Automatically Generated Large Scale Alien Planets

BSc Computer Science (Game Engineering)

May 2014

Stephen Michael Cathcart (10270122)

Project Supervisor Dr. Graham Morgan

Word count: 16364

Abstract

Procedurally generated terrain has become increasing popular over the years for various reasons; it lowers development costs by not needing to employ as many artistic staff, it reduces the memory footprint of the game by not having to store prefabricated game content and it also allows huge new worlds to be built instantaneously.

This dissertation looks at the creation of a terrain generation tool to procedurally create large scale planets with an emphasis on ease of use. This will give the user the ability to create vast worlds at the click of a button whilst also providing them with a large array of options to be in full control over the landscape.

Declaration

"I declare that this dissertation represents my own work except where otherwise stated".

Acknowledgements

I would like to thank my project supervisor Graham Morgan and lecturer Gary Ushaw for their support throughout my degree. I would also like to thank Richard Davison who was always there to provide technical advice day or night.

Contents

Chapter 1: Introduction

1.1: Purpose

As gaming platforms continue to improve in terms of processing power and memory, users' expectations of game worlds become higher in terms of terrain size and complexity. When faced with creating terrain in modern games we normally have two options; either employ artistic staff to manually create the game world, or allow the game engine to procedurally generate the world.

Manually creating the world is both time consuming and expensive; more artists are needed throughout the development period and any changes needed to content are again time consuming. It is also expensive on memory from having to store the prefabricated game content.

Procedurally generated terrain has become increasing popular over the years for various reasons; it lowers development costs by not needing to employ as many artistic staff, it reduces the memory footprint of the game by not having to store prefabricated game content and it also allows huge new worlds to be built instantaneously.

However, there are still many problems with this approach. Not only can this mean game designers have very little control over the levels, it can also be incredibly difficult to get the world generating 'realistic' terrain and not just large bland zones. John Carmack, Technical Director at id Software, along with many other developers believe that "humans can only be engaged by content that was created by other humans" (Cepero M, 2011).

This dissertation looks at the creation of a terrain generation tool to create large planets with an emphasis on ease of use. This will give the user the ability to create vast worlds at the click of a button whilst also providing them with a large array of options to be in full control over the landscape.

1.2: Hypothesis

Based on the current situation of terrain generation described above, the hypothesis is that the creation of a terrain generation tool that is intuitive to use as well as providing a high level of control over the terrain will increase the ease of generating, modifying and exporting large realistic planets.

1.3: Aims and objectives

An aim has been set to provide an answer to the above hypothesis which is:

"To create a tool which allows the user to easily generate, modify and export realistic large scale alien planets that are customisable to the user's needs."

In order to achieve this aim, three objectives have been specified:

- Investigate and implement modern algorithms for generating random terrain, optimising the terrain and generating realistic atmospheres.
  - *As this is a popular area of research, there are many proposed solutions around. This objective is to condense these down and choose which methods will be most suitable to the project.*

- Allow the user to be able to easily manipulate the generated planet through a GUI in real time.
  - *As the project emphasises a high level of usability, a GUI will need to be implemented that will interface with the core program allowing the user maximum control.*

- Evaluate the tool based on performance, aesthetics and ease of use by analysing feedback from both application and user testing.
  - *Evaluating the performance of the program is simple enough however, evaluating the usability and aesthetics is quite subjective and difficult to test. This is the reason why user feedback will be analysed.*

1.4: Dissertation structure

Chapter 1: Introduction

This chapter provides an introduction to the dissertation, outlining the purpose and reasoning behind doing this project. Based on the hypothesis, a main aim and three objectives are set out in order to achieve this.

Chapter 2: Research

Briefly explains the history of terrain generation and outlines a research strategy to gather information relevant to the project. This includes not only researching terrain generating algorithms and rendering techniques, but also additional software libraries to aid in development.

Chapter 3: Methodology

Provides a detailed account of how and why the project was implemented such as time management plans, outlining of the system architecture and the user interface design. This chapter also highlights key technologies used to achieve the projects overall aim.

Chapter 4: Testing & Results

This chapter provides a detailed analysis of the finished project which includes all application, performance and user testing. The results also highlight any issues that arose when testing.

Chapter 5: Evaluation

The overall project is evaluated here, the time management plan is analysed as well as the satisfaction of the aim, objectives, hypothesis and both functional and non-functional requirements. Any remaining work is also evaluated here.

Chapter 6: Conclusion

An overview of personal development on this project i.e. what has been learnt, what could have been done better? The continuation of the project will also be discussed.

Chapter 2: Research

2.1: Overview

This chapter will focus on researching procedural generation techniques as well as advanced rendering techniques in the context of terrain. Where applicable the advantages and disadvantages of each technique will also be outlined. As the area of terrain generation is quite large, with each area such as lighting effects potentially being a dissertation themselves, this project will instead attempt to implement a wide range of these techniques at a basic level as a proof of concept. The project will be designed with extensibility in mind to ease the job of implementing further advanced features in any future work.

2.2: Research strategy

As a lot of what will be researched below is relatively new in the field of computer graphics, the main resource that will be used are up to date websites such as Gamasutra, which contain very active forums relating to terrain generation techniques. I have also found that the best resource to fully get to grips with OpenGL programming still remains to be the OpenGL SuperBible book which goes into great depth about various rendering techniques that will be more than enough for this project. To further strengthen my research, other research papers in the area of terrain generation will also be analysed as well as looking at terrain generation in modern games.

2.3: Background research

2.3.1: Terrain generation techniques

Heightmaps/Heightfields

The basis of any terrain generation algorithm is in the generation of a heightmap. A heightmap is essentially just a two-dimensional grid of elevation values (Smelik R M, 2009); with dark and light values representing low and high elevations respectively. Without a heightmap, every mountain, rock and pebble must be hand crafted which just isn't a feasible option for large terrains (Archer T, 2009). The oldest techniques used to generate these heightmaps were subdivision based methods; the heightmap is iteratively subdivided adding controlled randomness on each iteration (Smelik R M, 2009). However, what are used more commonly now are fractal noise generators. These types of algorithms work by sampling and interpolating across points in the grid, with several layers of noise then added together to give much more realistic, mountain like terrain. As you can imagine from looking at the heightmap, shown below (see fig. 1), a disadvantage of using basic heightmaps are that complex terrain cannot be made from them such as cliff overhangs or caves; we are only adjusting points along the y-axis (see fig. 2).

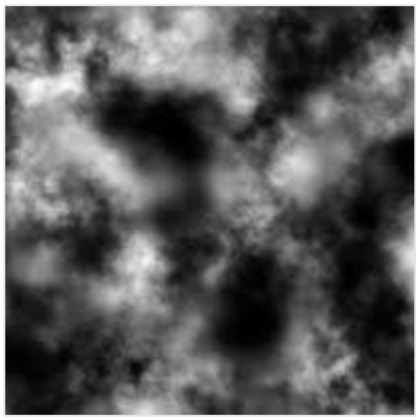

Fig. 1 *Heightmap* (2006)

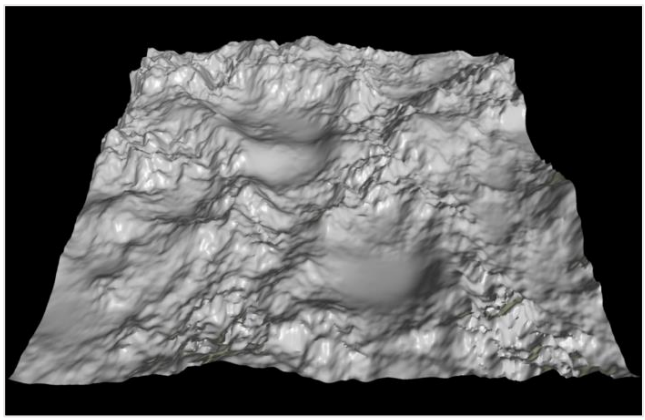

Fig. 2 *Heightmap Rendered* (2006)

A recent method which calculates the terrain on the GPU provides a more elaborate structure with different material layers that supports rocks, arches, overhangs and caves (Peytavie et al, 2009). The resulting terrains that are generated using this method are visually very plausible and natural (see fig. 3).

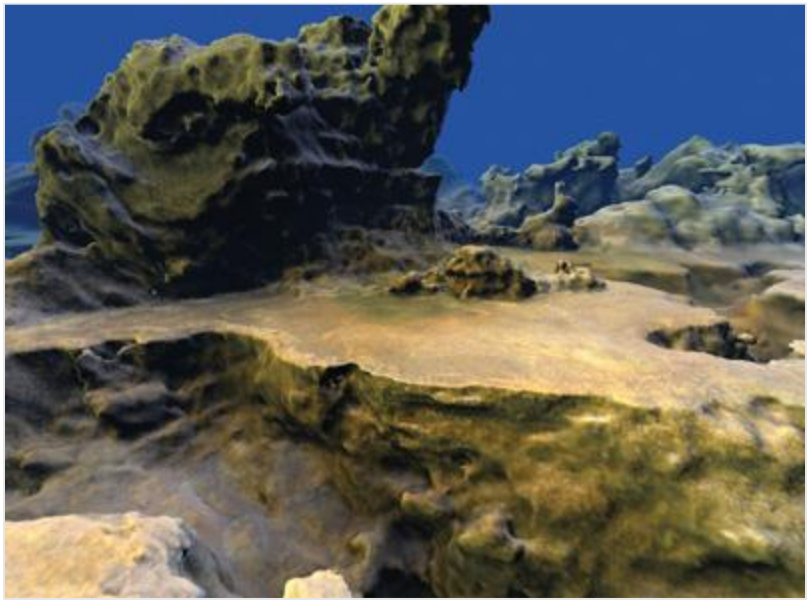

Fig. 3 *Terrain Created Entirely on the GPU* (2006)

Various post-processing filters can also be applied to the generated heightmap to further enhance their realism. An example of this is the application of a smoothing filter to avoid sharp mountain peaks. We can also apply different types of erosion to the heightmap to further enhance the realism of the terrain (Smelik R M, 2009). These techniques will be discussed in more detail below.

Common Noise Algorithms

As discussed above, we generate terrain by displacing each vertex along the y-axis by an amount based on the value of the heightmap at that particular point. So how do we create the heightmap?

