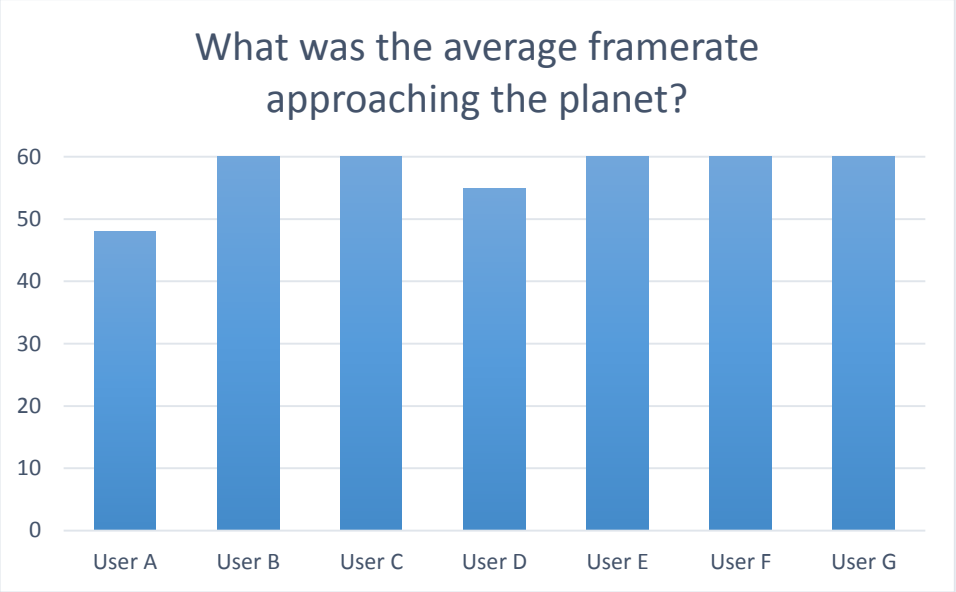


Mean: 4

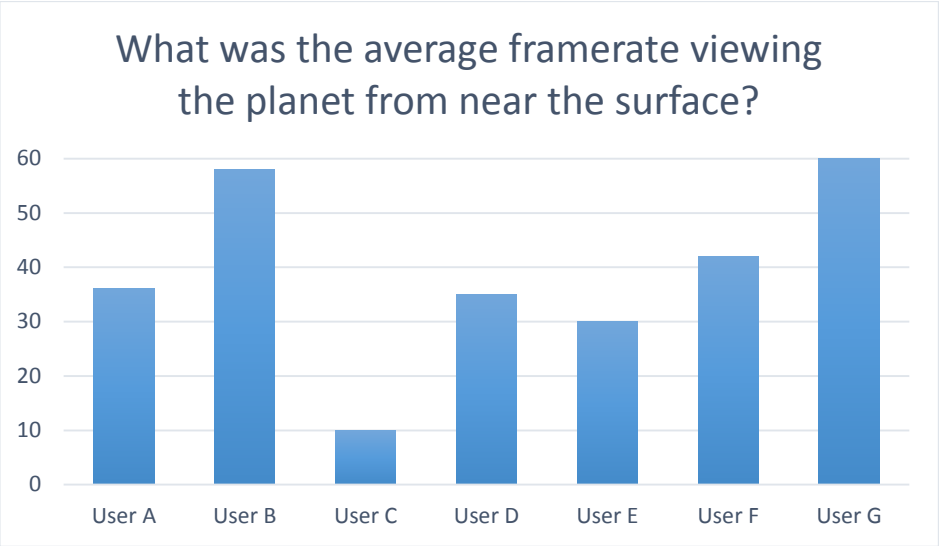
Another varied result. Users with stronger hardware had less of an issue with framerate drops, where ones with weaker hardware had more of an issue.



Mean: 60



Mean: 58



Mean: 38

From these FPS counts, it is clear there are significant performance issues at surface level.

