Procedurally Generated Planets

Matthew Lowe

15 April 2019

A project report submitted in partial fulfilment for the degree of
Master of Computing in Computer Games Development

School of Physical Sciences and Computing
University of Central Lancashire

Abstract

Creating and rendering near real size planets is no easy task. A robust and well-designed solution must be implemented. The solution must be able to render planets at a distance and close up, whilst taking up no more than the average memory available on a standard computer. Additionally, the terrain should be aesthetically pleasing and include the use of biomes, water, clouds and an atmosphere. If an engine with these features exist, it could be used as the backbone for many games.

This project aims to achieve a solution to rendering very large planets, including some aspect of realism to the terrain and other phenomena like atmospheres and water. It would then be well on the way to creating a generic, procedural planet renderer. Furthermore, this report investigates methods to assemble the planets into a star system. This future work could support large scale distance, and smaller scale distance travelling.

To solve the problem, rigorous research was carried out beforehand. Then agile development techniques similar to DSDM were applied - with source control to aid, to rapidly iterate features. Due to the timeboxed nature of the methodology, features were put on hold if they took too long. This made way for other features which may be faster to complete.

The solution in this paper achieves most of the initial objectives. The engine can render very large planets, with atmospheres, water, clouds and biomes. The engine can be compiled stand alone, for use in other projects which supports the idea of a generic planetary renderer. The engine didn’t achieve full scale star systems, but the sufficient research has been done in order to potentially implement this work in the future.

Attestation

I understand the nature of plagiarism, and I am aware of the University's policy on this.
I certify that this document reports original work by me during my University project.

Signature *Matthew Lowe* **Date** *12/04/2019*

Acknowledgements

I would like to thank my supervisor Gareth Bellaby, for all the great feedback and expertise which allowed me to complete this project.

Also, I would like to thank Laurent Noel, who gave me some excellent initial pointers before I started the project, so I could get started on the research.

Finally, I would like to acknowledge Neil Osbaldeston for coincidently doing the same project as me, as the competitive aspect helped kept me motivated and pushed my abilities.

Table of Contents

List of Figures

List of Tables

List of Listings

1 Introduction

1.1 Background and Context

Procedurally generated planets are, in the context of this project, large scale and realistically generated planets. These planets are near real size, like you would find in our own solar system. Procedural planets could have a multitude of uses in games development. Whether it's used for an exploration game, or just a visual artefact displayed in the sky, it adds considerable value to the user experience. Since procedural planets have been popping up in game in recent years, my aim is to create a general engine which can be used in many scenarios in games. The engine can render planets very far away and close up to suit these different scenarios.

1.2 Scope and Objectives

The main goal is to create an engine which can generate and render procedurally generated planets. These planets can be Earth scale and be viewed from space or on the ground. Other objectives include being able to render real phenomena like clouds, water and an atmosphere. In addition to planets being randomly generated, the seed can be saved as to load in the same generated planet. Furthermore, an extra feature would be to render multiple planets and moons which can orbit each other, as well as orbiting a star.

1.3 Achievements

Most of the goals were achieved in this project. The planetary rendering engine renders different levels of detail in order for the planets to be viewed up close or from a distance. The engine also supports water, clouds and atmospheres. The engine does have support for rendering multiple planets orbiting each other, however attaching a camera gives visually displeasing effects due to floating point errors. This makes it only viable to use in its current state if you were to observe a planet move from a distance and not follow it.

1.4 Overview of Report

Chapter 2 investigates literature surrounding the topics required to be able to implement the project.

Chapter 3 explains the planning and preparation process underwent, plus potential legal and ethical issues.

Chapter 4 describes how the system was designed and relations between classes in order to make implementation easy to carry out.

Chapter 5 explains how each specific part of the project was implemented with each section describing the steps took to complete that part.

Chapter 6 shows the testing strategy used to ensure that all parts of the engine work together and each individual part does its job correctly.

Chapter 7 Goes into depth about how the project was executed, reflecting on what went well and what didn’t, it also describes potential future work.

2 Literature Review

2.1 Introduction

In this chapter is a comparison of the literature on methods associated with level of detail geometry, procedural terrain generation and rendering large scale planets.

2.2 Level of Detail Methods

2.2.1 Continuous LOD Using Quadtrees (CLOD)

A quadtree is a tree data structure where each node may be empty or contain four child nodes, first named by Raphael Finkel and J.L. Bentley in 1974. They are flattened versions of octrees, where an octree consists of eight child nodes. Quadtrees are particularly useful in terrain generation due to their ability to split and merge according to the data it contains. If terrain geometry is stored in the nodes, it allows a natural progression towards creating a terrain LOD system.

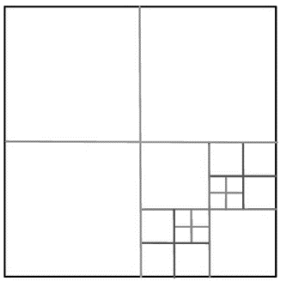


Figure 1 - Visualisation of a Quadtree

The CLOD method described by Ulrich in 2000, uses a 3x3 grid of vertices stored in each quadtree node. The vertices on each edge midpoint store an additional value of whether it's enabled or not (see Figure 2). Using triangle fans when rendering allows skipping of these disabled vertices easily. Vertices are flagged as disabled when a neighbouring quadtree node's edge has a lower LOD, this prevents cracks in the terrain due to LOD differences.

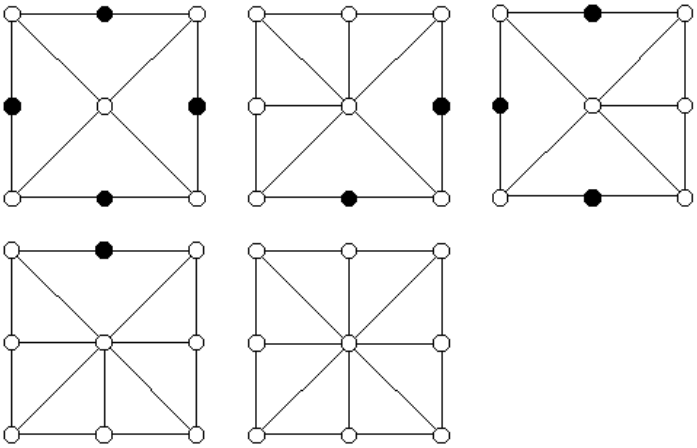


Figure 2 - 3x3 grid of vertices with disabled vertices in black

A similar approach by Jian Wu, 2010 uses a 5x5 grid of vertices per node and describes two methods to eliminate cracks. The first method deletes a vertex if an adjacent quad is of a lower LOD when rendering. However, this method only works if the LOD difference is exactly one. The second method, original to the paper, loops through each quad in the grid and detects if the adjacent quad's LOD is greater than one. If so, then add it to a list of cracks. Finally, loop through this list and generate triangles to fill in the cracks. An optimisation of the first method is the pre-generation of these deleted vertices (Andersson, 2009). There is a total of nine permutations for LOD differences of one (Figure 3). Therefore, instead of deleting vertices, the index buffer is switched to one of these permutations once the combination of neighbouring LOD levels is calculated.

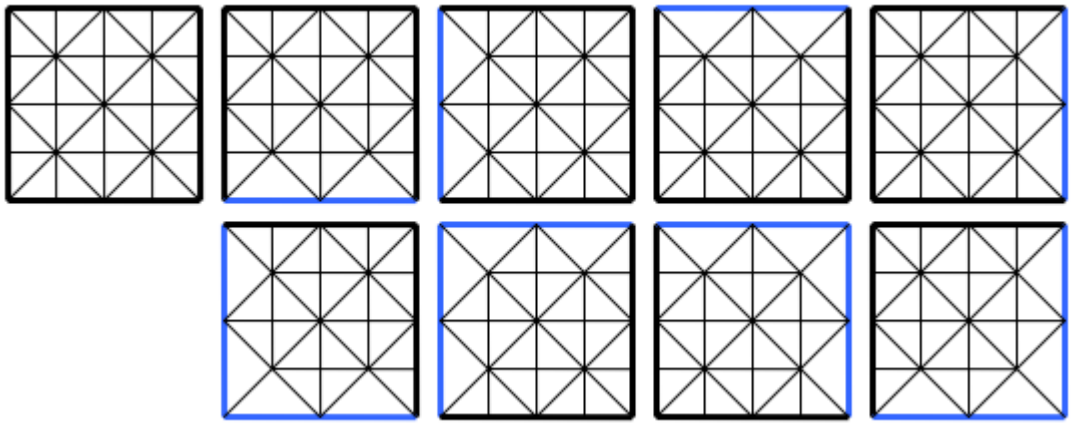


Figure 3 - The 9 permutations of the geometry due to neighbouring LODs

2.2.2 Geoclipmaps

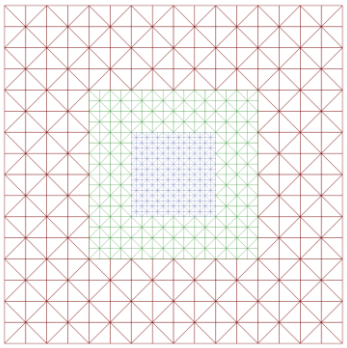


Figure 4 - A set of clipmaps

Geometry clipmaps were first implemented by Frank Losasso in 2004. The idea is synonymous to texture mipmapping, except it generates mipmaps of the geometry at many resolutions. Additionally, the method exercises the idea of large, complex terrain which is where clipmaps come into play. A clipmap is a section of a mipmap. If you were to generate mipmaps of complex terrain geometry, the data may still be too large, so the mipmaps are only generated around the camera position, hence the idea of clipmaps. Figure 4 shows an example of a set of Geoclipmaps.

Shortly after the first implementation of geometry clipmaps, a new GPU based method was described (Arul Asirvatham, 2005). It aimed to reduce the CPU computation time and utilise the GPU further to increase overall performance. It was realised that the terrain geometry can be represented as a set of images, allowing the work to be done on the graphics card.

Frank Losasso describes that additional work is needed to solve the cracks between the LOD levels. To solve this, they provide a morph formula which is applied to geometry near the outer boundary of each region.

2.2.3 Geomorphing

Using any LOD method, the geometry is subject to ‘popping’. This is when a transition occurs due to a change in the LOD, so vertices may snap to a slightly different position (Wikipedia, 2019). Geomorphing isn’t a LOD technique, just a method to stop this popping effect.