Special algorithms called noise generators are used to produce these heightmaps (Archer T, 2009). A common noise algorithm currently used in industry is the Diamond-Square algorithm, which is an improvement on the Midpoint-Displacement algorithm. The reason why the Diamond-Square method is better is due to the elimination of square-shaped artefacts that could appear in the latter method; this happened due to using two points to calculate some values whilst using four points to calculate others (Archer T, 2009). The algorithm is visually shown below (see fig. 4):

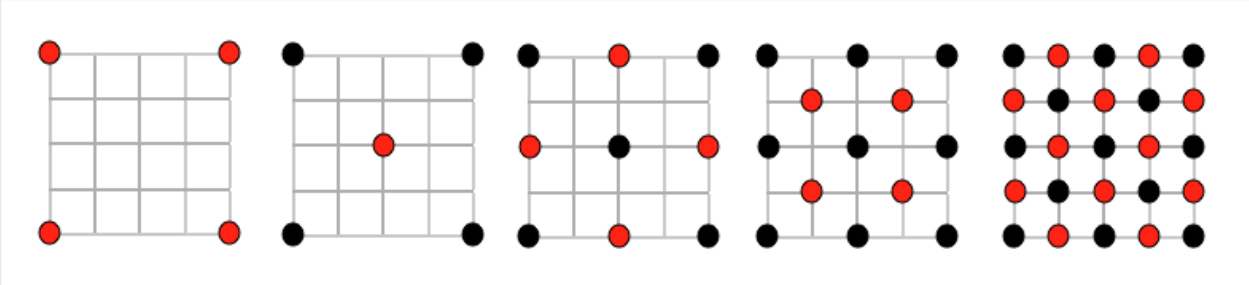

Fig. 4 *Diamond Square Algorithm* (2010)

The first step is to set the four corners to some starting value, these could be random or pre-set, it really doesn't matter at this stage. The next step is the diamond step, here we take the four square shaped initial points, average them and add a random value to this average which creates a diamond shape. Next is the square step which averages the four corners of the diamonds created during the previous step. The centre values can now become the corners of the next square step (Beard D, 2010). Both of the stages are then repeated until every point in the grid has been assigned a value (Beard D, 2010).

While the Diamond-Square method generates good results, a more common and visually appealing noise algorithm is Perlin noise. This is the standard in the noise industry for many reasons: it is very fast, needs little memory and produces high quality noise (Archer T, 2009). It is however, more complicated to implement than the previous algorithm. The most apparent feature over the Diamond-Square algorithm is that it generates coherent noise over a space. Coherent meaning that for any two points in the space, the value of the noise function changes smoothly as you interpolate over the surface so there are no discontinues (Zucker M, 2001).

When describing the function mathematically it can be quite daunting however, the basic theory behind it is quite easy. Essentially it is a two-step process; the first step we generate a number of arrays containing smooth noise with each array called an octave. Each octave has a different level of smoothness. Step two blends these octaves together. The image below is an example of this process (see fig. 5):

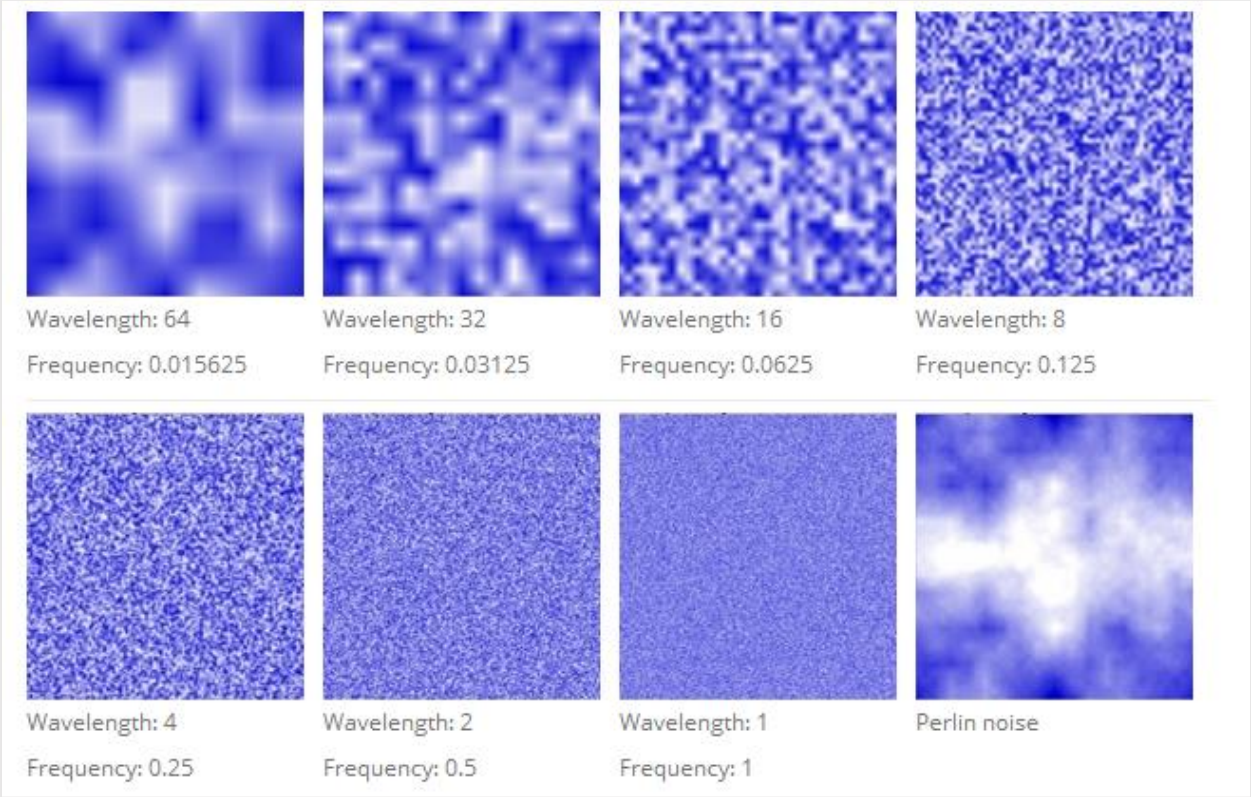

Fig. 5 *Perlin Noise* (2012)

It is also worth noting that in 2001, Ken Perlin created simplex noise, which is similar but uses a simpler space-filling grid, alleviating some problems with Perlin "classic noise"; among them, computational complexity (Perlin K, 1999). This algorithm is currently starting to overtake Perlin noise in industry with regards to terrain generation.

Thermal and Hydraulic Erosion

Erosion techniques can be applied to the generated heightmap afterwards to give an extra layer of realism and are worth mentioning due to the dissertations aim of creating realistic planets. Firstly, thermal erosion models the effect of gravity eroding cliffs that are too steep (Archer T, 2009). We see this on hills where soil and pebbles build up at the base from rolling down the hill. The algorithm for this is surprising simple and very fast. It works by calculating the difference in height from one plane point and its neighbouring point. If that distance is greater than a specified value, then we remove a random amount of the terrain

(soil, pebbles etc.) and deposit it onto the lower plane. This approach can then be repeated until the desired amount of erosion is reached. A downside of this algorithm is that the parameters need to be constantly tweaked to get the desired result; it can initially lead to not very convincing results (Archer T, 2009).

Hydraulic erosion models the effect of water on the terrain such as you see during a rainstorm whereby soil deposits are washed up, and then when the water evaporates it leaves these soil deposits behind. The level of realism this technique provides is fantastic however, it does have its drawbacks; it is a very slow algorithm to calculate as well as needing much iteration before producing believable results. On top of that it also requires a large amount of memory (Archer T, 2009). More memory is needed as the algorithm requires the use of a water table which is an array the size of the heightmap which holds the amount of water each pixel holds, as well as a sediment table used to track the amount of sediment in the water. We then specify various variables such as the rain amount, solubility, evaporation and capacity. By adjusting these variables we are able to change how the erosion behaves i.e. higher evaporation levels move soil shorter distances while lower evaporation levels create pools of water. As said previously, this is a slow algorithm as the erosion process runs on every pixel each iteration however, the results are great (see fig. 6). This uses three times as much memory as thermal erosion (Archer T, 2009).

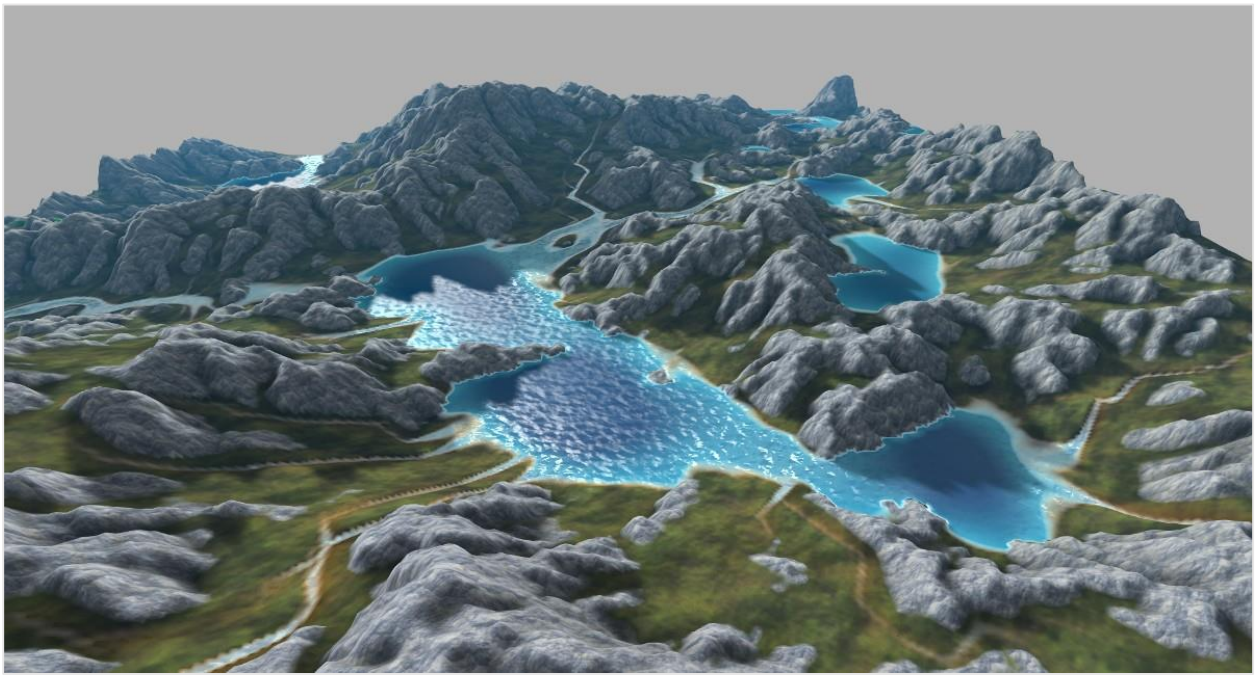

Fig. 6 *Eroded* (2011)

Smoothing the terrain

A very easy technique to implement which increases the realism of the terrain is the application of a smoothing filter to the heightmap. A common issue that arises when generating heightmaps using algorithms such as Diamond-Square is the appearance of very sharp mountain peaks. To elevate this issue we can do some post-processing on the heightmap using an average box filter which will produce smooth hills. This works by scanning a square array (usually to the power of 2 + 1 i.e. 3x3 or 5x5) along each row of the heightmap; taking the average of the surrounding pixels to give to the center pixel (Beard D, 2010). While it is a simple algorithm, it results in much more believable terrain (see fig. 7-10).