What issues, if any did you have?

- Perfectly gridded foliage object spawns and no skybox around planet when looking up from the surface.
- It ran smoothly, but the surface took a long time to render fully

If you had to pick one thing to be improved, what would it be?

- gridded object spawns
- Render speed
- The textures on the surface from a distance

14 EVALUATION

14.1 Evaluation of user feedback

From the user feedback, summarized in 12.1.1, it is clear there are significant performance issues at surface level. This is due to the grass & surface detail generation, which needs to either be disabled on weaker hardware, or improved. However, the realism of the planets and the performance when the grass & tree placement is disabled is excellent.

14.2 Why it's not a plugin yet

The planet generation framework is not ready to be dropped into a game yet. There are various reasons for this. Creating a real-time planet generator turned out to be more work than expected.

I spent a large amount of my time working on the visuals of the project, and this may have been better spent on creating the setup to make it an easy to use plugin / framework. However, I felt my time was better spent improving the actual generation system instead of making it easy to drop into a game, as there would not be much demand for an unimpressive planet generator.

There are various technical issues involved with making a game support planets of this size, with surface level detail. Unity represents all positions using Vector3, a data structure consisting of 3 floats, each which has 32-bits of precision. The msdn documentation says these have reliable precision for 7 digits (Microsoft, 2013)

Whilst this is normally enough precision for anything in an average game, in a space game, the scales are so immense this can cause problems. Take our typical solar system as an example.

Assume our sun is at (0,0,0) and earth is ~150,000,000,000meters away. If we stuck with the standard unity scale of 1 unit = 1 meter, this could put our planet somewhere near (150000000000,0,0)

When stored as a float, due to the limited precision, we end up with our planet at 149999993000. Whilst this inaccurate is the location of the planet is not too much of an issue, the real issue is any positions near this value will also be imprecise. At this range, any point within about 5km end up as the same position

This issue can be solved, but requires some fairly substantial changes to either the underlying engine, or how things are positioned in the game. Since

Unity3D is a closed source engine, we are unable to modify something as low level as the datatypes used for positioning things. Because of this, this issue would have to be fixed a different way.

How can this be fixed?

14.3 Floating Origin

> *"I understand how the engines work now. It came to me in a dream. The engines don't move the ship at all. The ship stays where it is, and the engines move the universe around it." - Cubert J. Farnsworth (Futurama, 2000)*

In the Unite 2013 conference, the developers of Kerbal Space Program talked through how they overcame these issues (Falaghe ,2013). A method known as ‘Floating Origin’ is used. Instead of moving the camera towards far away objects, we essentially keep the camera where it is, and bring distance objects closer to it. This allows the high precision float values to always be kept in the space closer to the camera, thus resolving the spatial jittering caused by the lack of floating point precision.

14.4 Future improvements

Planet generation is a problem that could be considered as fairly open ended – These are so many different factors that it is an extremely large amount of work to create an all-encompassing system.

There are many things that still need doing, and could be done; there are issues that need to be fixed, optimizations that could be made, places the quality could be improved, extra-planet features to be considered, intra-planet features to be added, and more.

14.4.1 Optimizations

14.4.1.1 Performance increases

The generation is current very CPU limited. The GPU is barely utilized for the generation,

To increase the performance, OpenCL/Compute shaders / CUDA etc could be used.

14.4.1.2 Use of idle cpu time

In its current state, when you get within a set distance of a mesh, it will start generating the new mesh and texture, and as soon as it is done, load it in. This means there is a lot of time when you are approaching a mesh, and the cpu is idle until you get closer. The generation could be improved to generate the highest probability next N amount of meshes & textures whilst it is idle (at a low priority), and then load these in when needed. This should result in faster generation when it is needed, but may also waste CPU cycles if the player changes their direction.