To morph the geometry, one extra piece of data needs to be stored, a target position. Whenever the LOD changes, the current position is the target position of the previous LOD, and the target position is now the next LOD’s final position. As the camera position changes, linear interpolation is performed between these two

positions. This allows the vertices to smoothly move into position. Geomorphing is typically carried out using shader techniques, due to performance increase (Wagner, 2019).

2.3 Procedural Terrain Generation

2.3.1 Perlin Noise

Figure 5 - An example of 2D Perlin noise

Perlin noise is a type of coherent noise used for many techniques within computer graphics, but generally to increase realism (Perlin, 1983). Some of the applications for this include generating procedural textures, procedural terrain, clouds and fire. In the context of procedural terrain, a huge drawing point of this algorithm is that it can be generated the same consistently without storing any terrain geometry except for an initial seed value. The obvious trade-off is that more real-time computation will be needed.

Eighteen years after Ken Perlin's original Perlin Noise paper he published another similar algorithm called Simplex Noise (Perlin, 2002). This newer algorithm has less computational complexity and fewer directional artefacts. Due to a patent on Simplex noise, there was a similar algorithm invented called OpenSimplex noise. OpenSimplex uses a slightly different underlying algorithm which produces a smoother surface but is slightly slower then Simplex noise.

All these noise algorithms can be expanded to higher dimensions. For example, consider a 2-dimensional simplex noise generator, this can produce a static procedural texture. But now it's possible to produce a cinematic moving effect by expanding it to use 3 dimensions, where the 'z' value is affected by time. The same could be done in 4 dimensions, e.g. producing a 3-dimensional effect but scrolling the 4th dimension through time to make it animated.

Typically, terrain generation methods include using a 2-dimensional noise generator because you only need a height value and an x and y coordinate to produce 3-dimensional terrain. However, if generating spherical terrain, you can use 3-

dimensional noise sampled at the surface of a sphere to produce the geometry with no seams. If this was done with conventional 2-dimensional noise, then there needs to be a method implemented to seamlessly wrap the terrain around the sphere. There are possible projections to use which could solve this, but regardless there will always some form of artefacts or distortion in the geometry.

2.3.2 Diamond Square Algorithm

The Diamond Square Algorithm is a method of terrain generation and a variation of the 3-dimensional version of the Midpoint Displacement Algorithm. Both algorithms begin with a grid of vertices of size 2^n+1, where n is the dimension of the terrain. The Midpoint Displacement Algorithm will take the two or four vertices depending on the dimension, average them and place them in the middle of the averaged points, but then offset this new point by a small amount. These steps are then repeated, but each time the offset scale is reduced. Though, this method produces square artefacts which is what the Diamond Square Algorithm improves upon (Archer, 2011). The Diamond Square Algorithm splits the step into two: the diamond step and the square step. The diamond step uses the corners of the square to produce and offset one point in the centre. The square step uses the corners of the square to produce and offset four new points in the centre of each square edge.

There are still a couple of limitations of this algorithm. Firstly, it still produces artefacts due to calculations are all based on a square or rectangular grid. Secondly, it can be difficult to tile the terrain, although still possible. One method could be to split the terrain into a grid and apply the algorithm to each grid section, using the corners and edges of the adjacent grid's as seed values.

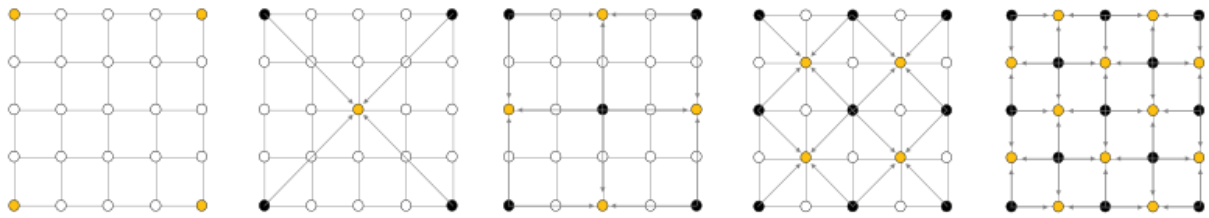


Figure 6 - The steps involved in each iteration of the algorithm

2.3.3 Biomes

The method to create biomes is to use a noise generator as discussed earlier. Instead of using the sample values as height values, they are used for moisture (Red Blob Games, 2015). Using combined elevation and moisture sampling, biomes can be deduced. For example, a grass biome may appear at a moisture value of 0.6 to 1.0, and an elevation value of 0.2 - 0.5, but mountains may appear and any elevation above 0.8. Creating a table of all these values allows a method to map out any biome over any moisture and elevation value. To use this, a biome lookup texture can be created, this texture is then sent to the shader. To sample the texture on the shader, the moisture and elevation can be sent as a 2-dimensional UV. This GPU based method has practically no overhead, except for the storing of one extra texture globally. This makes it very fast.

An alternative way to render biomes is to start off similar to the first method. Two noise maps are needed and can be moisture and elevation, or other similar ones. An example of a different two maps include temperature and rainfall which is what Minecraft uses (Gamepedia, 2019), see Figure 7. The major difference is that different noise parameters are used. Each biome will have its own unique settings. On the CPU, the type of biome will need to be determined in order to calculate the height. The hard part of this method is fixing the borders between biomes. If they aren't fixed there may be mountains bordering a flat grassland, producing a sheer drop which is totally unrealistic. One method to solve this uses Voronoi diagrams (Orr10c, 2018). The definition of this is to split a plane into different regions which are based on distance to points in the plane. These points can now represent biomes, so the distance from the points can be used as interpolation values between the biomes, creating a smooth transition.

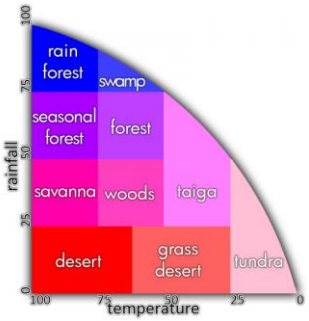


Figure 7 - Minecraft's biome map

2.4 Planetary Rendering

2.4.1 Atmosphere

Figure 8 - GPU Gems atmosphere example

To render a realistic planetary atmosphere like the Earth's you first need to understand why it happens in terms of physics. The Earth's atmosphere appears blue because blue light is scattered by particles within the atmosphere (Rayleigh scattering). There is also Mie scattering which only applies to larger particles within the atmosphere. This same effect can be calculated in real-time graphics applications. The general method described in GPU Gems 2 (O'Neil, 2006) to render an atmosphere is to start off with a sphere which is slightly bigger than the planet and culls the front facing triangles, this gives an outline to the planetary body. Next, implement volumetric rendering in the vertex shader by shooting a ray out from the camera at the atmosphere and sampling various points along the journey. Finally, send these accumulated values to the pixel shader where it works out the Mie and Rayleigh scattering colours.

2.4.2 Water Surface Rendering

The basis for any water surface rendering in games starts off with rendering a flat plane, or in the context of planets, a secondary sphere around the planet at water level. There are multiple methods to make this surface look like water with varying levels of complexity. The simplest is to colour the water blue with some alpha transparency then add a wave normal map on top. Scrolling the texture gives the impression of waves moving, however, this doesn't look too realistic but is good for a quick and simple implementation. A second method expands upon the first one, the difference is that multiple normal maps are used at different scales, then each map is scrolled at different velocities.

The first method will never look completely realistic. The more complex approach is to model the real effects which happen when looking at a water surface. The Fresnel effect contributes greatly to the appearance of water. When looking at shallow

angles, the light tends to bounce off the water, so you see a reflection of the sky. However, when viewed from above, the light ray goes into the water. This means you can sometimes see the bottom of the waterbed (N¨atterlund, 2008). Modelling the Fresnel effect, reflection and refraction can produce amazing and realistic looking water.

2.4.3 Level of Detail

In order to merge the idea of LOD terrain and planets, the problem of creating seamless spherical terrain needs to be solved. One way to do this is using a spherified cube (Pulido, 2013), where each face of the cube is a quadtree. This links in very nicely with CLOD as described earlier. To spherify a cube there is a couple of mathematical mappings. The first being normalising each point. This means converting each 3D points components to a value between 0 and 1. The only possible downside to this method is that there isn't an even distribution of triangles. There will slightly squashed triangles towards the edge of each cube face. A better mapping is one described by Math Proofs in 2005, which produces a much smoother distribution of points. Figure 9 shows this mapping.

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} x\sqrt{1-\frac{y^2}{2}-\frac{z^2}{2}+\frac{y^2z^2}{3}} \\ y\sqrt{1-\frac{z^2}{2}-\frac{x^2}{2}+\frac{z^2x^2}{3}} \\ z\sqrt{1-\frac{x^2}{2}-\frac{y^2}{2}+\frac{x^2y^2}{3}} \end{bmatrix}$$

Figure 9 - Mathematical mapping of a cube to sphere

2.5 Summary

In this chapter is a discussion of various level of detail algorithms, methods of generating terrain procedurally and how to render various aspects of a planet.

3 Project Planning

3.1 Introduction

In this chapter are the methods associated with planning the project, including which methodology is used, how requirements are mapped out and how source control influences the development of the system. Additionally, there are a couple of legal issues to look out for.

3.2 Methodology

The methodology chosen to develop this project is agile based. It is close to the Dynamic Systems Development Method (DSDM) (Muslihat, 2018). This is due to iterative development, MoSCoW prioritisation and time boxing. Iterative development "is a way of breaking down the software development of a large application into smaller chunks" (Rouse, 2016). So, in each iteration, features are added which builds upon the system. Iterations are often used in conjunction with sprints and timeboxes. A sprint is a certain predefined time in which these iterations occur and the development during the sprint is reviewed at the end. A timebox is a length of time for a particular feature to be completed.

The approach taken in this project uses a pseudo-timeboxed method. There is no precise time keeping. Instead, if features are taking longer than previously thought, it's time to switch the feature to work on. This is only valid for lower priority features, as sufficient time must be spent on the fundamentals.

3.3 Source control

Source control is the organised tracking of code during development. It's especially useful when working in teams, but still beneficial even on single person projects. Source control allows you to commit changes and commits can be rolled back if something didn't work out. Branches extend the repository from a certain commit. Features are usually branched out, so that they can be merged into the main branch later. When merging, the source control software will try it's best to combine source files without conflicts, but sometimes conflicts occur and have to be fixed manually. Using branches to develop features is especially useful when a feature may not work

out very well. In this case, the branch can just be left, and development can continue on another branch. The advantage of this is that you don’t have to manually revert changes back.