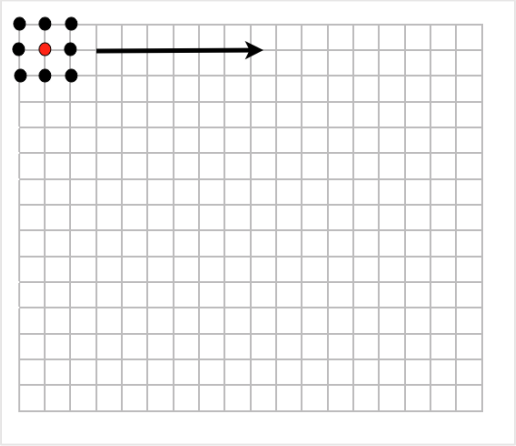

Fig. 7 *3x3 Box Filter* (2011)

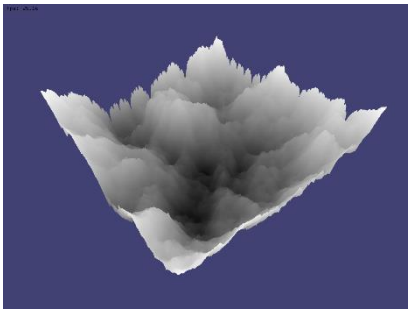

Fig. 8 *No smoothing* (2011)

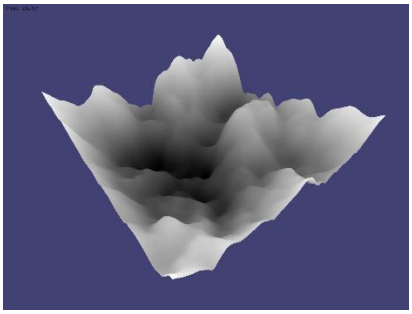

Fig. 9 *Smoothed Terrain* (2011)

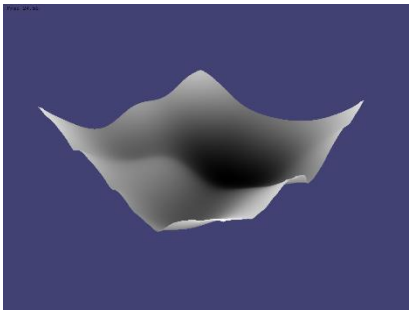

Fig. 10 *Filtering done 50x* (2011)

Real time dynamic level of detail terrain rendering with ROAM

Common types of algorithms used in the area of terrain generation are level of detail (LOD) algorithms. Graphics programmers are constantly balancing the scenes detail (primitive count) with acceptable frame rates. These algorithms are used to efficiently render terrain by ensuring that the number of primitives (often triangles) used to represent the mesh are kept as close to an optimum level as possible (White M,

2008). These types of algorithms basically determine which parts of a landscape need more detail to look correct (see fig. 11). This area is vast and could take up a dissertation on its own so a general overview is given to provide the basics. A simple and commonly used LOD algorithm is the Real-Time Optimally Adapting Mesh algorithm (Turner B, 2003) which optimizes the terrain mesh. This works by defining the scene as a hierarchal binary tree structure of triangles, often called a binary triangle tree. Each node represents a triangle that is a lower detail version of its two children nodes. Leaf nodes represent the highest LOD triangles whilst the root represents the lowest. From here it is a recursive task to traverse the binary triangle tree and decide whether to "tag" the relevant triangle or to step a level deeper into the tree (Turner B, 2003). Normally the virtual camera in the scene (which is often the player) is used as the focus point in the algorithm, so the scene will be of a higher LOD closer to the player whereas meshes in the distance such as hills/mountains will be of lower LOD.

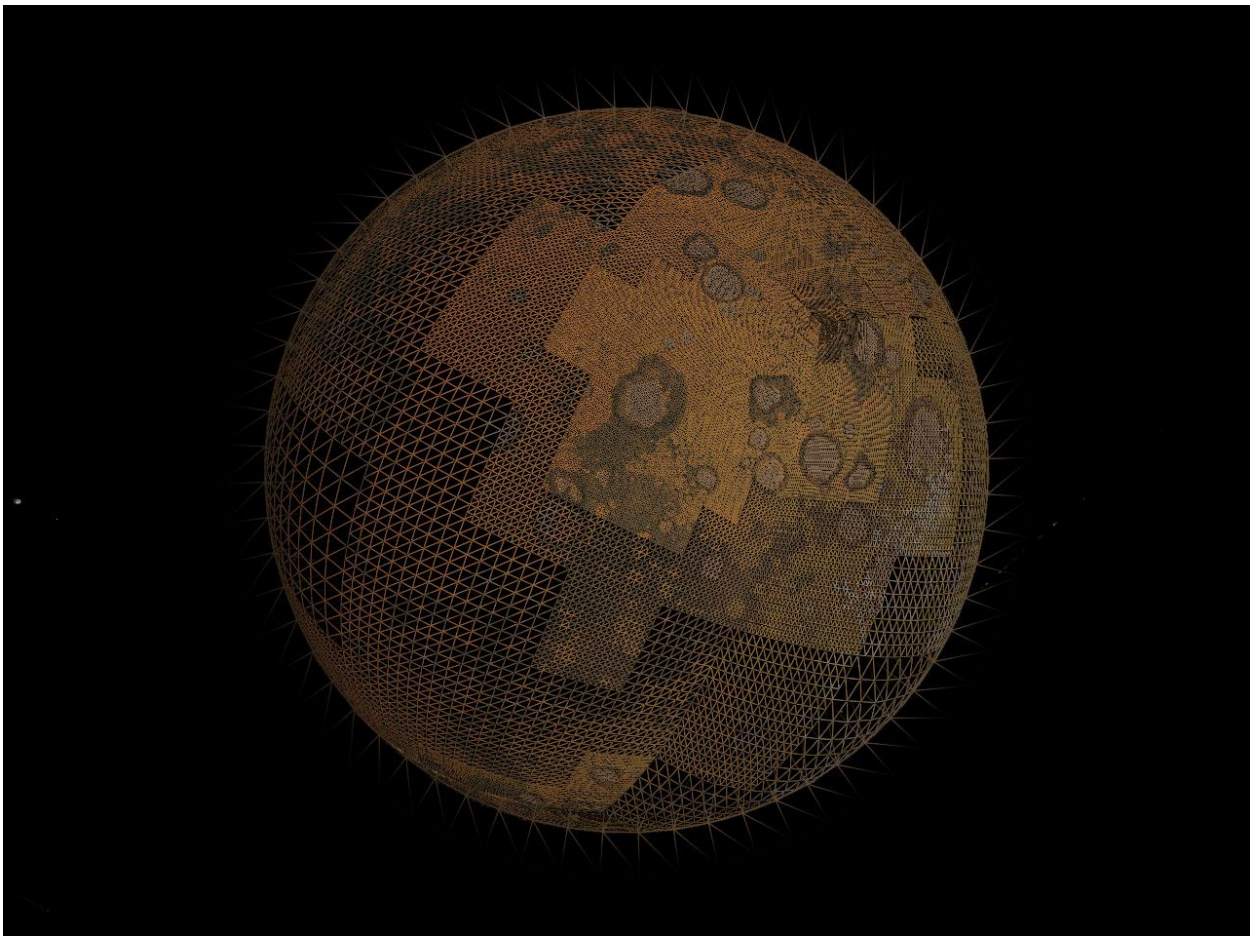

Fig. 11 *ROAM System* (2009)

Implementing this algorithm will drastically reduce the number of triangles in the terrain to be rendered which will improve the efficiency of the program.

Generating the sphere

The above algorithms and techniques are crucial to any terrain generation software however, so far we have only discussed terrain which is flat. The purpose of this project is to generate realistic worlds. To do this we need to programmatically create a sphere mesh to implement these algorithms on. As we need a fine level of control over the sphere, especially when it comes to manipulating and texturing the primitives, using the default OpenGL *gluSphere* wouldn't be advisable. Another approach is to use an icosahedron (a 20 sided triangular faced shape), tessellate it and push each new vertex outwards along its normal to form a sphere (Rideout P, 2010). This approach also has the advantages of all primitives being evenly separated so there is no bunching up at the poles; which can be a common problem with spheres. The below image (see fig. 12) shows different tessellation levels being applied to an icosahedron:

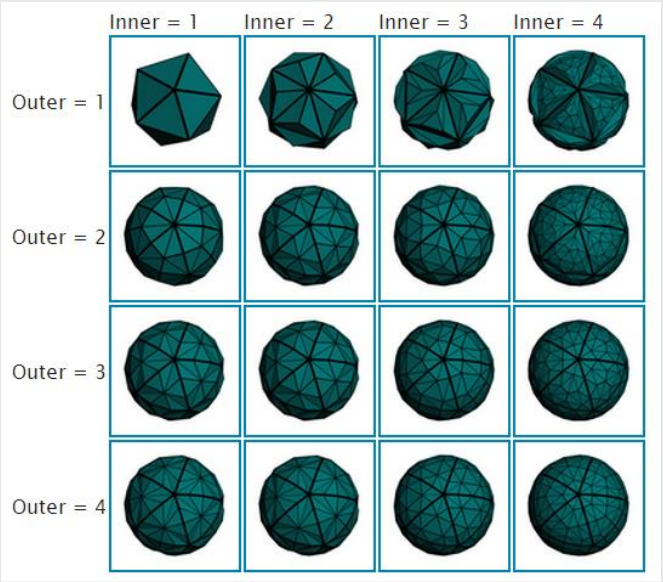

Fig. 12 *Icosahedron Tessellation* (2009)

As shown, when the tessellation levels get higher than four the icosahedron takes on the shape of a perfect sphere. Allowing the user to control the tessellation levels would be a great feature from a usability point of view and something that should be incorporated into the graphical user interface.