14.4.2 Improvements

14.4.2.1 Separating mesh change and texture change

At the moment, the CubeGen waits until both the mesh and texture are generated before it swaps out the current mesh for the new one, with the new texture. The mesh generation is a less intensive process, and will generally finish first. The texture required for this mesh would not be generated yet, but the texture of the parent would include texture needed in the respective quarter of it, at the same quality it was previously. The shader could be modified to use this already generated texture, and use the respective quarter of it to render the new, higher quality mesh, with the old, same quality texture, until the new texture is done generating.

14.4.2.2 Planetary features

One of the most recognisable features of the planet earth from space is the cloud coverage. In addition, when on the surface of earth, the cloud coverage of the sky is very important to realism. Procedural clouds are a complicated topic, and there any many ways this could be solved. Generating cloud textures using perlin noise is a possible option, using a similar method as the planet's surface to ensure quality. However, these would become obvious they had no depth to them when you approached. Several different layers could be created, but this may have a large performance hit, and may still not look convincing. Real-time volumetric clouds would look good, but be a very heavy performance hit, and a substantial amount of work.

Along with clouds, simulation of the atmosphere would help improve the lack of realistic lighting and shading. Atmospheric scatting shaders would vastly improve the visuals of the planet, and make it much more appealing.

14.4.3 Intra-planetary features

The terrain generation could be improved. Using a combination of different noise functions, selectors and blenders, a much more realistic planet could be simulated, with different biomes, mountains, plains, deserts, rivers etc. However, the more complicated these are made, the slower the generation will be. If a generator this complex is required, a lot more optimizations may have to be made. However, the user testing results seemed to imply that the planets were acceptable in their current state. However, there is always room for improvement.

14.4.4 Extra-planetary features

Moons and rings around planets could also be generated. These would improve the visual variety of the planets somewhat. Generating moons would just be a case of generating another planet, with a slightly different set of properties. To generate rings, a textured plane, or ring mesh could be procedurally generated. This may have to be split up into separate parts to ensure correct draw sorting.

In its current state, the system focuses mostly on generating singular planets, without any consideration of their place in the solar system. A system for generating solar systems could be designed, starting with generating a sun, and a number of planets orbiting it,

Once solar systems are implemented, an entire galaxy could be created made of up solar systems. There would have to be a lot of consideration into float precision and scaling issues to achieve this however.