This projects development will use Git as the source control software. There is a great Git cloud hosting site called GitHub. Not only is great for backing up the project but working on the project on multiple machines is very easy because it can be cloned from GitHub locally, and changes are pushed back to the GitHub servers and available from other machines.

3.4 Requirements

3.4.1 Introduction

This section details the requirements using the MoSCoW technique. The method is to implement the must have features first, then work downwards through the list. This way, a priority system is in place to prevent the necessary features not being implemented due to time constraints.

3.4.2 Must have

- Level of detail planet with terrain which changes detail based on the camera position.
- Diffuse lighting support.

3.4.3 Should have

- Planetary atmosphere, this could be tricky due to the complicated maths surrounding atmospheric scattering, however there are sample implementations for reference.
- Add dynamically animated clouds, using 3-dimensional/4-dimensional noise.
- Surface water rendering.
- Completely smooth terrain with no cracks due to level of detail.
- Smooth lighting with no seams due to level of detail.

3.4.4 Could have

- Biomes, for example: desert, grassland, mountains.
- GPU based geomorphing in the terrain, so there is no popping.
- Multiple planets arranged in a system.
- Camera centred origin for travelling large distances.

3.4.5 Won't have

- Physics, in terms of colliding with the planet. This might be out of the scope of the project as the focus is on rendering the planets.

3.5 Tools and Techniques

There are a variety of renderers available, for example: DirectX, OpenGL and Vulkan. The latter two are cross platform, DirectX is Microsoft Windows only. Computer games are generally targeted for Windows because the operating system market share is Windows dominated (Statista, 2019). There isn't much demand for Linux based games which is why the project uses DirectX. Another option would be to use a game engine. Using a well-established game engine would speed up development and could target multiple operating systems. This approach was considered but for maximum flexibility and performance the project uses purely C++ and DirectX.

The integrated development environment commonly used to develop C++ and DirectX applications is Microsoft Visual Studio. It contains many debugging tools including a graphics debugger. GitHub Desktop will be used for a Git graphical interface to assist with source controlling the project. To generate documentation for the project, Doxygen will be used. Doxygen works by parsing comments attached to source files, it will then produce a HTML documentation for all the classes and methods.

3.6 Legal and Ethical Issues

A possible legal issue is Ken Perlin's patent on Simplex Noise. The patent is only valid however, for image generation. Whereas in this project it's used for terrain

generation. On the other hand, there is a similar project called OpenSimplex noise, which is free to use in any scenario.

Lastly, when texturing the planet, only royalty free textures will be used. As a developer and not an artist, producing original assets is out of scope. Extra attention is paid to finding textures online to ensure there aren't any copyrights attached to them and they state they're free to use.

3.7 Summary

This section discussed the methods used throughout the development of the project and how certain practices can increase productivity. There is a couple of legal issues to note down but there shouldn't be any issues.

4 Design

4.1 Introduction

This chapter documents how the main sections of the engine are mapped out in a way that it can easily be implemented. There are methods explained to reduce global memory consumption and help with the abstraction of low-level code.

4.2 Class Structure

When designing the system, emphasis was put on designing a firm class structure. The principle of dependency inversion is used. This states that high level classes should not depend on low level ones as this creates tight coupling. The principle states that instead, they should depend on abstractions (Gkatziouras, 2018). Throughout the design process there are many interfaces created where there are always concrete implementations. Figure 10 shows the class diagram for the physical bodies. Physical bodies include objects such as planets and stars. Planets and stars, though two different types of objects, still have common properties. These could include, but are not limited to: mass, position and velocity.

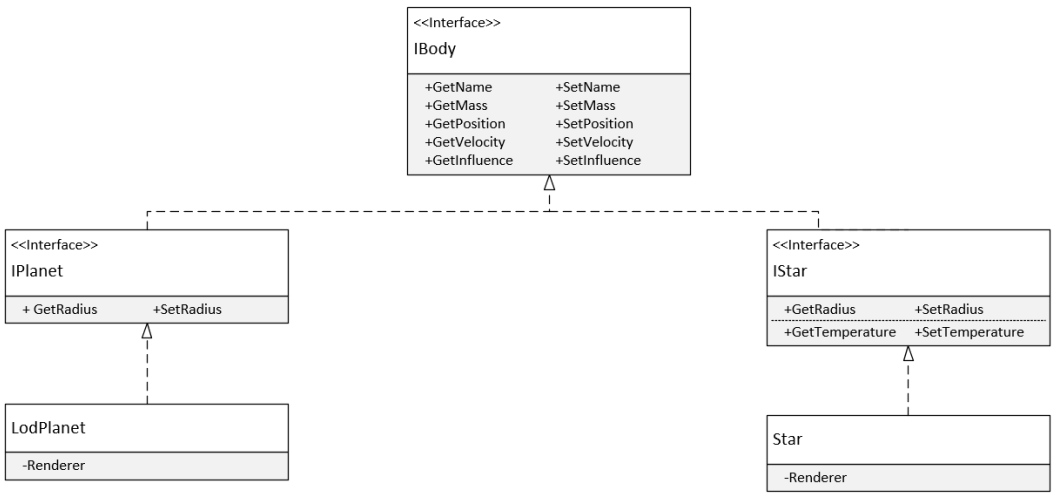


Figure 10 - Class diagram for physical bodies

Now the classes for the main physical bodies are laid out, they still need a method to be rendered. A design decision in this project was made to separate out the

rendering code from the logic of controlling the bodies. The main reason for this is there could be multiple renderers for one body. This is to accommodate the potential for another level of detail method in addition to quadtree level of detail. This method could be not render quadtrees when the planet is a large distance away, instead render just a sphere to save processing time. Hence the decision to support a pointer to a renderer rather than implement one within the container class. Figure 11 shows the class diagram for rendering bodies. The Drawable class is a helper class for rendering DirectX geometry to the screen. The SimpleStarRenderer and QuadtreeTerrainNode are classes which yield this ability. Also note the dependency inversion for the spherical terrain classes. QuadtreeTerrainNode depends on SphericalQuadtreeTerrain and vice versa. This is called a circular dependency. To solve this, the SphericalQuadtreeTerrain class is ‘inverted’, this means an interface is created and it implements it. This way the QuadtreeTerrainNode can depend on an abstraction rather than a concrete class, hence avoiding a circular dependency. Additionally, circular dependencies are allowed, they just create high coupling, and this is something to be avoided in favour of loose coupling. Loose coupling is an approach where a class doesn’t need to know how the dependency is implemented, instead it just needs to know how to use it.

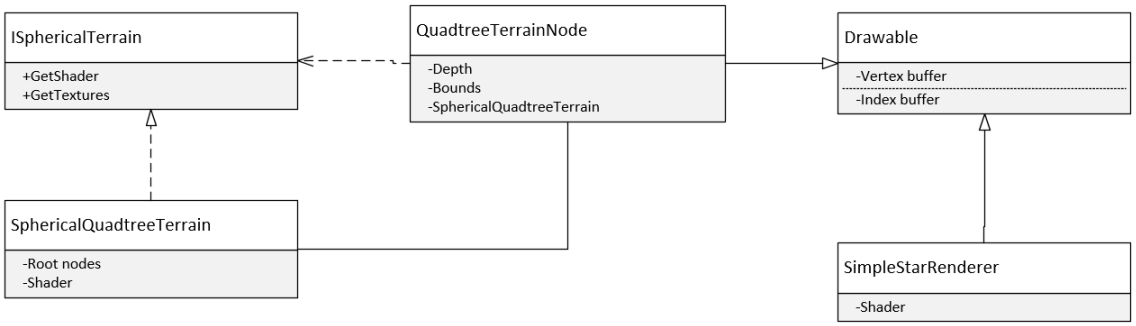


Figure 11 - Class diagram for renderers

4.3 Manager Classes

Manager classes are classes which manage the lifetime of resources. The resources which need to be managed in this project are textures and shaders. Manager

classes take the bloat away from other engine classes and separate it out into secondary interfaces. They manage the creation, deletion, loading, saving and clean-up of resources. There is multiple way to handle the clean-up of resources. The first being deleting them all at the end of the program. The second is when there may be memory constraints, so they may be deleted if the resource hasn't been accessed for a while.

In this project the texture and shader manager classes will be implemented using lazy initialisation (Microsoft, 2017). This is where the resource is only loaded until its first needed. This allows a very simple implementation as there is no need to preload a list of resources at the start of the program, which would otherwise add complexity.

4.4 Helper Classes

The project will make use of a few classes to help abstract away lower level code. The first being the shader class. It's responsible for loading in a vertex and pixel shader pair. Due to shaders being written in High Level Shader Language (HLSL), when they're loaded in it takes some time as the DirectX libraries have to parse and compile the shaders. Once the shaders are loaded the buffers need to be stored. Due to the two buffers being encapsulated in the class, it's easy to manage them without losing track. Grouping data is one of the main purposes of classes. The second helper class is the Drawable class. In a similar fashion to the former helper class, there are a few pieces of data which need to be stored, and it's better if they're stored together. The Drawable class is synonymous to a mesh. The purpose is to store the vertex and index data. Traditionally, a model is an instance of a mesh, meaning you can render multiple models, but the actual geometry is only stored once. In this project there are no multiple instances this time, but the framework is in place if there is need to hold a repeating model.

4.5 Summary

This chapter discussed the fundamental engine design for the project and how it's useful to abstract out particular parts of the code. There are also some important memory optimisation techniques are discussed using manager classes.

5 Implementation

5.1 Introduction

In this chapter you will find the details of the implementation for each area within the engine. Code listings of various algorithms are also described.