2.3.2: Rendering techniques

Texturing the terrain

Texturing the generated planet poses two problems: how do we map the rectangular heightmap image around the generated sphere and how do we give different parts of the planet a different texture (such as a rocky texture for higher regions and a grass texture for lower regions)? The process of fitting the image onto the mesh is called texture mapping and is simpler than it first looks. What we need are the normal vectors for the vertices; these will usually be pre-calculated anyways for the purpose of lighting (Fernandes, A R, 2012). The UV texture coordinates are then generated based on the angle of the surface at each point (Dunlop, R, 2005). The u-coordinate is calculated from the x-coordinate of the normal, which will vary from -1.0 to 1.0, representing the longitude of a corresponding point on a sphere. The v-coordinate is calculated in the same way however, it will be calculated using the y-coordinate of the normal and represents the latitude of the point. These are calculated using the below formula (see fig. 13-14):

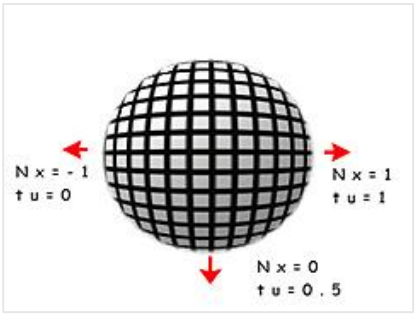

Fig. 13 *Sphere* U (2005)

tu = asin(Nx) / PI + 0.5

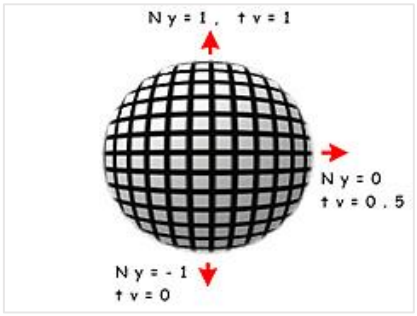

Fig. 14 *Sphere V* (2005)

tv = asin(Ny) / PI + 0.5

The result is that the coordinates are non-linear and will therefore tend to stretch the texture around the y and z poles whilst compressing it around the x poles (Dunlop, R, 2005). This shouldn't pose too much of a problem as the texture can be manipulated afterwards This technique will be very helpful when deciding upon how much each vertex should be pushed out when comparing it against the heightmap at that particular point.

Finally, a common technique to apply different textures to different regions of the mesh is by using ground types (Dexter J, 2005). To do this we specify a certain amount of ground types such as grass, rock and snow; each of which will have its own texture associated with it (see fig. 15). Now we just need to let the program decided on which ground type to choose for each vertex of the terrain. A simple approach, using the above

ground types as an example, is to let the top third of the vertices to be snow, the middle third to be rocks and the lowest third to be grass (Ebert, D S, 2003). The problem with this is that the boundaries between ground types are sharp giving unrealistic results. To elevate this issue we can simply blend boundary textures together to give a more pleasant result (see fig. 16).

Fig. 15 *Ground Types* (2005)

Fig. 16 Ground Types Blended (2005)

2.3.3: Additional libraries to aid development

Rendering to the window

OpenGL is an application programming interface for rendering 2D and 3D vector graphics and is typically used to interact with the graphics processing unit. It is ideal for this project as it is cross platform; meaning users will be able to use the software on different platforms. One of the most important features of OpenGL is portability however, OpenGL alone isn't enough to create complete programs; you need a window and a rendering context. On top of this you have no choice but to write operating system specific code to handle these things (Gomila L, 2013). This is where the Simple and Fast Multimedia Library (SFML) comes into play. SFML is a portable, simple interface to various modules that eases multimedia and game programming (Gomila L, 2013). It provides an OpenGL ready window, rendering context and handles user input among many other things such as audio and networking. For the basis of this project it is a perfect foundation to begin creating the terrain generation software.

Creation of a GUI

As the terrain generation tool needs to be able to let the user manipulate the planet a graphical user interface (GUI) needs to be implemented. To do this a GUI library needs to be integrated into the project. The Simple and Fast Graphical User Interface (SFGUI) is a C++ GUI library for SFML that will be used throughout this project. SFGUI provides a rich set of widgets and is highly customizable in its looks (Iteem, 2013). The library has been designed with flexibility and extensibility in mind and provides a modern and clean C++ API (Iteem, 2013).

2.3.4: Terrain generation in modern games

The Elder Scrolls IV: Oblivion

As discussed in the introduction, solely relying on procedural generation for terrain is just not a feasible solution at present when it comes down to creating immersive modern video games. What normally happens is developers initially use some sort of terrain generation technique and then go in afterwards by hand to tweak the terrain and add prefabricated content in. This significantly lowers development times and eases the artists' job by not having to spend long periods of time creating complex mountains, for example. Game producer Gavin Carter who worked on The Elder Scrolls IV: Oblivion (see fig. 13-14) was interviewed about their use of terrain generation in the game, confirming the use of both techniques:

"Our research into procedural generation systems was primarily something we did for efficiency purposes. The time required to generate the landscape for Oblivion was significantly lower than for Morrowind. And whereas Morrowind featured primarily smoothed over hills due to being done by hand, our programmers incorporated erosion algorithms into our landscape generation, giving us awesome craggy mountain vistas that would take us forever to do by hand." (RPGamer, 2006).

Fig. 17 *Oblivion Landscape* (2005)

2.4: Summary

The techniques that have been researched above provide a solid foundation in the area of terrain generation. A wide range of techniques and third-party libraries have been researched such as basic noise algorithms, texturing of the terrain and the use of a integrating a GUI library to aid development. The above research will drastically simplify the implementation stage and make it easier to estimate the time needed for specific tasks.

Chapter3: Methodology

3.1: Overview

This chapter provides a detailed account of how and why the project was implemented such as time management plans, outlining of the system architecture and the user interface design. It also explores key technologies used to achieve the projects overall aim and objectives.

3.2: Planning

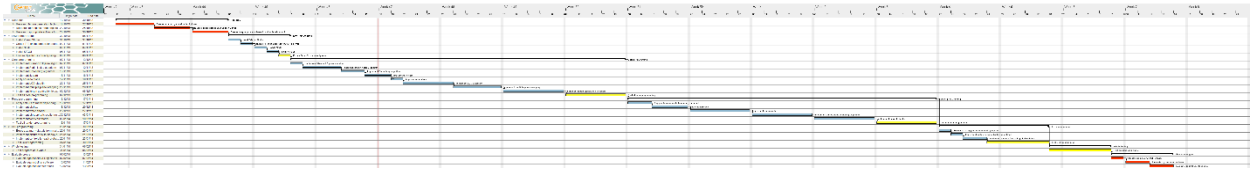

Fig. 18 *Gantt Chart* (2014) (For larger view see https://i.imgur.com/UTXYJI4.png)

The Gantt chart (see fig. 18) is separated into various stages due to the software development methodology chosen (incremental development). The reason why this methodology has been chosen will be explained in a later section. It is also worth pointing out for now that red bars in the Gantt chart indicate research or evaluation tasks, blue bars indicate development tasks and yellow bars indicate testing tasks.

The first couple of weeks were dedicated to researching the problem and collecting as much reference material and example source code as possible. This covered topics such as modern terrain generation techniques, modern rendering techniques and appropriate software which would aid in development. The goal was to try and not reinvent the wheel as much as possible which is the reason why the project takes advantage of various APIs such as SFML to handle window creation, user input etc. Using as much external software as possible reduces the risk that the project will run over as they will help speed up development times.

Next is the environment setup stage. This was some time put aside to setup and install software that would be required for the project. This included installing the IDE, setting up source control (which reduces the risk of losing source code), setting up various APIs such as SFML/SFGUI and making sure OpenGL was working correctly.

The next three development stages are split into different categories. These are the core programming, render programming and GUI programming stages with each stage designating appropriate testing time at the end. The core programming stage mainly concentrates with developing the basic skeleton of the project

such as implementing various noise algorithms, creating a camera to move around the world, implementing LOD etc. The render programming stage is mainly focused with implementing various shaders to texture the terrain, apply water effects, apply atmospheric scattering etc. Finally the GUI programming stage concentrates on implementing various control panels that will allow the user to manipulate the terrain with ease and will also allow them to export and load the terrain.

The final stage is the evaluation stage which is used to evaluate the program against the original aim and objectives and against user feedback. It is also worth pointing out that the timescales for each task has been decided by a combination of past experience and also from feedback given during relevant University graphics modules.

3.3: Elicitation

To ensure that the main aim and objectives are met it is necessary to outline some functional and non-functional requirements of the terrain generation system. A functional requirement will describe what the system should do, while the non-functional requirements place constraints on how the system will do so.

3.3.1: Functional requirements

Basic manipulation of the terrain

The system must allow users at least a basic level of manipulation over the generated terrain. This includes resizing, texturing and shaping the terrain. To accomplish this, the system will present the user with a GUI featuring various widgets to control these.

Importing and Exporting

The system must allow the user to import and export their generated planets. The GUI will include buttons to help export the generated terrain by storing the planets parameters in a JSON file, which can then be imported in the future.

Basic control over rendering

As the system requires the generation of realistic terrain, the system must allow the user to alter features such as light intensity, light colour, shading techniques etc. Again, the GUI will include widgets to satisfy this requirement.

3.3.2: Non-functional requirements

Efficiency

When randomly generating the terrain in real-time the user should only experience minimal delay. The application itself should be optimized from a memory and processing point of view so that it may target a wider range of computers.

Effectiveness

The generated terrain should be convincing enough to keep the user interested in using the system again. To achieve this, algorithms used to generate the heightmap should be complex enough to spawn a wide range of different patterns.

Fault tolerance

In case the system should crash, the user should experience minimal distress. To achieve this, the system should constantly take backups of the current settings so the user may import them after failure and pick up where they left off.

Portability

The system should be able to run across multiple platforms such as Windows, Mac and Linux to attract a wider range of users. To achieve this, the technologies decided upon prior to development should all be cross-platform.

Testability

Every feature in the system should be easy to test which will reduce the amount of bugs when released. To achieve this, the development of the system will take a modular approach; each feature shouldn't rely on another feature and can potentially ship on its own.