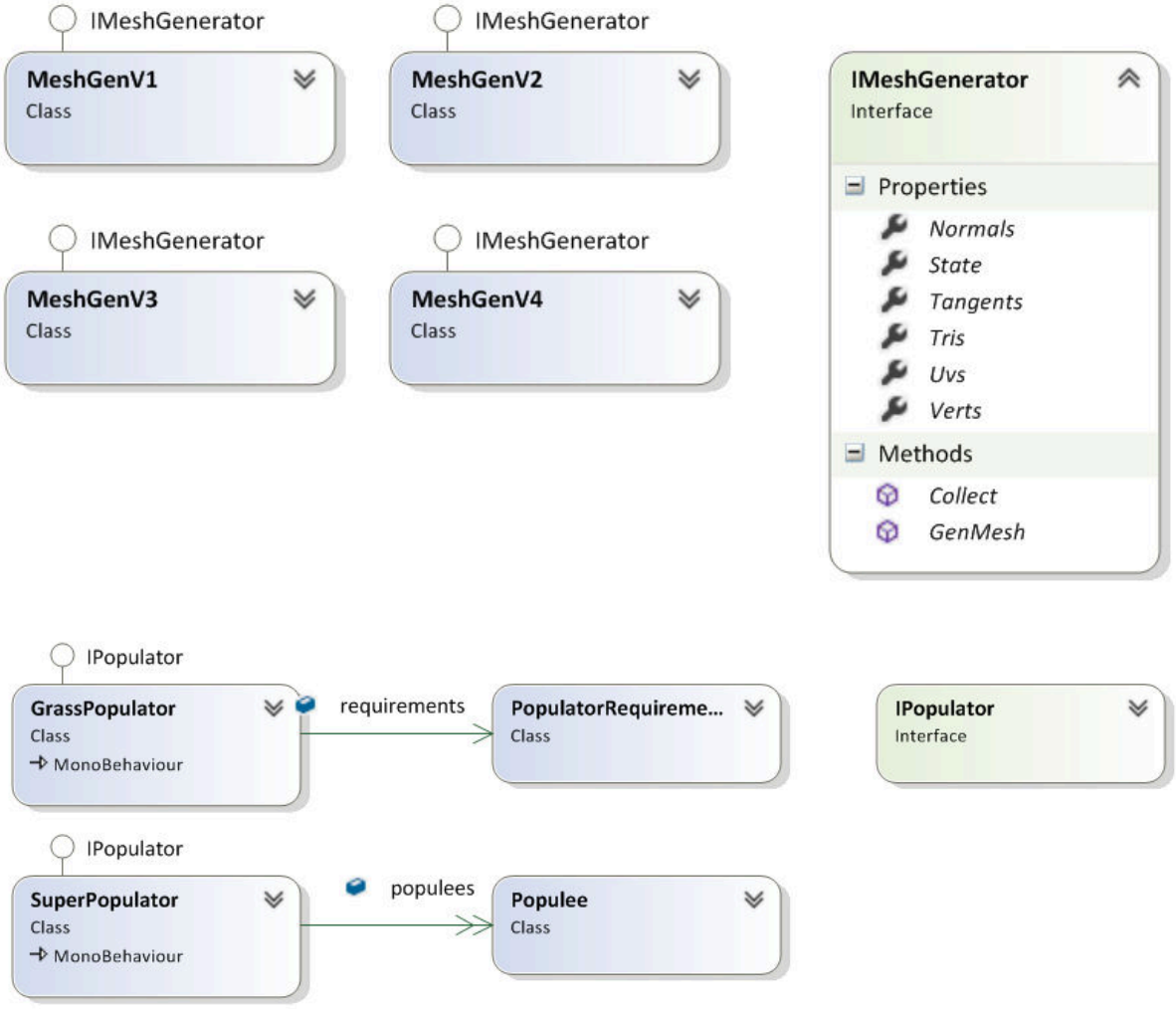
14.4.5 Planet types

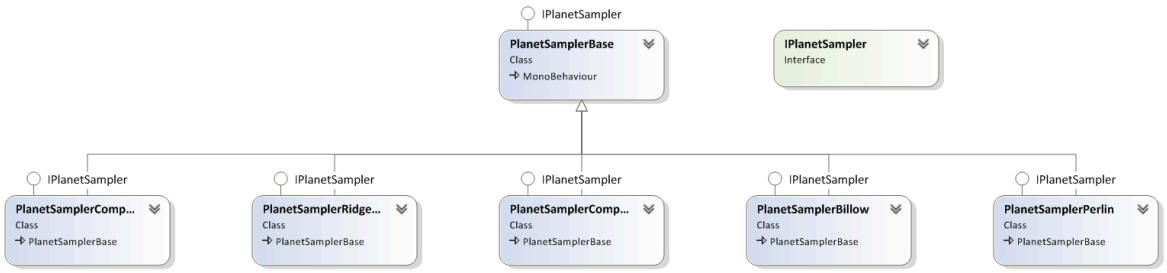
A larger variety of planet types could be created. Now, most of the planetary generation was focused on earth-like planets. However, gas giants, suns, moons, and asteroids could all be simulated, as well as other types of planets. This would mostly need just different types of shaders and samples created.

14.4.6 Expandability

Each part of the generation is based off an interface, and can be swapped out, expanded, or replaced.