5.2 Quadtree Planet

5.2.1 Quadtree basics

Firstly, to implement LOD terrain we need to start off with a basic quadtree implementation. A basic quadtree stores a pointer to each of its four children; these children pointers may be null, so in that case it's a leaf node. The code listing below shows the data structure of a quadtree, containing these pointers plus some methods to modify the quadtree's state.

```
struct Square
{
    float x, y;
    float size;
};

class TerrainNode
{
        public:
            enum EQuad { NE, NW, SE, SW };

            TerrainNode(TerrainNode *parent, Square bounds);

            bool IsLeaf();
            void Split();
            void Merge();
            bool IsRoot();

        private:
            Square m_bounds;

            TerrainNode *m_parent;
            TerrainNode *m_children[4];
};
```

Listing 1 - [TerrainNode.cpp] Stripped down version of the TerrainNode (quadtree) class

The second fundamental is to be able to split and merge node(s) in the quadtree. The listing below shows a pseudocode for the split function. The function creates four new quadtree nodes at north west, north east, south west and south east. Each child node will be a quarter of the size of the parent (half the width and half the height creates one quarter). In addition, the split function checks if the node is a leaf node (deepest node, with no children). It does this to it doesn't accidently split a node which already has children, and it will propagate down the tree until it finds a lead node to split. This makes it a recursive function.

```
Function Split
    Return is maximum depth is reached

    If this node is a leaf node
        Create 4 new nodes at North East, North West, South East, South West
    Else
        Foreach child node, call the split function on the child
```

Listing 2 - [TerrainNode.cpp] Function to split the node into four more nodes

Lastly, we have the merge function, which is recursive just like the split function. Its purpose is to merge four leaf nodes into one node. The merge function should only be applied on nodes where its four children are leaf nodes. If the current node contains children which aren't leaf nodes, then it will traverse down the tree until it finds a node which its children are leaf nodes. It will then free up the memory of the children and reset the pointers which effectively declares the current node as a leaf node.

```
Function Merge
    Return if this node is a leaf node

    If any child of this node is a leaf node
        Delete all this nodes' children
        Notify all neighbours of this node that an LOD change happened
    Else
        Foreach child node call the merge function of the child
```

Listing 3 - [TerrainNode.cpp] Function to merge four nodes into one

Now the basic quadtree structure is in place, the following code creates the root node. This node's bounds range from (-0.5, -0.5) to (+0.5, +0.5). This snippet is located in a terrain manager class, which is responsible for storing data which will be accessed by many nodes of the quadtree.

```
TerrainNode *node = std::make_unique<TerrainNode>(null, Square { -0.5f, -0.5f, 1.0f });
```

Listing 3 - [SphericalQuadtreeTerrain.cpp] Creating the root quadtree node

5.2.2 Spherical quadtree terrain

Firstly, each quadtree node needs to store geometry and be able to render it to the screen. There is a helper class called Drawable to isolate the DirectX rendering. It handles the initialisation of the vertex and index buffers, plus it provides a pre-render function to call DirectX context functions in preparation for rendering. The code listing below shows the updated TerrainNode class definition.

```
struct Square
{
    float x, y;
    float size;
};

struct PlanetVertex
{
    DirectX::SimpleMath::Vector3 position;
    DirectX::SimpleMath::Vector3 normal;
    DirectX::SimpleMath::Vector3 uv;
};

class TerrainNode : public Drawable<PlanetVertex>
{
        public:
            enum EQuad { NE, NW, SE, SW };

            TerrainNode(TerrainNode *parent, Square bounds);
```

```
            bool IsLeaf();
            void Split();
            void Merge();
            void Generate();
            void Render(Matrix view, Matrix projection);
            void Update(float dt);
            bool IsRoot();

        private:
            Square m_bounds;

            TerrainNode *m_parent;
            TerrainNode *m_children[4];
};
```

Listing 4 - [TerrainNode.hpp] Updated class definition of TerrainNode

Now there is a method to render geometry to the screen, but we still need geometry to render. This next code listing provides a way to create a flat plane with support for calculating normals.

```
Function Generate
    Variable step -> (Bounds.size / Gridsize - 1)

    // Generate vertices
    Loop y from 0 to gridsize
        Loop x from 0 to gridsize
            Add vertex at (Bounds.x + x * step, 0, Bounds.y + y * step)

    // Generate indices
    Loop y from 0 to gridsize - 1
        Loop x from 0 to gridsize - 1
            Add 2 triangles

    // Calculate normals
    Loop through all faces
       Calculate normal of face
       Add normal to 3 vertices of face (they will be normalised in shader)
```

Listing 5 - [TerrainNode.cpp] Generating a flat plane

Finally, the terrain should be spherical. This is done by arranging six quadtrees into a cube, then using a mathematical mapping to convert the cube into a sphere. The following code listing shows the six quadtrees being constructed in the appropriate orientations.

```
std::array<Matrix, 6> orientations = {
    Matrix::Identity,
    Matrix::CreateRotationX(XM_PI),
    Matrix::CreateRotationX(XM_PI / 2),
    Matrix::CreateRotationX(-XM_PI / 2),
    Matrix::CreateRotationZ(XM_PI / 2),
    Matrix::CreateRotationZ(-XM_PI / 2)
};

for (int i = 0; i < 6; ++i)
    m_faces[i] = std::make_unique<TerrainNode>(this, Square {-0.5f,-0.5f,1.0f});
    m_faces[i]->SetMatrix(orientations[i]);

for (int i = 0; i < 6; ++i)
    m_faces[i]->Generate();
```

Listing 6 - [SphericalQuadtreeTerrain.cpp] Creating a quadtree cube

Next, the geometry generation code in the quadtree node is modified to spherify the terrain. Previously, the flat geometry is rendered at a height of 0 units. However, rendering at 0.5 units away from the origin will form a unit cube (side lengths of 1), and subsequently a sphere with a radius of 1. The code listing below shows a method to spherify the geometry. This method requires the terrain to be rendered at 0.5 units away in its orientation, so then normalising each vertex will adjust it to form a sphere.

```
Vector3 pos = Vector3(xx, 0.5f, yy);
pos.Normalize();

PlanetVertex v;
```

```
v.position = pos;


m_vertices.push_back(v);
```

Listing 7 - [TerrainNode.cpp] Transforming a cube into a sphere

Lastly, each quadtree node should increase/decrease in LOD depending on the camera position. To do this we need to have a split distance scalar. This scalar variable is the distance between quadtree depths at which to split at. To use it we first calculate the distance from the camera to the centre of the node, then check if it's less than the split distance scalar multiplied by the scale of the node. The scale of the node is defined as 1/depth.

To split and merge the nodes correctly, we need to define some rules. The node should only merge if it shouldn't be divided and isn't a leaf node, because to merge it needs child nodes. The node should only split if it is a leaf node (to prevent duplicated children) and it should be divided due to the camera being close enough to the scaled node distance. Also note this is a recursive function so if none of these conditions are satisfied then it will propagate down the tree. The code listing below puts these calculations and rules into practice.

```
Function Update
    Variable Divide -> If Distance to node is less than Scale x SplitDistance

    If not a leaf node and Divide
        Merge node

    If leaf node and Divide
        Split node
    Else if not a leaf node
        Foreach child node, update the child
```

Listing 8 - [TerrainNode.cpp] Update function to determine if the node is to be split or merged

With all of the above implemented, Figure 12 shows what the planet looks like at this stage. The vertices are randomly coloured, so it shows up clearer.

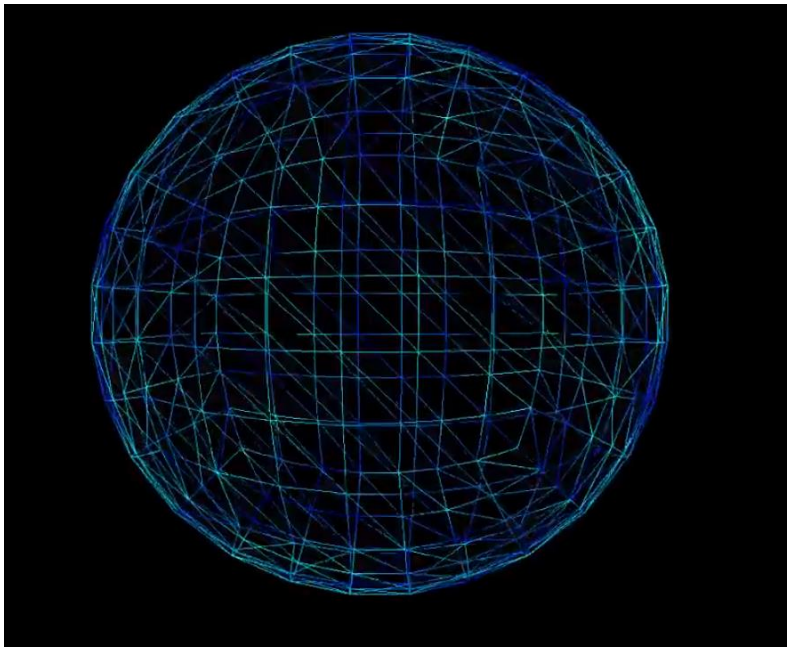


Figure 12 - Spherical 6-quadtree planet based off a cube

5.2.3 Fixing terrain cracks

As certain adjacent quadtree nodes can be at different depths, this will cause cracks in the geometry. To implement a fix for this, we first need to be able to locate a node's neighbours. This requires a neighbour finding algorithm, the one implemented is based on the examples provided from Geier in 2017. The two algorithms are located in appendix 3 and 4.

Figure 13 - Terrain cracks

Next, whenever there is a LOD change, the neighbours of the split/merged node need to be notified of this change, so the geometry can be stitched together. The following function will call the stitch edge function on each of its neighbours.

```
Function NotifyNeighbours
    Variable Neighbours : List

    Neighbours += GreaterThanOrEqualToNeighbours for North, East, South, West
    Neighbours += GetSmallerNeighbours for North, East, South, West

    Foreach Neighbour call function FixEdges
```

Listing 9 - [TerrainNode.cpp] Function to notify other neighbours of an LOD change

To finish, the edges of the nodes' geometry need to be stitched together. There are nine different index permutations of the geometry which are generated in a preprocessing step at the start of the program. The vertices themselves don't need to be changed, we can just change which triangles are rendered. A part of the preprocessing step is shown below.

```
/*
    Generate 'Top' permutation
*/

var triangles = [FlatPlane]
var index = 0

// Remove all even top triangles
loop through triangles
    if (index is even) and (triangle's first point is 0)
        remove triangle

    increment index

// The removed triangles are replaced by larger ones to match a lower LOD
loop from x = 0 to gridsize - 2 incerement by 2
    Add bigger triangle to fill the pair of triangles removed earlier
```