3.4: System architecture

To ease development, the system architecture has been outlined. This includes the main features of the system as well as how they related to each other. To help visualize the design of the system a UML class diagram has been created (see fig. 19). UML is a graphical language for visualizing, specifying, constructing and documenting the artefacts of a software intensive system (Sparx Systems, 2000). The tool chosen to create the diagram is called WhiteStarUML, it is an open source UML tool and is a fork of the now abandoned StarUML project (Janszpilewski, 2014).

3.4.1: UML Class Diagram

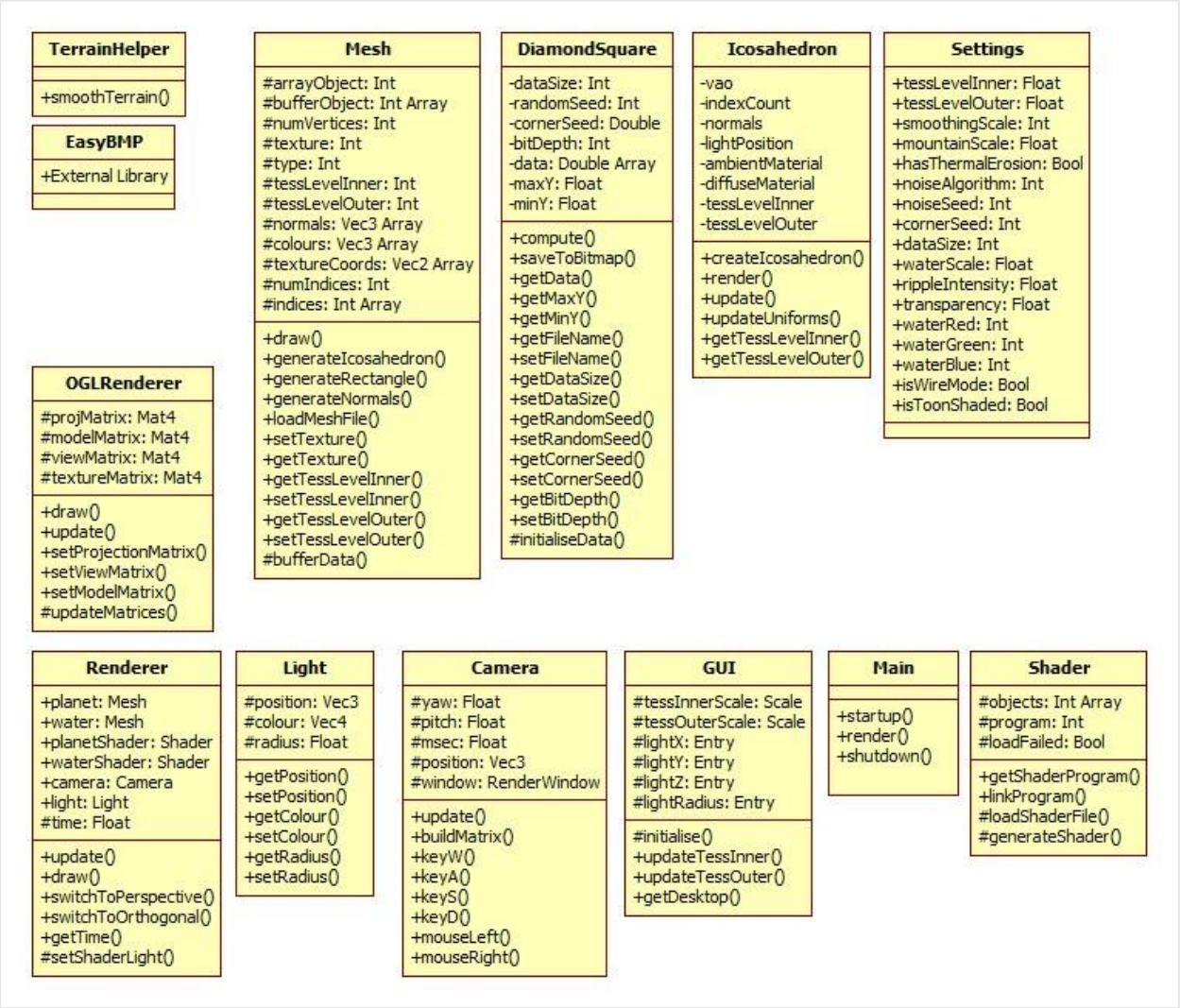

Fig. 19 *UML Class Diagram* (2014)

3.4.2: Relationship between classes

The overall terrain generation tool was broken down as much as possible. The responsibility of rendering the scene falls inside the *OGLRenderer* and *Renderer* classes. These classes hold the planet, water and atmosphere *Mesh* data as well as containing the functionality to switch from and to perspective and orthogonal views (this comes in useful when drawing the GUI on top of the 3D scene). They also contain two other crucial objects; a Camera and Light class. The Light class contains variables to alter its colour, position and radius whilst the Camera class contains the position variable of the camera (the view that the user will have over the scene).

The GUI will be initialised and updated separately from the rest of the scene which is why a GUI class has been created; this class is responsible for initialising all of the widgets and linking them to their respective callback functions.

The most important class here is the *DiamondSquare* class. This contains an array of elevation values which are computed using the Diamond Square algorithm that was covered in the research chapter. It also contains the functionality to save the generated heightmap to a bitmap file.

The *TerrainHelper* class contains functions which manipulate the data generated from the above *DiamondSquare* class; at present the only function that exists is a simple smoothing function.

As the program needs an initial icosahedron to be present for the later tessellation to take place, an *Icosahedron* class has been made which contains the data of a readymade icosahedron. The creation function is called once at the start of the program and leaves the rest of the shapes manipulation to the tessellation shaders. These shaders are created at runtime through the *Shader* class; this is responsible for compiling, linking and storing the shader data.

Finally, to simplify the program a *Settings* class has been created which contains various global static session data. This data is used to initialise all of the GUI widgets i.e. the *tessLevelOuter* float is used to initialise the tessellation outer level widget. This makes it easier when saving and loading projects as we only need to update this one class.

3.5: Software development model

3.5.1: Overview

A software development methodology in software engineering is a framework that is used to structure, plan and control the process of developing an information system (IT Info. 2005). The traditional methodology still used by many developers today is the Waterfall model. The disadvantages of this model however, is that no working software is produced until late during the projects life cycle. It is also a poor model for long term projects and is not suitable for projects where the requirements have a high chance of changing. Also once the system is in its testing stage, it can be very difficult to go back and alter the systems design. In terms of this project, using the waterfall model would not be the best way to go as there is a large amount of ambiguity in the projects' design; meaning there is a high chance the system will change throughout the development process. Another common methodology is the incremental development approach, which was mentioned earlier in the planning stage. Below is an analysis of this method.

3.5.2: Model analysis

Incremental Development Methodology

This approach is essentially a set of mini waterfall stages. It breaks the project down into a set of milestones that upon completion, results in a piece of software that could potentially be shipped (see fig. 20). Each iteration adds more functionality to the system. At the end of each cycle there is room to test the new features and change the functionality based on client feedback (IT Info. 2005).

As the project is quite programming intensive and has a high level of ambiguity, this approach will be used for various reasons; it allows larger tasks to be broken down into smaller milestones that can be tested thoroughly during development, this reduces the risk of not having enough time at the end for testing. It will also be beneficial when developing the GUI as this can then be developed in stages as each piece of the program is created. Each main stage also has a testing period at the end which has been made long enough to allow changes in the plan if needed, this is the contingency plan.

This model however, is still without its flaws. A common problem with incremental methodologies is the risk of a project over running; if a certain milestone takes up longer than was expected there may not be enough time to complete the project as a whole. As stated above, the contingency plan for this is to plan for longer testing periods which can be used up for situations like these. Finally, another common problem with these methodologies is that when additional functionality is added to the system, problems may arise related to system architecture which was not evident in earlier prototypes. The only way to reduce the risk of this happening is to briefly plan the out the whole system beforehand and to keep the code very modular during development with little dependencies between classes.

Fig. 20 *Waterfall vs Iterative* (2009)

3.6: Testing strategy

The testing of the application will be split into three main areas: application, performance and user testing. The application testing will cover whether the implemented features are working correctly without any undefined behaviour. For example, when resizing the window does the application scale correctly? Or when changing the tessellation levels in the GUI does the planet respond in the right manner? Due to the software methodology chosen above, the actual application testing will be fairly straight forward; each development cycle includes a testing period at the end which will mainly be used to catch these sorts of errors. As there will be plenty of testing time to move about, the prediction is that there will be minimal application errors in the final solution.

The project will also be tested based on its performance. By performance we are talking about the systems memory and CPU usage. These will most likely be the issues that will cause problems at the end of the project as they can be difficult to spot early on. To reduce the risk of producing badly performing software the use of a profiling tool will be used throughout the development process to aid in tracking down memory leaks, inefficient allocation of memory and CPU spikes.

The last testing area that the project will be benchmarked against is user testing. By user testing we mean the process of a user giving us qualitative and quantitate feedback related to the applications ease of use and aesthetics. The reasoning behind this is that by this point all crashing bugs and performance issues should already have been caught by the other testing areas. By getting this feedback we can then go on to evaluate the finished project against the original objectives set out.

By following these testing strategies and allowing time to modify the system afterwards the outcome should be a very stable and aesthetically pleasing solution for the user.

3.7: Tools and technologies used

3.7.1: Overview

Below provides a detailed account of all technologies used to implement the system. Some of the technologies that have been decided upon were discussed earlier in the research chapter and thus, will only be briefly recapped.

3.7.2: C++

C++ will be the programming language of choice to develop the terrain generation tool for various reasons. As it is a lower level language than Java or C# it allows us to have greater control over memory management

and general system performance. There are also a large amount of open source libraries developed in C++, such as SFGUI and SFML, which will be very useful when developing the terrain generation tool; by using C++ these will integrate easily into the solution. These libraries are covered in more detail below as well as a brief discussion listing the advantages of using them.