Here are the various interfaces, and use of them:




IPlanetSampler
PlanetSamplerBase
Class
MonoBehaviour
IPlanetSampler
Interface
IPlanetSampler
PlanetSamplerComp...
Class
PlanetSamplerBase
IPlanetSampler
PlanetSamplerRidge...
Class
PlanetSamplerBase
IPlanetSampler
PlanetSamplerComp...
Class
PlanetSamplerBase
IPlanetSampler
PlanetSamplerBillow
Class
PlanetSamplerBase
IPlanetSampler
PlanetSamplerPerlin
Class
PlanetSamplerBase

15 CONCLUSIONS

This project was a large undertaking, and due to time restrictions, did not meet all of the projects goals.

The projects goals were:

1. The planets need to be such quality that they are reasonably believable.
2. The generation needs to work in real-time. It is not acceptable for the game to pause and make the player wait as the planet generates a higher quality mesh for the cliff they are approaching.
3. The framework needs to be easy to use. It should be able to be integrated into a game without significant manual work required, and have easy to understand parameters & settings.
4. The framework needs to be easily expandable, should the user want to change how any part of it works, or add more features, it should be constructed in a way that makes this easy to do.

The developed solution meets goals one, two and four. However, due to a lack of time, and several complications (See 13.2), goal three was not met.

The basis of an expandable, real-time procedural planet generation framework was created, but more work is required for it to be ready to be distributed as a Unity plugin.

16 RECOMMENDATIONS

Due to time constraints, and other limitations, there are a number of features that were not completed. In this chapter, we briefly look at some of these and consider improvements.

16.1 Finishing creating an easy-to-implement plugin

One of the main goals not met was creating an easy to implement plugin. To meet this goal, there are still some significant issues that need solving. The floating origin system (13.3) would have to be implemented, with helper components to be dropped onto the user's ships and cameras.

The render settings and generator settings would need to be tidied up into some simpler components, and an easy to use editor for the generators would need to be created, with obvious variable names that do not require you to know what variables like 'frequency' and 'lacunarity' affect. These could be abstracted to more obvious parameters like "continent size", "sea level", etc.

16.2 Improve performance of the surface details

One of the major issues from user feedback was the performance of the surface details (grass, trees, rocks, etc) on slower machines. To resolve these issues, methods to optimize their performance will need to be explored.

These issues may be fixable by combining multiple meshes together to reduce draw calls and the amount of instantiations, or it may be that they have to be reduced / disabled on weaker hardware. Better use of LOD models on the trees where they are reduced to billboards at distance may also help performance.

17 REFERENCES

Górski, K. M. (2005) 'HEALPix: A Framework for High-Resolution Discretization and Fast Analysis of Data Distributed on the Sphere', The Astrophysical Journal, Volume 622, Issue 2, pp. 759-771. doi:10.1086/427976.

Maxis / Electronic Arts. (2007) 'Creating Spherical Worlds', SIGGRAPH '07, ACM SIGGRAPH 2007 sketches, Article No.82 . doi: 10.1145/1278780.1278879 (http://www.cs.cmu.edu/~ajw/s2007/0251-SphericalWorlds.pdf)

Hamilton, M. (2013) 'Quadrilateralized Spherical Cubes (Quadspheres) in Eve Flight Sim for Rendering Planets', Nerd H*ck - Engineering blog of Mark Hamilton, 21/07 Available at: http://www.markhamilton.info/2013/07/quadrilateralized-spherical-cubes-cubespheres-in-eve-flight-sim-for-rendering-planets/ (Accessed: 26/11/2013).

Proland Documentation - Core Library (2012) Available at: http://proland.inrialpes.fr/doc/proland-4.0/core/html/index.html (Accessed: 26/11/2013)

Microsoft (2013) *float (C# Reference)*. Available at: http://msdn.microsoft.com/en-us/library/b1e65aza(v=vs.120).aspx (Accessed: 20/03/2014).

jashan (2009) 'Boo, C# and JavaScript in Unity - Experiences and Opinions', *Unity3D Forums*, 02/27. Available at: http://forum.unity3d.com/threads/18507-Boo-C-and-JavaScript-in-Unity-Experiences-and-Opinions (Accessed: 26/03/2014).