Listing 10 - [SphericalQuadtreeTerrain.cpp] Generation of index permutations

Now that all the permutations are generated, the penultimate step is to select the correct index permutation derived from which neighbours have a lower detail. The code listing below demonstrates this. There is an extra step at the end called to check if the detail difference is equal, if so it uses the same normals as its neighbour's edge. This ensures smooth lighting across quadtree node boundaries.

```
Function FixEdges
    Variable Neighbours -> GetGreaterThanOrEqualNeighbours for all directions
    Variable LowerLods  -> Get all depths from each of the Neighbour directions
```

```
    If no LowerLods -> Select permutation ‘None’
    If LowerLods[North] -> Select permutation ‘Top’
    If LowerLods[East]  -> Select permutation ‘Right’
    ...
    If LowerLods[North] And LowerLods[East] -> Select permutation ‘TopRight’
    ...
    Foreach Neighbour
        If detail difference is equal to neighbour
            Update vertex normals along edge to that of the neighbour
```

Listing 11 - [TerrainNode.cpp] Choosing the index permutation for a quadtree node

5.2.4 Optimisations

The first optimisation implemented is the reuse of vertices. When splitting a quadtree, even indices of children will overlap with their parents. Instead of calculating the vertex position again, we can get the associated vertex from the parent, thus saving processing time. Listing 12 shows the pseudocode for getting the start x and y index for the parent, due to the child being in a particular quadrant of its parent. Listing 13 shows the modified vertex loop.

```
switch Quad
    NW: start_x = 0, start_y = 0
    NE: start_x = gridsize / 2, start_y = 0
    SE: start_x = gridsize / 2, start_y = gridsize / 2
    SW: start_x = 0, start_y = gridsize / 2
```

Listing 12 - [TerrainNode.cpp] Translating the current quadrant into start coordinates of the parent’s vertices

```
PlanetVertex v

if (hasParent && (x mod 2 == 0) && (y mod 2 == 0))
    // Translating to parent space
    var x_half = start_x + x / 2
    var y_half = start_y + y / 2

    v = parent->GetVertex(x_half + y_half * gridsize)
```

```
else
    ... Calculate vertex normally
```

Listing 13 - [TerrainNode.cpp] Vertex loop reuses parent vertex if possible

The second optimisation implemented was not rendering quadtree nodes if they're beyond the visible horizon. The general formula for the distance to the horizon, where 'h' is height above the planet and 'R' is the radius of the planet is $d = \sqrt{h(2R + h)}$ (Wikipedia, 2019). The pseudocode below shows the method for checking if the node should be visible or not.

```
var distanceToNode = length(camera, nodeCentre)
var heightAbovePlanet = length(camera - planetPosition) - planetRadius
var horizon = root(heightAbovePlanet * (2 * planetRadius + heightAbovePlanet))
var isNodeVisible = distanceToNode < horizon
```

Listing 14 - [TerrainNode.cpp] Culling nodes beyond the horizon

5.3 Terrain Generation

5.3.1 Noise

To implement random terrain, a library called FastNoise. This is because it already implements fractal simplex noise. However, there needs to be more control over the terrain to create more realistic planets. As the planet is very large, using one noise map will not utilise all the space there should be more detail up close. The solution is to use multiple noise maps and layer them over each other. One for the general shape of the geometry and one 20x smaller for fine detail up close.

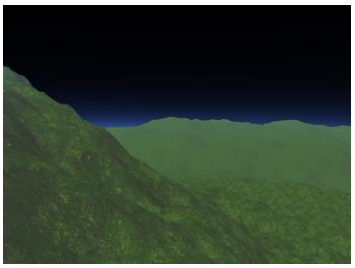


Figure 14 - Terrain generated with Simplex Noise

```
float GetHeight(vec3 p)
{
    float mod = 1.0f;

    // Initial noise map
```

```
    float e = m_noiseMaps[0].GetNoise(p.x, p.y, p.z);

    // Finer noise maps
    for (int i = 1; i < numNoiseMaps, ++i)
    {
        mod *= settings.NoiseMaps[i].Mod;
        h += m_noiseMaps[i].GetNoise(p.x, p.y, p.z) * mod;
    }

    return h;
}
```

Listing 15 - [SphericalQuadtreeTerrain.cpp] Function to calculate height of geometry at a particular point

5.3.2 Biomes

Biomes are implemented using noise maps, but this time instead of using noise to calculate the height, it uses it for moisture. At the start of the program a biome lookup map is generated from a settings file. The settings file lays out which biomes appear at certain moisture and elevation values. The x axis of the image represents moisture, and the y represents elevation. The biome lookup map is used on the CPU side to determine which texture to use, and the GPU side to sample the colour in the pixel shader.

Figure 15 - Generated biome lookup map

5.3.3 Water

Water is implemented in a similar way to flat terrain. On flat terrain the standard way is to render a flat plane geometry at a predefined water level over the previous geometry. In this case a sphere is rendered as the geometry is spherical. The spherical water uses the same quadtree CLOD method as the terrain. This is because a pre-generated sphere isn't smooth enough up close. The only difference between the water and the terrain is that the water is rendered using a different

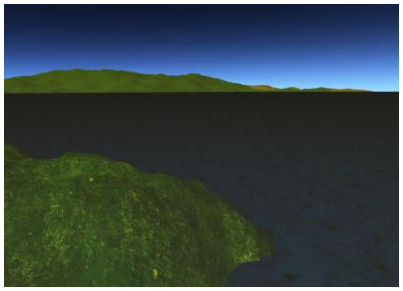


Figure 16 - Water layer

shader, and no noise values are sampled due to water being smooth. The water shader uses a very simple technique where the normal map texture coordinates are moved slowly over time. The code snippet below shows the only lines needed for this to work.

```
cbuffer WaterBuffer : register(b2) {
    float mScroll; // Updated every frame CPU side
}

...

float2 scrolledUV = float2(v.UV.x + mScroll, v.UV.y + mScroll);
float3 normalMap = NormalTex.Sample(Sampler, scrolledUV);
```

Listing 16 - [WaterPS.fx] Scrolling the normal maps to produce a simple wave effect

5.4 Atmosphere

5.4.1 Atmospheric scattering

Atmospheric scattering is implemented based off the code from GPU Gems 2 (O'Neil, 2006). It uses volumetric rendering which means sampling various points along a ray to calculate a colour. To set it up, a sphere with a slightly larger radius of the planet is rendered. However, instead of using cull back, cull front is used which gives a silhouette of the sphere around the planet. Therefore, giving the effect of an atmosphere around a planet. The same shader can be adapted to work with surface scattering. This means from space the ground looks like it's beneath the atmosphere. Plus, on the ground, distant hills are fainter due to the atmosphere in between just like in real life. This is different to fog however, because fog is the water vapour in the atmosphere.

Figure 17 - Atmosphere from space as seen from sunrise

5.4.2 Clouds

Adding clouds starts off with a similar method to the atmosphere. A sphere is rendered which is slightly larger than the planet, how much is at what level you want the clouds. The sphere is rendered with vertices only containing a position. This is because the vertex shader then implements a 3D simplex noise which can layer this noise to create fractal noise. In addition, on the CPU side, a timer is sent every frame to the shader to move the clouds. This fractal noise is then converted to grayscale. The code listing below shows this vertex shader.

```
float3 pos = v_in.vPosition;
pos.z += time;

float col = snoise(pos);
col += snoise(pos * 2) * 0.5f;
col += snoise(pos * 4) * 0.25f;
col += snoise(pos * 8) * 0.125f;

...
Output.Colour = float4(col, col, col, col);
```

Listing 17 - [CloudsVS.fx] Clouds vertex shader using 3D noise

Figure 18Figure 18 - Picture of the planet with the cloud layer shows an example of what the clouds look like from space. When running the program, the clouds are animated, and due to the 3-dimensional noise the clouds actually change shape over time.

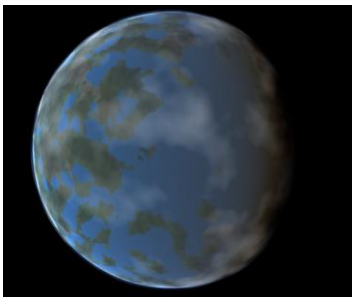


Figure 18 - Picture of the planet with the cloud layer

5.5 Summary

This section described all the details of implementation, and some extra bits of research that went into the knowledge required to do so.

6 Test Strategy

6.1 Introduction

In this section are test strategies for examining the building blocks of the system, up to testing it against the intended requirements. Some sample tests are provided for the project.

6.2 Unit Testing

Unit testing is the name given to the process of testing the smallest testable parts of a system (Rouse, 2017). Unit tests are often performed on class methods as it's generally the smallest part of a system which groups functionality. On average, systems can have hundreds, if not thousands of these methods. Therefore, unit tests are often automated as going through them manually is possible, but it takes much longer. Unit testing is often a part of test-driven development (TDD), which typically results in predictable systems.

In this project, an example unit test would be testing the quadtree neighbour finding algorithm. The unit test would first set up a basic quadtree with some child nodes. Then by hand the tester would work out the neighbours expected in the test quadtree. Lastly, the unit test would then check if the neighbour's algorithm function returns all the correct neighbours.

6.3 System Testing

System testing is the process of testing the entirety of the system to check whether it complies with the requirements (Software Testing Fundamentals, 2011). A black box testing strategy is generally used in conjunction with system testing. This strategy hides the implementation from the tester, unlike unit testing. System testing attempts to find errors mainly in behaviour.

System testing is used in this project to check whether it meets the initial requirements. The requirements earlier in this project are laid out using MoSCoW. These can be easily carried out as system tests.

6.4 Other Types of Testing

Another type of testing is integration testing. This is where unit tests are combined into groups and tested if they work together. An example using this project is fixing the quadtree terrain cracks. The individual units required to fix them include neighbour finding and a fix edges function. The goal is to test that the output from the neighbour unit successfully integrates with the fix edges function, hence fixing the cracks.

Furthermore, acceptance testing is another type of testing which is used to evaluate the systems compliance with the business requirements. System testing is carried out by the developers, whereas acceptance testing is performed by the clients. If this project were being developed for a client, then this stage would be needed. However, there are no clients involved.

6.5 Test Results

Table 1 shows the unit test results. These are the smallest possible tests that can be carried out with the lowest level of functionality.

Table 1 - Unit testing results

Test #	Test case	Expected outcome	Actual outcome	Action required
1	Sample quadtree constructed and neighbours are searched from within a node	Quadtree 1_2_2 neighbours with nodes 1_1_2_3 and 1_1_2_4	Quadtree 1_2_2 neighbours with nodes 1_1_2_3 and 1_1_2_4	None
2	Testing the IsRoot method on a quadtree root node	True	True	None
3	Testing the IsRoot method on a quadtree child node	False	False	None

4	Quadtree GetDepth function tested on a nested child node	4	4	None

Table 2 shows the system testing results. These test against the initial requirements of the system.