3.7.2: SFML

When researching for an appropriate OpenGL context we discussed the Simple and Fast Multimedia Library (SFML). We learnt that SFML is a portable simple interface to various modules that eases multimedia and game programming providing an OpenGL ready window, rendering context and handles user input among many other things (Gomila L, 2013). Let's see how easy some of these modules are to use in practice. The first thing we need to start rendering with OpenGL is to set up a rendering window and initiate the main loop, this is simple in SFML (see fig. 21):

```
int main()
{
    // Create the main rendering window
    sf::RenderWindow App(sf::VideoMode(800, 600, 32), "SFML Graphics");

      // Start game loop
      while (App.IsOpened()){
        // Process events
        sf::Event Event;
        while (App.GetEvent(Event))
        {
            // Close window : exit
            if (Event.Type == sf::Event::Closed || (event.type ==
                  sf::Event::KeyPressed && event.key.code ==
                  sf::Keyboard::Escape))
                App.Close();
        }

        // Clear the screen (fill it with black color)
        App.Clear();

        // Display window contents on screen
        App.Display();
    }
    return EXIT_SUCCESS;
}
```

Fig. 21 *SFML Render Window* (2014)

The example above is a minimal application to get SFML up and running. The code also demonstrates how easy it is to poll for user input; all that is needed is to check the event key code against whichever key you would like to activate for. In this case when the escape key is pressed, the application will close.

The SFML base window is enough to render a window and poll for events, but it isn't capable of drawing anything from the graphics module. This is where the *RenderWindow* class comes in; it inherits from the base window and adds features for easily displaying graphics.

A common problem that arises when mixing raw OpenGL commands with the SFML graphics modules is that SFML doesn't preserve OpenGL states by default. OpenGL is essentially a large state machine so when you call a command such as *glEnable(GL_DEPTH_TEST)*, this will be enabled until the program exits or it is disabled somewhere else in the program. This can lead to undefined behaviour that can be hard to track down. Luckily SFML provides us with the option to preserve OpenGL states manually (see fig. 22):

```
// Render using OpenGL here
// ...
application.pushGLStates();
// Render the GUI using SFML graphics module here
GUI.Render();
application.popGLStates();
```

Fig. 22 *OpenGL State Preservation* (2014)

The *pushGLStates()* function saves the current OpenGL render states and matrices while the *popGLStates()* function restores them (Gomila L, 2013). By using this we are free to mix raw OpenGL with the SFML graphics module.

3.7.3: SFGUI

When researching for an appropriate GUI library to integrate into the project we discussed the Simple and Fast Graphical User Interface (SFGUI). We learnt that SFGUI provides a rich set of widgets and is highly customizable in its looks (Iteem, 2013). Let's see how easy it is to create a simple button which, when pressed, changes its label:

```
// Create a button
auto button = sfg::Button::Create( "Hello" );

// Create buttons function
void OnButtonClick() {
    button->SetLabel( "World" );
}

// Register a callback
button->GetSignal( sfg::Button::OnLeftClick ).Connect(
          std::bind( &OnButtonClick )
);
```

Fig. 23 *Creation of Button and Callback Function* (2014)

The above code (see fig. 23) simply creates a new SFGUI button, creates a function called *OnButtonClick* and then registers this function by binding it to the button using std::bind. To finally render the button, we will add it to a window, and the window to a desktop; which is like a workspace (see fig. 24) (Gomila L, 2013).

```
// Create window for gui
auto window = sfg::Window::Create();
window->SetTitle( "Window Title" );
window->Add( button );

// Add window to new desktop
sfg::Desktop m_desktop;
m_desktop.Add(window);
```

Fig. 24 *Creation of Window and Desktop* (2014)

The last step is telling SFGUI to render and update the GUI (see fig. 25) (Gomila L, 2013).

```
// Somewhere in the main loop
// ...
desktop.Update( 1.0f ); // Update
sfgui.Display( an_sfml_render_window ); // Render
```

Fig. 25 *Rendering and Updating GUI* (2014)

3.7.4: OpenGL

As discussed earlier in the research chapter, OpenGL is an API for rendering 2D and 3D vector graphics and is typically used to interact with the GPU, but how do we do this? The OpenGL Shading Language (GLSL) is a high level programming language created to give developers direct control of the graphics pipeline however, they are not stand-alone applications; they utilize the OpenGL API and are essentially just a set of strings that are passed to the hardware driver for compilation (Kessenich K, 2014). As the project requires

complex terrain rendering such as per-pixel lighting calculations, most of the functionality will be written with GLSL to increase not only the speed on compilation, but also the run-time speed of the application. An overview of the OpenGL pipeline is shown below (see fig. 26):

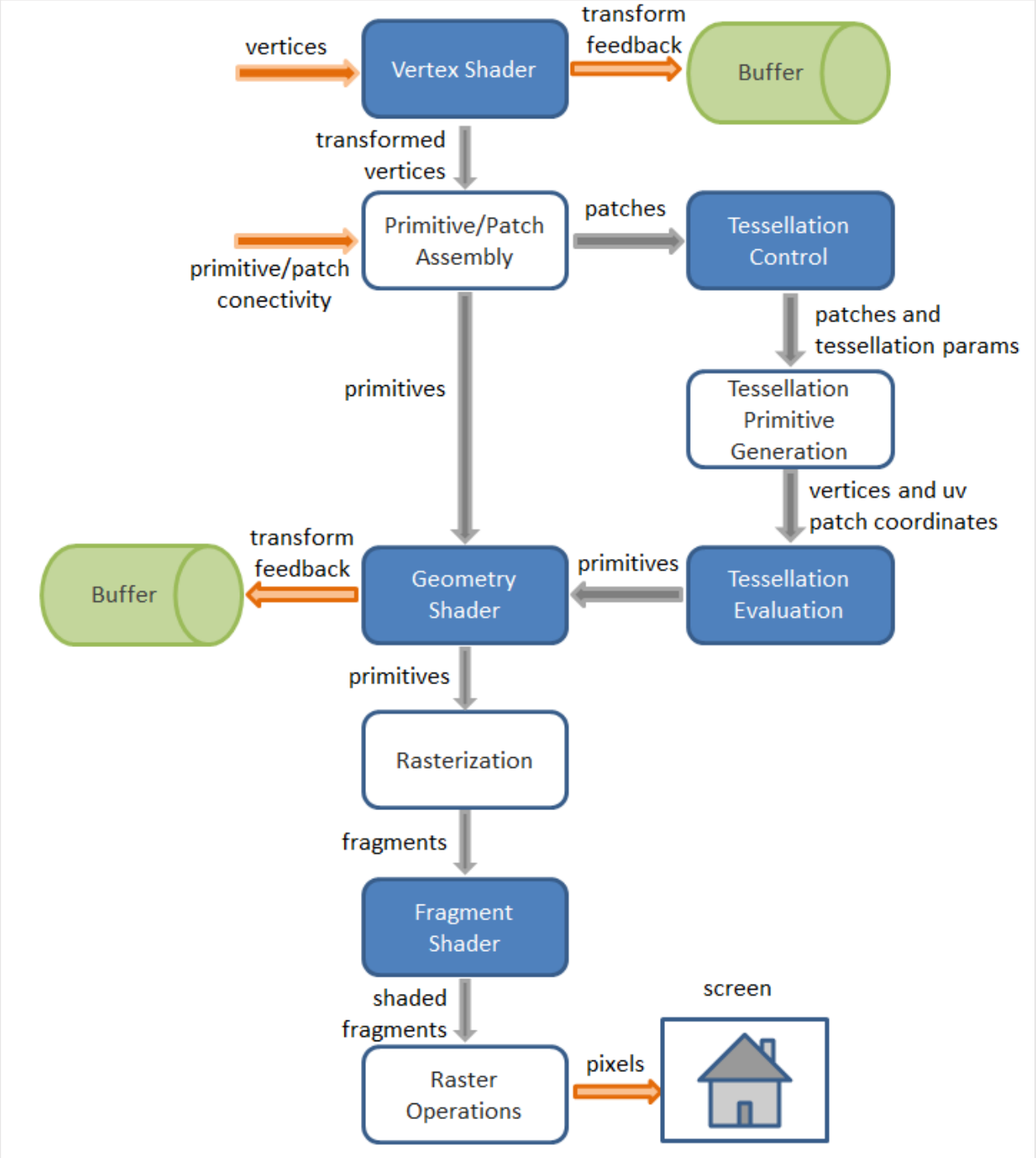

Fig. 26 *OpenGL 4.0 Pipeline* (2011)

The programmable pipeline stages are highlighted blue. As we will be calculating most of the planets data on the fly, the vertex shader will simply be a pass through shader i.e. it doesn't do anything special. It is also unlikely that the geometry shader will need to do anything special also. Where most of the work will be done is in the two tessellation shaders (control and evaluation) as well as the fragment shader. Recapping from the research section, tessellation is the process of subdividing vertex data into smaller primitives. For the purpose of this project, the tessellation shaders will be mostly concerned with tessellating the generated icosahedron to form a sphere and to also push out individual vertices based on the heightmap data. Let's see this in practice; below is a simple tessellation control shader (see fig. 27):

```
#version 420 core

layout(vertices = 3) out;

uniform float tessLevelInner;
uniform float tessLevelOuter;

void main(void){
    if (gl_InvocationID == 0) {
        gl_TessLevelInner[0] = tessLevelInner;
        gl_TessLevelOuter[0] = tessLevelOuter;
        gl_TessLevelOuter[1] = tessLevelOuter;
        gl_TessLevelOuter[2] = tessLevelOuter;
    }
    gl_out[gl_InvocationID].gl_Position =
    gl_in[gl_InvocationID].gl_Position;
}
```

Fig. 27 *Tessellation Control Shader* (2014)

In order to use the tessellation features of OpenGL we need to use a minimum GLSL version of 4.0. The tessellation control shader is concerned with setting how much tessellation a particular patch gets. In the above example the outer and inner levels are passed into the shader using two uniform variables (global variables that can be passed into the shader program from the application). Below is a screenshot of a tessellated triangle to demonstrate what we mean by inner and outer tessellation levels (see fig. 28):

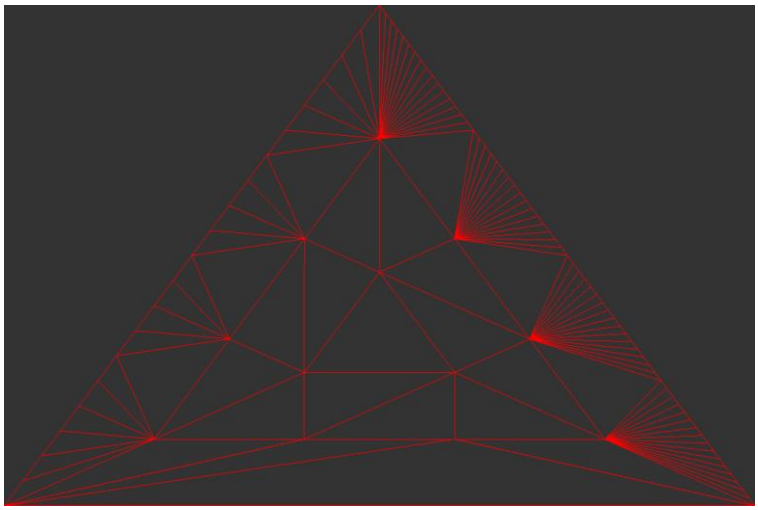

Fig. 28 *Tessellated Triangle* (2014)

As you can see by looking at the bottom edge of the triangle, there is no subdivision. This is because the bottom outer tessellation level has been set to zero. However, the outer level of the right-hand side has been given a tessellation level of sixty-four. The inner tessellation levels have all been set to three as you can see in the centre of the triangle.