Falaghe, F. (2013) *Unite 2013 - Building a new universe in Kerbal Space Program* [Video]. Available at: https://www.youtube.com/watch?v=mXTxQko-JH0 (Accessed: 26 March 2014).

NVIDIA (2007) *Chapter 7. Rendering Countless Blades of Waving Grass*. Available at: http://http.developer.nvidia.com/GPUGems/gpugems_ch07.html (Accessed: 20/03/2014).

Unity (2013) *Unity Script Reference: Graphics.DrawMesh*. Available at: http://docs.unity3d.com/Documentation/ScriptReference/Graphics.DrawMesh.html (Accessed: 20/03/2014).

GRAAF, Twan De. 2013. *Procedural planets into detail*. [online]. Available from World Wide Web: <http://www.twandegraaf.nl/Art/Documents/Procedural%20planets%20into%20detail,%20Twan%20de%20Graaf%202012.pdf>

NASA - What Kinds of Planets Are Out There? 2007. [online]. [Accessed 17 Oct 2013]. Available from World Wide Web: <http://www.nasa.gov/vision/universe/starsgalaxies/earthsized_planets.html>

'A Clone of My Own' (2000) *Futurama*, Series 2, Episode 15. Fox, April 9th.

BRUNETON, Eric. 2012. *Eric Bruneton*. [online]. [Accessed 17 Oct 2013]. Available from World Wide Web: <http://evasion.inrialpes.fr/~Eric.Bruneton/>
Continents on a Sphere. 2010. [online]. [Accessed 18 Oct 2013]. Available from World Wide Web: <http://www.dungeonleague.com/2010/02/15/continent-samples/>

18 APPENDIX

18.1 User feedback

18.1.1 User A

Please answer the following questions on a scale of 1 to 10, where 1 lowest, and 10 is highest.
How well did you feel the program ran? - 7

In a space exploration game, how well do you feel these planets would be suited? - 8

How severe was the framerate jitter, if there was any? - 6

Did you experience any framerate drops? yes, when rendering objects on the planet and at certain points on approach.

If so, how much of an issue do you feel these were? - 7

Numeric answer questions:

What was the average framerate viewing the planet from the default distance? - 60

What was the average framerate approaching the planet? - 48

What was the average framerate viewing the planet from near the surface? - 36

Detailed feedback questions:

What issues, if any did you have?

Perfectly gridded foliage object spawns and no skybox around planet when looking up from the surface.

If you had to pick one thing to be improved, what would it be?
gridded object spawns

18.1.2 User B

If possible, run the program at 1920x1080, fullscreen, Fantastic quality. If not, please provide the settings you are running it at:

Please provide details on your machine specifications:
CPU: i5 4670k
GPU: nvidia gtx 560
Please answer the following questions on a scale of 1 to 10, where 1 lowest, and 10 is highest.

How well did you feel the program ran?
8
In a space exploration game, how well do you feel these planets would be suited?
8
How severe was the jitter in framerate, if there was any?
0 (no jitter)
Did you experience any framerate drops?
0 no
If so, how much of an issue do you feel these were?
Numeric answer questions:
What was the average framerate viewing the planet from the default distance?

61
What was the average framerate approaching the planet?
61
What was the average framerate viewing the planet from near the surface?
58
Detailed feedback questions:
What issues, if any did you have?
It ran smoothly, but the surface took a longtime to render fully
If you had to pick one thing to be improved, what would it be?
Render speed

18.1.3 User C

How well did you feel the program ran?
7
In a space exploration game, how well do you feel these planets would be suited?
10
How severe was the jitter in framerate, if there was any?
6 Only bad near surface with plants and trees
Did you experience any framerate drops?
Lag when creating a plnaet, and at planets surface with the trees and plants turned on. Without them, solid framerate.
If so, how much of an issue do you feel these were?
3
Numeric answer questions:
What was the average framerate viewing the planet from the default distance?
60
What was the average framerate approaching the planet?
60
What was the average framerate viewing the planet from near the surface?
10, 60 with grass and trees turned off

Detailed feedback questions:
What issues, if any did you have?