Table 2 - System testing results

Test #	Test case	Expected outcome	Actual outcome	Action required
1	Planet contains clouds	Animated clouds scroll across the planet	Animated clouds scroll across the planet, but they disappear when the camera is underneath	Fix the cloud layer to render when the camera is beneath them
2	LOD works as intended	Pressing ‘Q’ and flying close to the planet shows the LOD increasing	Pressing ‘Q’ and flying close to the planet shows the LOD increasing	None
3	The terrain contains no cracks	Terrain when up close contains no clear seams and is completely smooth	Terrain when up close contains no clear seams and is completely smooth	None

6.6 Summary

This chapter explained the fundamental testing techniques for systems. It covered how these strategies can be applied to this project, with some sample tests for reference.

7 Evaluation, Conclusions and Future Work

7.1 Project Objectives

In this project most of the objectives were achieved. The engine is able to render an Earth sized planet using level of detail techniques. Some degree of realism to the planet is added through the use of clouds, atmospheric scattering, water and biomes. There was a couple of other objectives like organising multiple planets and a star into a solar system. Unfortunately, even though progress was made on this, it wasn't at a satisfactory level. With more time this side goal could have been accomplished.

7.2 Self-Evaluation

Reflecting back onto the start of this project, I was very ambitious. I planned out many features with the hope that this would challenge myself to complete more work. I combined my passion for space and games programming, which gave the perfect project which allowed me to be motivated the entire time. At the start of this project I knew the basics of DirectX and rendering, but nothing about rendering life sized planets. After many days of research, I managed to grasp the core concepts, but as development was underway I was still learning and researching techniques.

My main weakness is underestimating the time it takes to complete a feature. The original Gantt chart incorporated all the initial features. There should have been more time assigned to some areas of the development to allow for leeway. My strengths included the general motivation to carry out the development. Plus, good researching skills needed to dig out rare information related to specific topics within LOD planets.

7.3 Project Evaluation

The level of detail system implemented works perfectly as intended. The biggest hurdle in the project was fixing the cracks in the terrain. Although in the early stages of the development, an implementation for fixing the cracks was in place, there was no smooth lighting. It was only nearing the end of the project when it was figured out how to do this, as there is very little documentation on smooth lighting across quadtree boundaries. One of the other methods tried for fixing these cracks was

geometry tessellation. When researching it, there was a specific piece of information missed. So, when it was implemented it was realised that it doesn't actually fix terrain cracks. Due to the use of version control, attempted features and fixes like this could just be discarded, and development can continue from the last working version. Or, if features were just taking too long, the branch was left, and development may have resumed later when other features were finished.

There were many more hurdles and failed features during development. Early on, GPU noise was experimented with to improve the render performance. However, in order for biomes to be based off elevation, or potential future work for colliding with the terrain, it has to be done on the CPU. Many optimisations were carried out throughout development, until it got to the point where the bottleneck was in the noise generation library itself. In an attempt to optimise further, a repository for a SIMD implementation of FastNoise (Peck, 2016) was stumbled upon. However, this didn't work out, mainly due to the library working in integer sampling as opposed to floats. It was never figured out if it's possible or not, it probably is but more research and time would be needed. Other failed features included: geomorphing (to remove popping), specular lighting (planet became way too shiny) and using double precision floats to try and remove camera jitter as planets moved, but this didn't solve it. A last big hurdle was debugging, since there were so many nested quadtree nodes, without a game engine there was no method to find which node was which easily. So, testing for neighbours proved difficult at first but it eventually worked after giving each quadtree debug names to assist with this.

Throughout the development, many optimisations were applied. The reason there was so much optimisation is because as a side project to this one, a planet sandbox application was developed. It was a fairly quick to develop it, and its purpose was to customise the planets by adjusting the properties, so they could be imported back into the main project. Optimisation was needed as when adjusting a slider, the planet would have to be regenerated. Eventually, the planet generation time was lowered to ~60ms from seconds. The biggest time knocked off was due to the loading of textures and shaders every time the planet was generated. Secondary optimisations which still had significant performance increases were: using parents' vertices when possible, pre-generated index permutations, texture arrays and multithreading.

Furthermore, the planet sandbox application supported the objective of being used as a general engine. It was fairly straightforward to compile the existing engine as a library, which was used to develop the sandbox application.

7.4 Applicability of Findings to the Commercial World

As mentioned earlier, one of the objectives was to turn the project into a general engine to be used for games which require procedural planetary rendering. This has commercial potential to be used in a variety of games. Especially if collision is implemented into the engine. Additionally, the skills learnt during the development could be very valuable to companies working on similar projects. In the past few years there have been many games based on this premise, and probably more to come.

7.5 Conclusions

In conclusion, this project was a success. A generic planetary renderer has been developed which supports biomes, water, atmospheres and clouds. A side project used this renderer to be able to customise these planets and provided many hours of fun during development. The knowledge gained from this project is massively useful and can certainly be reapplied in the future.

7.6 Future Work

One limitation is the flexibility of the system. The planet class consists of multiple renderers: clouds, terrain, water and atmosphere. Currently, to add a component, more renderers need to be coded into the class. This is where a component system should have been used. This way, renderers can be unique components which can be added or removed from the planets. It makes much more sense to use this way in a generic engine.

There is a fairly simple optimisation that could be made when rendering. View frustum culling could be used to prevent the rendering of non-visible terrain nodes. This combined with the horizon-based culling would only render terrain nodes in view and would increase the framerate. The second optimisation which could be added is

compute shaders. As discussed earlier, GPU noise is faster, but we need access to the new geometry to be able to do collision, plus various other things. Instead, a compute shader can be run to generate the geometry, then the results streamed back to the CPU to be used. This would improve the framerate when flying around as geometry is generated quicker.

Lastly, to render very large worlds, and travel between Earth sized planets, there needs to be some sort of floating origin. If there was enough time this could have been implemented, as the sufficient research had already been done. One technique is whenever there has been an appropriate distance travelled, all the objects in the scene are shifted back, so the camera is then back at the origin (Kerbal Space Program, 2013). This means floating point errors cannot accumulate over time and produce shaky movement due to insufficient precision. The other method is to move all the objects in the scene around the camera, so the camera always stays at the origin.

References

Andersson, J. (2009) *Terrain Rendering in Frostbite Using Procedural Shader Splatting,* New Orleans: SIGGRAPH.

Archer, T. (2011) *Procedurally Generating Terrain,* Sioux City: MICS.

Arul Asirvatham, H. H. (2005) 'Terrain Rendering Using GPU-Based Geometry Clipmaps', in *GPU Gems 2,* Upper Saddle River, N.J.: Addison-Wesley, pp. 27-45.

Boer, W. H. d. (2000) *Fast Terrain Rendering Using Geometrical MipMapping,* s.l.: E-mersion Project.

Brano Kemen, L. H. (2009) *Logarithmic Depth Buffer,* https://outerra.blogspot.com/2009/08/logarithmic-z-buffer.html (accessed 12 4 2019).

Elek, O. (2009) *Rendering Parametrizable Planetary Atmospheres with Multiple Scattering in Real-Time,* Prague, Czech Republic: Faculty of Mathematics and Physics, Charles University.

Frank Losasso, H. H. (2004) 'Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids', *ACM Transactions on Graphics.*

Gamepedia, (2019) *Biome,* https://minecraft.gamepedia.com/Biome (accessed 12 4 2019).

Geier, D. (2017) *Advanced Octrees 4: finding neighbor nodes,* https://geidav.wordpress.com/2017/12/02/advanced-octrees-4-finding-neighbor-nodes/ (accessed 11 4 2019).

Gkatziouras, E. (2018) *SOLID Principles: Dependency Inversion Principle,* https://dzone.com/articles/solid-principles-dependency-inversion-principle (accessed 13 4 2019).

Jian Wu, Y.-y. C. Z.-m. C. X.-j. W. (2010) *A New Quadtree-based Terrain LOD Algorithm ,* China: The Institute of Intelligent Information Processing and Application, Soochow University.

Kerbal Space Program, (2013) *'Building a new universe in Kerbal Space Program',* Vancouver: Unite 2013.

Kunio Aizawa, K. M. S. K. R. K. a. J. F. (2008) 'Constant time neighbor finding in quadtrees: An experimental result', *2008 3rd International Symposium on Communications, Control and Signal Processing,* pp. 505-510.

Math Proofs, (2005) *Mapping a Cube to a Sphere,* http://mathproofs.blogspot.com/2005/07/mapping-cube-to-sphere.html (accessed 12 4 2019).

Microsoft, (2017) *Lazy Initialization,* https://docs.microsoft.com/en-us/dotnet/framework/performance/lazy-initialization (accessed 13 4 2019).

Muslihat, D. (2018) *Agile Methodology: An Overview,*
https://zenkit.com/en/blog/agile-methodology-an-overview/
(accessed 12 4 2019).

N¨atterlund, M. (2008) *Water Surface Rendering,* Umeå: Ume˚a University.

O'Neil, S. (2006) 'Accurate Atmospheric Scattering', in *GPU Gems 2,* Upper Saddle River, N.J.: Addison-Wesley.

Orr10c (2018) *Terrain generation - Interpolating between multiple biome height-maps,*
https://stackoverflow.com/questions/52878336/terrain-generation-interpolating-between-multiple-biome-height-maps/53563419
(accessed 12 4 2019).

Peck, J. (2016) *FastNoiseSIMD,*
https://github.com/Auburns/FastNoiseSIMD
(accessed 12 4 2019).

Perlin, K. (1983) *Noise and Turbulence,*
https://mrl.nyu.edu/~perlin/doc/oscar.html
(accessed 13 4 2019).

Perlin, K. (2002) 'Improving Noise', *Proceedings of the 29th annual conference on Computer graphics and interactive techniques,* 21(3), pp. 681-682.

Pulido, L. (2013) *Which is the best LOD method for planet rendering?,*
https://gamedev.stackexchange.com/questions/55731/which-is-the-best-lod-method-for-planet-rendering
(accessed 12 4 2019).

Red Blob Games, (2015) *Making maps with noise functions,*
https://www.redblobgames.com/maps/terrain-from-noise/
(accessed 11 4 2019).

Rouse, M. (2016) *Iterative development,*
https://searchsoftwarequality.techtarget.com/definition/iterative-development
(accessed 12 4 2019).