This is all great however, how does this relate to the generation of the planet? Well if we take the icosahedron data that has already been predefined in the application, tessellate it and push the points along their normal, we can form a perfect sphere. The code snippet below (see fig. 29) shows how the tessellation evaluation shader can accomplish this.

```
...
void main(void) {
    vec3 p0 = gl_TessCoord.x * gl_in[0].gl_Position;
    vec3 p1 = gl_TessCoord.y * gl_in[1].gl_Position;
    vec3 p2 = gl_TessCoord.z * gl_in[2].gl_Position;
    vec3 tePosition = normalize(p0 + p1 + p2);

    ...

    // SET FINAL MVP POSITION
    mat4 mvp = projMatrix * viewMatrix * modelMatrix;
    gl_Position = mvp * tePosition;
}
```

Fig. 29 *Tessellation Evaluation Shader* (2014)

What this shader is essentially doing is taking the three vertices of the passed in triangle, normalizing the results and then setting its final position by multiplying it with the model view projection matrix. As stated previously, this results in an evenly spaced sphere as shown below (see fig. 30-31):

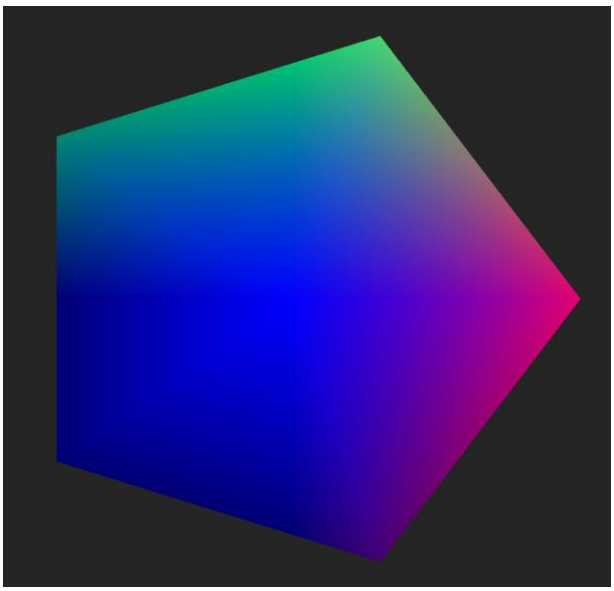

Fig. 30 *Un-tessellated Icosahedron* (2014)

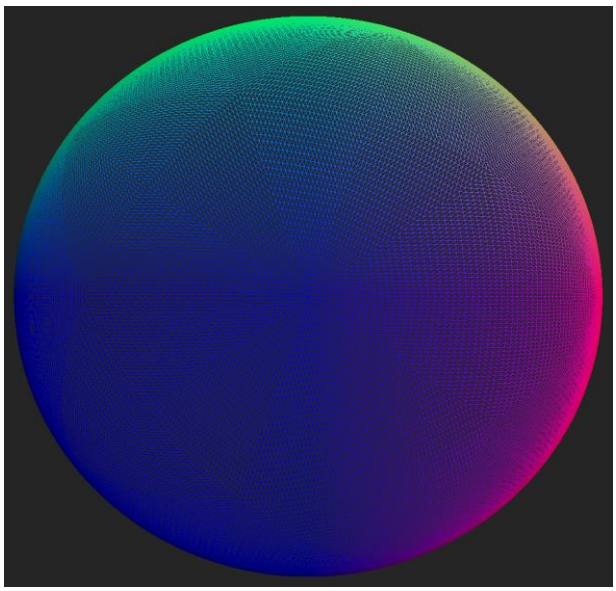

Fig. 31 *Tessellated Icosahedron* (2014)

All that is left to do is to generate the UV texture coordinates, which was discussed in the research chapter, and then push each vertex along their normal by an amount based on the heightmap colour. This results in a nice start to the terrain generation tool.

3.7.5: JSON & JsonCpp

As the main aim requires users to be able to import and export their generated planets, we need a way to store various session data, such as the noise algorithm used, the seed number, the light intensity etc. Although it is simple enough to just store these settings in a plain text file, this can become quite cumbersome to work with as it gets larger. Another solution is to save and load from a JSON file. The JavaScript Object Notation format is a lightweight data-interchange format and is easily readable to humans (Lepilleur B, 2000). It is also easy for machines to parse and generate. It can represent integer, real number, string, an unordered sequence of values and a collection of name/value pairs and as such, is ideal for the projects import and export features. A typical JSON file will look like this (see fig. 32):

```
//Configuration options
{
	// Default algorithm for generator
	"algorithm" : "Diamond-Square",

	// Thermal erosion
	"thermal-erosion": true,
	"thermal-erosion-intensity": 25,

	"light": {
		"colour": { "red" : 255, "green" : 255, "blue" : 255 },
		"position": "x" : 0, "y" : 50, "z" : 0,
		"radius": 100,
	}
}
```

Fig. 32 *JSON example* (2014)

In the above JSON example, we store the noise algorithm in the field “algorithm” with the string value of “Diamond-Square”. The “light” field is an object that has three fields inside of it; “colour” and “position” are arrays and the “radius” field has an integer as a value. In order to read and write JSON files the JsonCpp library will be used. This library is a portable, open source JSON reader and writer written in C++ (Lepilleur B, 2000). It is also incredibly easy to use and will therefore speed up the development process. Below is a quick example of how straight forward it is to read a value from a JSON file (see fig. 33):

```
Json::Value  root;   // will contains the root value after parsing.
Json::Reader reader;
bool parsingSuccessful = reader.parse( config_doc, root );
if ( !parsingSuccessful ) {
    // report to the user the failure and their locations in the document.
    std::cout  << "Failed to parse configuration\n"
               << reader.getFormattedErrorMessages();
    return;
}

// Get the value of the member of root named 'encoding', return 'UTF-8' if
// there is no
// such member.
std::string encoding = root.get("algorithm", "Perlin" ).asString();
```

Fig. 33 *JsonCpp Reading values* (2014)

First we check if the parsing of the JSON file has failed and if so, we can display the error message and return. If it was successful we then only need to call the get method on the root object; the first parameter “algorithm” is the name of which value to return, the second parameter is the default value in case it is null. When it comes to exporting the users’ settings, JsonCpp includes a simple writer to do so (see fig. 34):

```
//...
// At application shutdown to make the new configuration document:
// Since Json::Value has implicit constructor for all value types, it is not
// necessary to explicitly construct the Json::Value object:
root["algorithm"] = getCurrentAlgorithm();
root["thermal-erosion"] = getCurrentThermalErosion();
root["thermal-erosion-intensity"] = getCurrentThermalErosionIntensity();

Json::StyledWriter writer;

// Make a new JSON document for the configuration.
std::string outputConfig = writer.write( root );
```

Fig. 34 *JsonCpp Writing values* (2014)

Here we are storing some of the systems settings using getters, creating a JSON writer object and then parsing the JSON to a string ready to be saved to file.

3.7.6: EasyBMP

EasyBMP is a simple, cross-platform, open source C++ library designed to easily read, write and modify Windows bitmap (BMP) images (Macklin P, 2011). It is important that the technologies being used are cross-platform to increase the amount of potential users. The need for a bitmap manipulation library stems from the need to generated and scan over heightmaps. It is also incredibly simple to work with as shown below (see fig. 35):

```
void DiamondSquare::saveToBitmap() const {
	BMP heightMap;
	// Set size to DATA_SIZE x DATA_SIZE
	heightMap.SetSize(dataSize, dataSize);
	// Set its color depth to 32-bits
	heightMap.SetBitDepth(bitDepth);

	for (int x = 1; x < dataSize; x++) {
		for (int y = 1; y < dataSize; y++) {
			// The colour to be used per pixel
			float colour = ((data[x][y] - minY) / (maxY - minY))
					* 255.0;

			// Set pixel colour
			heightMap(x,y)->Red = colour;
			heightMap(x,y)->Green = colour;
			heightMap(x,y)->Blue = colour;
			heightMap(x,y)->Alpha = 1;
		}
	}
	std::string name = fileName + ".bmp";
	heightMap.WriteToFile(name.c_str());
}
```

Fig. 35 *EasyBMP saving* (2014)

Here we are creating a BMP object, setting the size of the object to be as big as the data size (this is the array of height nodes) and then setting the bit depth; this is the number of bits used to represent the colour of a single pixel. All that is left to do is to loop through the data and for each pixel, set the red, green, blue and alpha values; alpha being how transparent or opaque the pixel should be ranging from zero to one respectively. To save the bitmap we only need to call the write to file function which takes in a string; this is what the file should be called.

3.7.5: Visual Studio & Team Foundation Server

Microsoft Visual Studio is the integrated development environment (IDE) of choice on this project for various reasons; it is generally regarded as one of the best IDEs on the market providing excellent intellisense and debugging capabilities to aid in development (Microsoft, 2013). It also integrates very well with Microsoft Team Foundation Server (TFS). TFS is a product which provides source code management, reporting, requirements management and testing capabilities (Microsoft, 2013). The terrain generation system will be quite large and complex upon completion so the use of source control is essential for backing up the code in case it becomes corrupted or lost. There are other advantages of using TFS too, such as providing a list of comments after each check-in so you're able to see what has or still needs to be done; this is especially useful if multiple people are working on the project at the same time. The best feature

that TFS offers for this project however, is the ability to branch and merge code. As this project will essentially be a prototype, a lot of trial and error will go into it. To help with this we can branch off our code from the main branch into another, let's call this the test branch. In this test branch we can implement new features into the system without the risk of breaking the main solution (even if we broke the test branch, the changes can just be scrapped by rolling back anyways). After we are happy with the new changes we can merge the new test features into the main solution (see fig. 36). This also reduces the risk of bugs being introduced into the main solution as the test branch is a sandbox were we can fully test the new feature first.