If you had to pick one thing to be improved, what would it be?
The textures on the surface from a distance

18.1.4 User D

Please answer the following questions on a scale of 1 to 10, where 1 lowest, and 10 is highest.

How well did you feel the program ran? 8
In a space exploration game, how well do you feel these planets would be suited? 9
How severe was the jitter in framerate, if there was any? There is no any jitter when you are far away from surface but when I get close to surface there is a lot of jitter
Did you experience any framerate drops? Yes when I'm really close at surface
if so, how much of an issue do you feel these were? 6
Numeric answer questions:
What was the average framerate viewing the planet from the default distance? 60
What was the average framerate approaching the planet? 50-60
What was the average framerate viewing the planet from near the surface? 20-50

Detailed feedback questions:
What issues, if any did you have?
- There a bug that when you closed the game it not actually closed the exe still working in task manager and I can't even close it.

If you had to pick one thing to be improved, what would it be?

- You should have exit menu.
- Sprint (Increase the ship speed).

18.1.5 User E

machine specifications: 8gb RAM, 1gb Video Card, 3.8ghz Quad Core processor, 1TB HDD
Program at 1920x1080, fullscreen, Fantastic quality

Please answer the following questions on a scale of 1 to 10, where 1 lowest, and 10 is highest.

How well did you feel the program ran? 7/10
In a space exploration game, how well do you feel these planets would be suited? 8/10 Needs more biomes
How severe was the jitter in framerate, if there was any? 7/10
Did you experience any framerate drops? Yes
If so, how much of an issue do you feel these were? Nearly constant issue, 8
Numeric answer questions:
What was the average framerate viewing the planet from the default distance? 60
What was the average framerate approaching the planet? 60
What was the average framerate viewing the planet from near the surface? 30

Detailed feedback questions:
What issues, if any did you have?
No in-game option to exit the game.

18.1.6 User F

Please answer the following questions on a scale of 1 to 10, where 1 lowest, and 10 is highest.

How well did you feel the program ran? 7
In a space exploration game, how well do you feel these planets would be suited? 8
How severe was the jitter in framerate, if there was any? 3
Did you experience any framerate drops?
If so, how much of an issue do you feel these were? 2
Numeric answer questions:
What was the average framerate viewing the planet from the default distance? 60
What was the average framerate approaching the planet? 60
What was the average framerate viewing the planet from near the surface? 42

Detailed feedback questions:
What issues, if any did you have?
Pop in effect is jarring

If you had to pick one thing to be improved, what would it be?
Pop-in

18.1.7 User G

CPU: Intel(R) Core(TM) i7-3930K CPU @ 3.20GHz (12 CPUs), ~3.2GHz
GPU: NVIDIA GeForce GTX 670
RAM: 16GB
How well did you feel the program ran? 7
In a space exploration game, how well do you feel these planets would be suited? 8
How severe was the jitter in framerate, if there was any? 2
Did you experience any framerate drops? 2
If so, how much of an issue do you feel these were? 2
Numeric answer questions:
What was the average framerate viewing the planet from the default distance? 61
What was the average framerate approaching the planet? 61
What was the average framerate viewing the planet from near the surface? 60

Detailed feedback questions:
What issues, if any did you have?
Not really

If you had to pick one thing to be improved, what would it be?
Interface menu, needs exit button
Render distance on trees and grass

18.2 User feedback summarized

How well did you feel the program ran?	User A	User B	User C	User D	User E	User F	User G
	7	8	7	8	7	7	7