Rouse, M. (2017) *Unit testing,*
https://searchsoftwarequality.techtarget.com/definition/unit-testing
(accessed 12 4 2019).

Software Testing Fundamentals, (2011) *System Testing,*
http://softwaretestingfundamentals.com/system-testing/
(accessed 12 4 2019).

Statista, (2019) *Global market share held by operating systems for desktop PCs,*
https://www.statista.com/statistics/218089/global-market-share-of-windows-7/
(accessed 12 4 2019).

Ulrich, T. (2000) *Continuous LOD Terrain Meshing Using Adaptive Quadtrees,*
http://www.gamasutra.com/view/feature/131841/continuous_lod_terrain_meshing_.php
(accessed 11 4 2019).

Wagner, D. (2019) *Terrain geomorphing in the vertex shader,* Karlsplatz: Vienna University of Technology.

Wikipedia, (2019) *Distance to the horizon,* https://en.wikipedia.org/wiki/Horizon#Distance_to_the_horizon (accessed 12 4 2019).

Wikipedia, (2019) *Popping (computer graphics),* https://en.wikipedia.org/wiki/Popping_(computer_graphics) (accessed 13 4 2019).

Appendix 1 – Project Proposal

Name: Matthew Lowe **Course:** Computer Games Development

Discussed with (lecturer): Nick Mitchell/Laurent Noel

Current Modules (and previous modules if Computing or direct entrant)

Standard games course modules, no optional modules.

The Project Title

Procedurally Generated Planets

Project Context

I intend to generate and render planets which could be as large as Earth. The terrain will be procedurally generated, containing different biomes. The rendering will make use of Levels of Detail (LOD) to be able to render a planet so large. You'll be able to view the planet from space, or from the surface, in space there will be a visible atmosphere effect. I also intend to add multiple planets to the scene to form a small planetary system, with a central star with multiple planets/moons, simulating basic gravity of them orbiting each other.

The main issue is finding a way to render planets so large. I'll have to use a method of rendering different levels of detail depending on where the camera is (on the surface/in space).

Specific Objectives

1. Render a sphere with LOD
2. Procedurally generated terrain
3. Render atmosphere
4. Terrain biomes + water
5. Newtonian gravity in a planetary system

References

[1] Schaal, J., 2017. Procedural Terrain Generation. A Case Study from the Game Industry. Game Dynamics, pp.133–150.

[2] Aizawa, K. et al., 2008. Constant time neighbor finding in quadtrees: An experimental result. 2008 3rd International Symposium on Communications, Control and Signal Processing.

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

[4] Nowell, P., 2005. Mapping a Cube to a Sphere. Math Proofs. Available at: http://mathproofs.blogspot.co.uk/2005/07/mapping-cube-to-sphere.html [Accessed April 22, 2018].

[5] Ulrich, T., 2000. Continuous LOD Terrain Meshing Using Adaptive Quadtrees. Gamasutra. Available at: https://www.gamasutra.com/view/feature/131841/continuous_lod_terrain_meshing_.php?page=1 [Accessed April 22, 2018].

[6] Pi, X. et al., 2006. Procedural Terrain Detail Based on Patch-LOD Algorithm. Technologies for E-Learning and Digital Entertainment Lecture Notes in Computer Science, pp.913–920.

Potential Ethical or Legal Issues

There may be textures which I'll have to find online, for example: grass and water, to make the terrain more realistic. I'd make sure these textures are royalty free otherwise it could be copyright infringement.

Resources

I will make use of a graphics API, DirectX which supplies all the tools needed to create my project. Other resources include the papers I have referenced, which contain possible methods I'll need to use to be able to implement the goals I set out in my project.

Potential Commercial Considerations - Estimated costs and benefits

If this was done commercially the costs would come from paying the salaries for the developers. The project would be very worthwhile from a games company's perspective, since the project could be used as a general game engine to create multiple games from. For example, a space simulator, flight simulator, or an on-land game with a procedurally generated world. This saves development time greatly. Additionally, this project could potentially need artists, for creating realistically textured terrain, which includes more salaries.

Proposed Approach

1. [Week 1] Basic DirectX Engine
2. [Week 2] Render flat quadtree terrain [5] [6]
3. [Week 3] Map terrain to a sphere [4]
4. [Week 3] Use procedural method for generating random terrain [1]
5. [Weeks 4-5] Implement finding quadtree neighbours to fix detail differences [2]
6. [Weeks 5-7] Terrain biomes
7. [Week 8] Render water
8. [Weeks 8-9] Render realistic looking planetary atmosphere [3]
9. [Week 10] Orbital mechanics + spaceship
10. [Week 11] Add other planetary bodies

Appendix 2 – Technical Plan

Computing Project Technical Plan

Name: Matthew Lowe
Mode: Full time
Course: Computer Games Development
Supervisor: Gareth Bellaby

Title

Procedurally Generated Planets

Summary

The topic of the project is the rendering of a large, detailed planet. This has two aspects: rendering a large planet (up to the size of Earth) and secondly generating the terrain on the surface of the planet using procedural generation. The terrain will have various biomes including oceans, mountains and deserts, with simple trees and vegetation. A side goal of the project is to assemble a number of planets into a star system, along with simulating gravitational effects and being able to see the planets in the distance from afar.

The most viable solutions, in my opinion, include using QuadTrees for controlling LOD (levels of detail) which allows the computer to render a variable number of vertices, depending on the distance to the camera. There also has to be a resolution to the problem of rendering very distant objects. There are a couple of ways to go about this, one is to use a logarithmic or inverted depth buffer, which modifies the distribution to work well on large scales. The other method is to use three different cameras, one rendering close up, the next rendering midrange, and the final very distant objects.

I will use an agile based methodology for developing this project, so I can use iterative development to achieve many small goals in fast succession, leading to accomplishing the larger goals at the end.

Deliverables

An executable file running the graphical demonstration.

Constraints

The main constraint I have is time. I suspect there will be a considerable amount of tweaking to get the planet generation looking appealing. In addition to this, not all of the side goals I have set may not be possible to achieve within the time frame.

A secondary limitation may be hardware. The software will take up full usage of the CPU and GPU to be able to render large planets which could be the simulated size of Earth. Whether I will need to optimise or compromise on the engine significantly is something I will find out during development. Current hardware may be fast enough already, or the fact that my main emphasis is on achieving a level-of-detail system of rendering large planets, this may not be a problem.

Key Problems

The fundamental problem is rendering real scale planets. The amount of memory to store the Earth at 1km resolution, assuming we only store positions, would use ~5.7 terabytes of disk space. This obviously shows it's very unrealistic to store the geometry of a planet on a standard 1TB hard drive, let alone in the PC's RAM. My project will allow the player to roam the surface of the planet, needing much higher resolution than this. We will need some level of detail system in place to amend this.

A second problem is rendering extremely distant objects. If the player is on a planet and they can see to the horizon from eye level, we will need to render objects around 10 miles away; yet the sun is 91 million miles away. There are, of course, other planets millions of miles away which may need to be rendered too. In graphics programming, there needs to be a way to overcome this which aren't possible without requiring extra work.

The last problem is storing such large and small-scale values to work with each other. We need to store positions on planets, as well as positions of planets in the star system. Using the traditional approach will not give enough accuracy on a planetary surface if you're also storing it in the same way as planets are stored in the system.

System and Work Outline

I'll be using C++ and the DirectX11 SDK to produce the software. My first step is to create the Windows application and use DirectX to draw primitives to the screen. This also includes simple shaders. Next, I'll be organising the code into a more structured form in classes, which makes it easier to work on in the long run.

The subsequent stage is to start on a basic level of detail system. By creating and rendering one Quadtree which can increase or decrease in detail in relation to the camera, will form the basis of what we need to create a planet. Now, by arranging six of these Quadtrees into a cube, we can use some maths to visually transform it into a sphere.

Following on from the fundamental setup of a Quadtree sphere, we can start on the procedural generation. I'll be starting off with a fairly simple means of terrain generation, using 3D Simplex Noise. The problem with 2D noise is that it doesn't wrap very well around a sphere. There are methods to reduce the distortion but will never completely rid of it at the poles. However, using 3D noise will wrap allow it to wrap seamlessly. The sphere should now start to look like a randomly generated planet, though there'll be issues to fix. Depending on the resolution differences of neighbouring Quadtrees, there will be gaps at the boundaries. I will need to use some algorithm to fix this. There are two steps to fixing this. The first one is to find the Quadtrees neighbours, which seems trivial at first, however, that is not the case. As Quadtree nodes may be surrounded by much lower detail nodes or higher detail nodes, you need a Quadtree neighbour algorithm to find them all. The second step is to resolve the gap, methods include averaging every odd vertex, or removing every odd vertex, on nodes which have higher detail neighbours.

Once this is working I'll be experimenting with improved planet generation. Adding biomes is one which can be done by sampling more noise, then assigning a particular range of values to a biome type. One biome, in particular, I'll be adding is an ocean. Which can be rendered using shader techniques. I'd also like to add textures, trees and vegetation to the planet, which vary according to the biome. The textures will need to be interpolated between biomes too, and there are well-known solutions for this and shouldn't be a major problem. Adding trees will be as simple as

distributing them accordingly depending on the type of biome, the vegetation will just be grass, which can be done with shaders on the GPU.

Lastly, I wish to create multiple randomly generated planets and form a random star system. Simulating gravitational effects between the star, planets and moons. As a project on its own, this would be fairly simple to accomplish, however, when you take into account transitions from space to planets it suddenly creates an array of problems. There is not enough depth buffer precision to render these large and small scales in the same way. A solution is to use multiple depth buffers each with a different near and far plane. So when you render the world you decide which ones are drawn in which buffer depending on the distance to the camera. A second solution is to use a logarithmic depth buffer, which changes the distribution of the values in the buffer to give better resolution in the distance, as traditionally most of the precision is near the camera. Another problem which stems from the combining of large and small scales is the precisions of the positions of planets. For example, assuming the star is at the origin, you'll be dealing with values around ~9x10^9 at meter resolution. Unfortunately, single precision floats do not provide good accuracy at these ranges, producing very rapid camera jitter. A solution is to have the world move around the camera, meaning the camera is always at the origin. This gives an enormously better precision when moving around when the relative positions are much smaller. Instead, the far distant objects now have this reduced precision, but they're millions of miles away so to the player the errors are unnoticeable.