Average:	7.285714						
In a space exploration game, how well do you feel these planets would be suited?	User A	User B	User C	User D	User E	User F	User G
	8	8	10	9	8	8	8
Average:	8.428571						
How severe was the jitter in framerate, if there was any?	User A	User B	User C	User D	User E	User F	User G
	6	0	6	5	7	3	2
Average:	4.142857						
Did you experience any framerate drops? If so, how much of an issue do you feel these were?	User A	User B	User C	User D	User E	User F	User G
	7	0	3	6	8	2	2
Average:	4						
What was the average framerate viewing the planet from the default distance?	User A	User B	User C	User D	User E	User F	User G
	60	61	60	60	60	60	60
Average:	60.14286						
What was the average framerate approaching the planet?	User A	User B	User C	User D	User E	User F	User G
	48	61	60	55	60	60	60
Average:	57.71429						
What was the average framerate viewing the planet from near the surface?	User A	User B	User C	User D	User E	User F	User G
	36	58	10	35	30	42	60
Average:	38.71429						
What issues, if any did you have?	User A	User B	User C	User D	User E	User F	User G
	Perfectly gridded foliage object spawns and no skybox around planet when looking up from the surface.	It ran smoothly, but the surface took a longtime to render fully					
If you had to pick one thing to be improved, what would it be?							
	gridded object	Render speed	The textures on the surface from a distance				

18.3 Interface code samples

18.3.1 IPlanetSampler.cs

```
using UnityEngine;


/// <summary>
/// Planetary point sampler
/// Returns the height, temperature and moisture of a given point of a planet
/// </summary>
public interface IPlanetSampler
{
    /// <summary>
    /// Size of the planet
    /// </summary>
    float Size { get; set; }

    /// <summary>
    /// Height variance of the surface of the planet (Size vary + / - this
value)
    /// </summary>
    float SurfaceHeight { get; set; }

    /// <summary>
    /// Seed value - For a given position, a generator should always return the
same value for a given seed!
    /// </summary>
    int Seed { get; set; }

    /// <summary>
    /// Sampler should initialize anything expensive here
    /// </summary>
    void Awake();

    /// <summary>
    /// Set any internal variables to random values!
    /// </summary>
       void Randomize();

    /// <summary>
    /// Returns the height of the planet at a given surface point
    /// </summary>
    /// <param name="pos">Position on the surface of the planet (this should
always be unit length!)</param>
    /// <returns>Height offset (-1 to 1 range)</returns>
    float GetHeight(Vector3 pos);


    /// <summary>
    /// Returns the temperate of the planet at a given surface point
    /// </summary>
    /// <param name="pos">Position on the surface of the planet (this should
always be unit length!)</param>
    /// <returns>Temperature</returns>
    float GetTemperature(Vector3 pos);
```

```
    /// <summary>
    /// Returns the moisture of the planet at a given surface point
    /// </summary>
    /// <param name="pos">Position on the surface of the planet (this should
always be unit length!)</param>
    /// <returns>Moisture</returns>
    float GetMoisture(Vector3 pos);
}
```

18.3.2 IMeshGenerator.cs

```
using UnityEngine;

/// <summary>
/// Interface for the planetary mesh generator
/// Generates verts, normals, uvs and tangent data for the mesh
/// </summary>
public interface IMeshGenerator
{
    /// <summary>
    /// The current state of the mesh generator
    /// </summary>
    GenState State { get; set; }

    int[] Tris { get; }
    Vector3[] Verts { get; }
    Vector3[] Normals { get; }
    Vector2[] Uvs { get; }
    Vector4[] Tangents { get; }

    /// <summary>
    /// Called when the main thread has taken the data from the various arrays,
and it is safe for generating a new mesh
    /// </summary>
    void Collect();

    /// <summary>
    /// Generates the mesh for a given CubeGen
    /// </summary>
    /// <param name="gen">The cubegen component to generate the mesh for</param>
    /// <param name="sampler">The planetary sampler</param>
    void GenMesh(CubeGen gen, IPlanetSampler sampler);
}
```