During development, I'm going to follow along with the lines of the class layout below. I have chosen to use multiple interfaces which allow me to add different implementations for parts of the project if I ever need to. The other reason is that it abstracts away the underlying functionality to minimise coupling.

- Body
    - IBody
        - IPlanet
        - IStar
- Render

    - DirectX Effect
    - IRenderable
    - IPlanetRenderer
    - IStarRenderer
    - Terrain
        - ITerrain
        - SphericalTerrain
        - Quadtree node
- Scene
    - Camera
    - IStarSystem
    - ILightSource

There are a significant amount of skills that I will be learning along the way as this a project I've never attempted before. Fortunately, the graphics module gave me a solid basis of knowledge to work from, therefore, I can focus on learning the important parts of the project e.g. Quadtrees, Procedural Generation, Large world rendering; eliminating having to learn DirectX beforehand. After the project, I hope to have learnt all of these topics enough to be able to apply them to other work afterwards.

Project Activities

Gantt chart is attached at the end of the document.

Risk Analysis

Risk	Severity	Likelihood	Action
Data loss	High	Low	Make sure the project is always pushed to GitHub, and store backups elsewhere.

Major hurdles taking too long to overcome	Medium	Likely	Continue on other activities which don’t depend on that hurdle when appropriate and take time to do additional research to help process the problem.
Other university modules needing extra work	Medium	Medium	Be time efficient, and only work on the most critical features of the project in the meantime.
Illness	Low	Low	Allow contingency time in the schedule, try to complete the project ahead of schedule.
Failure to create a working application by the final deadline	High	Low	Focus on the fundamentals first, and slowly build on top if progress is slow.

Options

I considered using a more cross-platform route of using OpenGL for graphics and the MinGW C++ compiler which would allow the software to run on Unix based systems as well as windows. I didn't choose this as I don't have as much experience in OpenGL as DirectX, and the demand to run it on Linux isn't high. This approach implies the only logical IDE to use is Microsoft Visual Studio, as it supports graphical

debugging with DirectX, which none other IDE's use, and no extra setup for the Microsoft C++ compiler. I also deduced that it wouldn't work well for mobile-based platforms as the graphics and processing will be very demanding. Consequently, the FPS will suffer.

I considered using a Waterfall approach to the project. However, due to the number of difficult hurdles I assume I'd have to overcome, it won't suffice for if I work linearly. Agile methodologies allow me to work on other features which may not depend on fundamentals.

Potential Ethical or Legal Issues

A potential legal issue to look out for is the patent surrounding Simplex Noise. The patent only covers image generation where it is shown on a display, whereas I will be using it for terrain generation. Regardless, there is still OpenSimplex noise, which has no patent attached.

The second thing I need to look out for is copyrights on textures. As I am going to use textured terrain I will be sourcing out textures online. I will only use textures which are royalty-free to avoid this.

Commercial Analysis

Factor name	**Description**	**Is this a cost or a benefit**	**Estimated Amount**	**Estimate of when paid**
Working hours	Assuming average salary of £20 and 350 hours of work	Benefit	£7,000	Last working day of every month
Visual Studio Professional standalone license	IDE used for development	Cost	£378	Pre-development
Microsoft Windows Pro	Operating system	Cost	£220	Pre-development

Graphic artist	A freelance graphic artist to create various textures for a few hours work	Cost	£200	End of development

Employability Contribution

This project will drastically improve my portfolio. Being able to put so many concepts together into one application would be a great achievement which would surely impress employers, especially games studios. The large-scale graphics techniques I'll use in my project seems to be a fairly unique concept. Whilst has been done a large number of times before, it doesn't seem to be a technique new games programmers would try out frequently. My aim is to produce a complete project and include all my goals. Showing that I can successfully integrate multiple concepts to complete it. Additionally, this the first piece of software I will write which is heavily concerned with speed. Therefore, I assume I will develop optimisation skills to achieve a high FPS despite the number of computations required to render dynamic LOD planets.

References

Schaal, J., 2017. Procedural Terrain Generation. A Case Study from the Game Industry. Game Dynamics, pp.133–150.

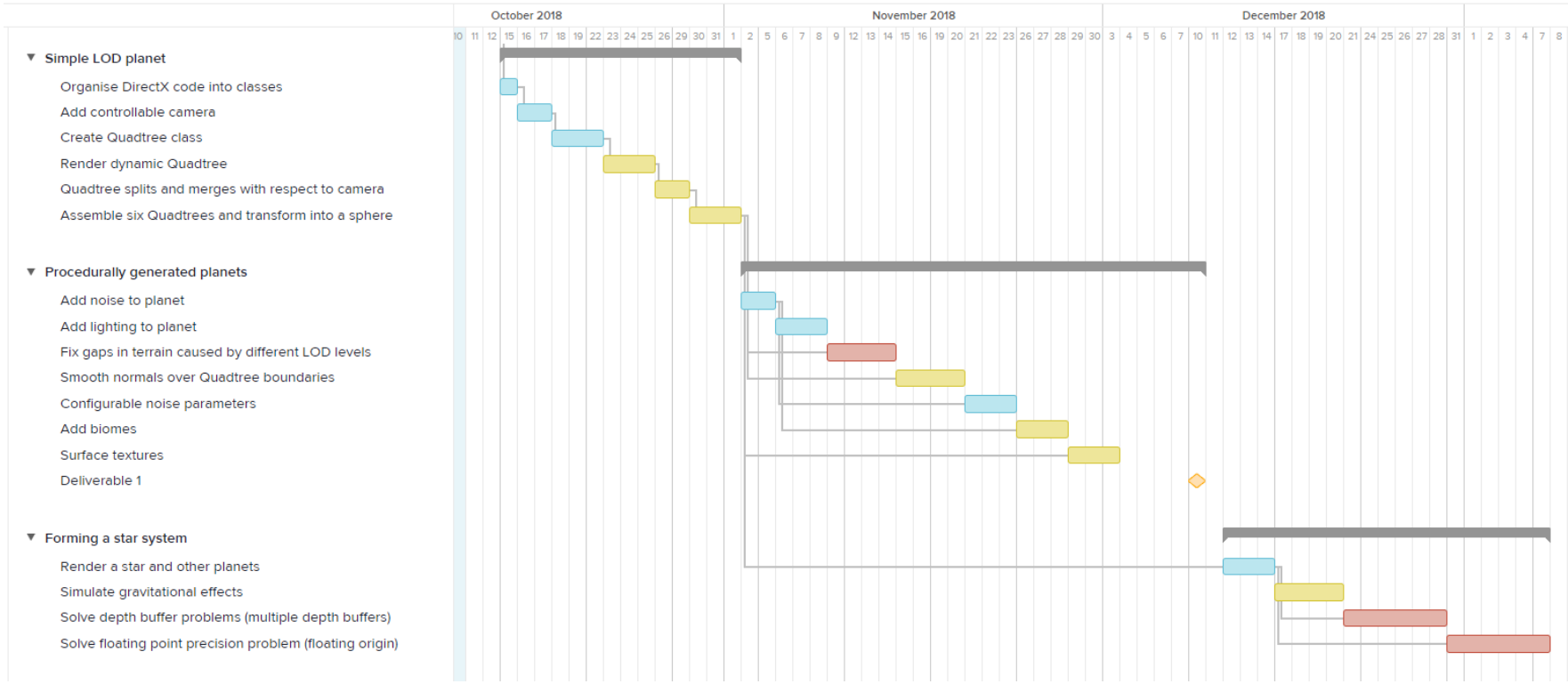
Aizawa, K. et al., 2008. Constant time neighbor finding in quadtrees: An experimental result. 2008 3rd International Symposium on Communications, Control and Signal Processing.

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

Nowell, P., 2005. Mapping a Cube to a Sphere. Math Proofs. Available at: http://mathproofs.blogspot.co.uk/2005/07/mapping-cube-to-sphere.html [Accessed April 22, 2018].

Ulrich, T., 2000. Continuous LOD Terrain Meshing Using Adaptive Quadtrees. Gamasutra. Available at: https://www.gamasutra.com/view/feature/131841/continuous_lod_terrain_meshing_.php?page=1 [Accessed April 22, 2018].

Pi, X. et al., 2006. Procedural Terrain Detail Based on Patch-LOD Algorithm. Technologies for E-Learning and Digital Entertainment Lecture Notes in Computer Science, pp.913–920.


October 2018
November 2018
December 2018
Simple LOD planet
Organise DirectX code into classes
Add controllable camera
Create Quadtree class
Render dynamic Quadtree
Quadtree splits and merges with respect to camera
Assemble six Quadtrees and transform into a sphere
Procedurally generated planets
Add noise to planet
Add lighting to planet
Fix gaps in terrain caused by different LOD levels
Smooth normals over Quadtree boundaries
Configurable noise parameters
Add biomes
Surface textures
Deliverable 1
Forming a star system
Render a star and other planets
Simulate gravitational effects
Solve depth buffer problems (multiple depth buffers)
Solve floating point precision problem (floating origin)

Appendix 3 – Finding neighbour nodes 1

```
std::vector<TerrainNode*> TerrainNode::GetSmallerNeighbours(TerrainNode
*neighbour, int dir) const
{
    std::vector<TerrainNode*> neighbours;
    std::queue<TerrainNode*> nodes;

    if (neighbour)
        nodes.push(neighbour);

    switch (dir) {
        case North: {
            while (nodes.size() > 0) {
                if (nodes.front()->IsLeaf())
                    neighbours.push_back(nodes.front());
                else {
                    nodes.push(nodes.front()->GetChild(SW));
                    nodes.push(nodes.front()->GetChild(SE));
                }

                nodes.pop();
            }

            break;
        }

        case South: {
        ...Other directions
    }

    return neighbours;
}
```

Appendix 4 – Finding neighbour nodes 2

```
TerrainNode *TerrainNode::GetGreaterThanOrEqualNeighbour(int dir) const
{
    auto parent = m_parent;
    auto self = this;

    switch (dir) {
        case North: {
            if (!parent) return nullptr;
            if (parent->GetChild(SW) == self) return parent->GetChild(NW);
            if (parent->GetChild(SE) == self) return parent->GetChild(NE);

           auto node = parent->GetGreaterThanOrEqualNeighbour(dir);

            if (!node || node->IsLeaf())
                return node;

            return (parent->GetChild(NW) == self) ? node->GetChild(SW) : node-
>GetChild(SE);

            break;
        }

        case South: {
        ... Other directions
    }

    return nullptr;
}
```