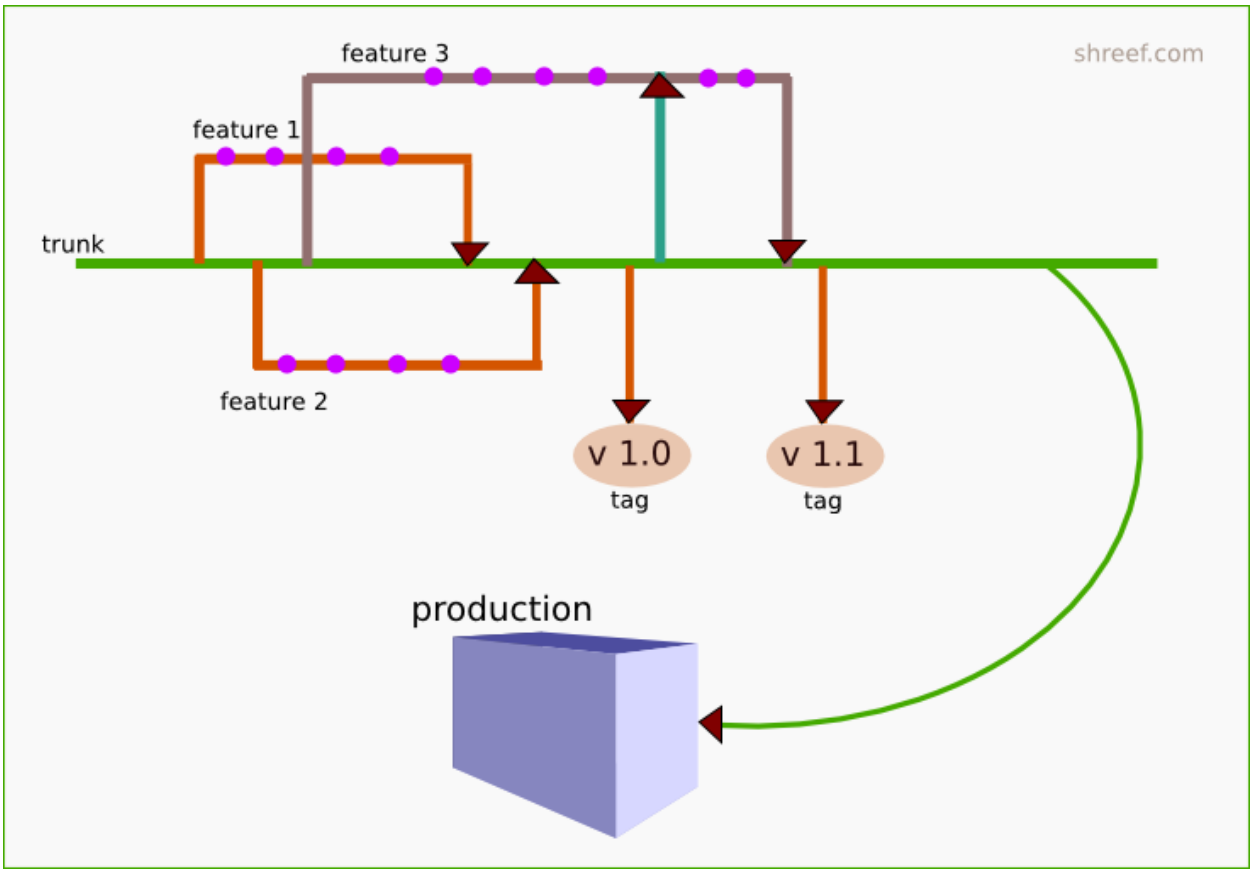

Fig. 36 *Source Control Branching* (2010)

3.8: User interface

When talking about GUI design, two issues often crop up: usability and accessibility. Although closely related they are not quite the same thing. Usability is about ensuring that the design is user-friendly, intuitive or easy to grasp. Accessibility is about making sure that the GUI doesn't create barriers for those who have disabilities or need to access your resource in alternative ways. If the GUI design is not usable it

will frustrate the user; if it is inaccessible, it may actually exclude potential users altogether (JISC Digital Media, 2000). The terrain generation software could become quite complicated as it is unknown how many parameters will need to be exposed to the GUI. The problem here is that the interface could become quite cumbersome, especially if it is designed alongside development. For this reason it is necessary to plan out a brief design to speed up development time and reduce the risk of creating a complicated GUI. Below is an example of how the application should look and function (see fig. 37):

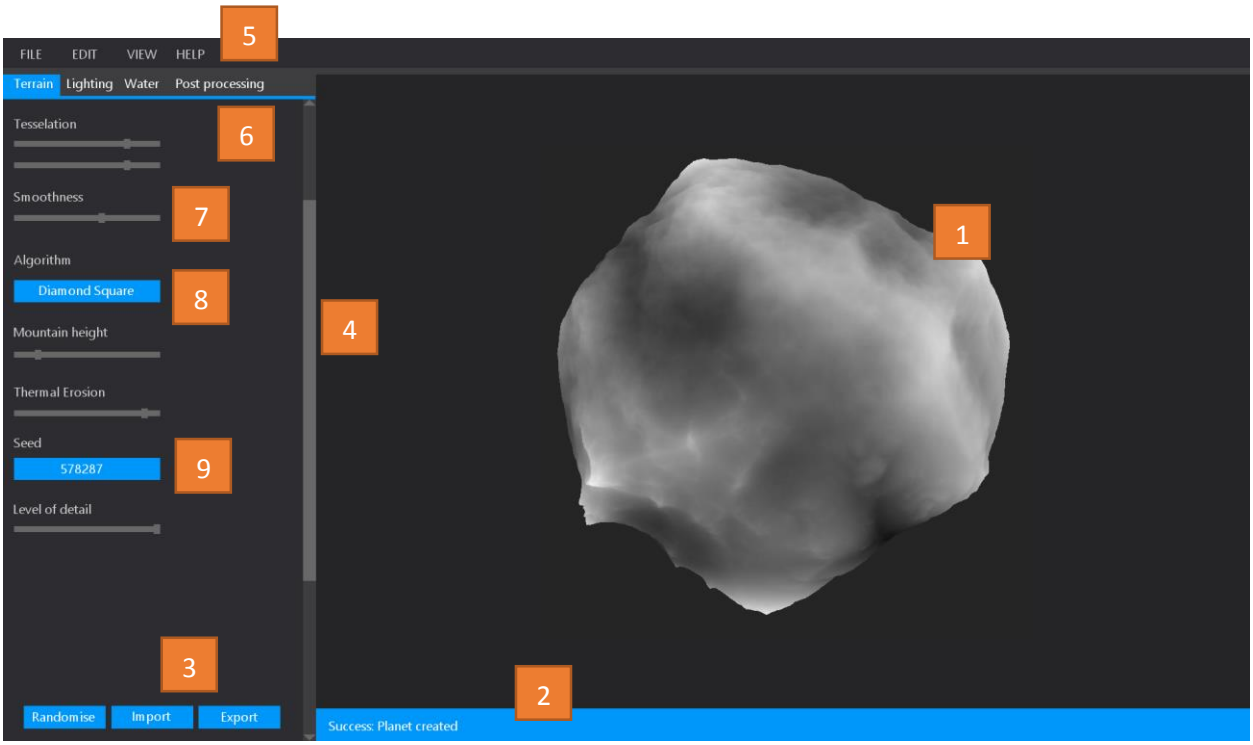

Fig. 37 *System Design* (2014)

1: OpenGL window

This is the main drawing area of the application showing the generated terrain. The user is able to hold the left mouse button and rotate the planet. The mouse scroll wheel will also allow the user to zoom in/out. To make this functionality clear on start-up, the user should be presented with a tooltip explaining this.

2: Feedback bar

This bar displays system information to the user to let them know when something important has happened, if this is an error or something that the user shouldn't do, it will display red instead of blue. For example, if a previous planet had been imported correctly it will show a blue bar and display "Success:

Planet imported correctly". However, if the import had failed it would show a red bar and display "Error: import failed". Blue and red have been chosen over green and red due to a large amount of the population having colour perception problems; this improves the accessibility of the GUI.

3: Main buttons

The most common buttons that the user will probably use have been added to the bottom of the scroll pane to increase usability. This includes a randomize button which will just display a random planet to quickly get started, as well as an import and export button to either load an existing planet or save the current planet respectively.

4: Scroll pane

As the amount of parameters to expose to the user are currently unknown, the widgets have been placed in a scroll pane in case the amount of widgets exceeds the screen height. This will make implementation simpler.

5: Menu system

All of the features listed in the scroll panes will also be able to be modified in the system menu. The system menu can also be controlled using hotkeys. This will improve accessibility for users who are operating the application with just a keyboard and also act as an error catch; if something goes wrong and a widget isn't working correctly, it may be working through the menu system.

6: Tab pane

The system will be split into different sections such the actual terrain generation, lighting, texturing and any post processing effects. The system will have separate tabs for these sections rather than fitting all of the widgets into one scroll pane. This will result in less bloated GUI which will improve usability.

7: Slider widget

This will be a simple slider widget whereby the user can drag from the minimum set value to the maximum.

8: Drop down box

This will be a simple pre-defined drop down menu for the user to choose from.

9: Text box

This is where a user may specify themselves values such as the random seed number, or the RGB colour of the light. These widgets need to be carefully validated as it is a point where users may inject incorrect data into the system causing undefined behaviour i.e. entering a string into the seed field when the system expects an integer.

3.9: Room for extension

The most important part of the terrain generation tool is to create the solution in such a way that it is easy for developers to add further functionality in the future. This will be accomplished by taking a modular approach and reducing dependencies between classes. To help with this a profiling add-on for Visual Studio can be used which highlights how dependent the code is, it will be important to keep track of this throughout development and alter the code if needed. Another reason the code should be easy to extend is if the project is released as open source software afterwards and published on somewhere like GitHub, for others to contribute. If the application is easy to extend then others developers would be able to customise the program for their needs.

3.10: Summary

The methodology chapter has outlined a few functional and non-functional requirements which, if met, will strengthen the overall project. We can also test against these requirements, along with the projects aim and objectives, during testing to help measure the projects' success. To help speed up the implementation of the system a software development methodology has been agreed upon and the system architecture has been briefly planned out. We have also experimented with the decided upon technologies that will be used during development to gain a greater understanding of these before creating the final solution.

Chapter 4: Testing & Results

4.1: Overview

In order to meet the projects objective of effectively evaluating the tool based on performance, aesthetics and ease of use, various testing methods were used. The following tests are broken down into application testing (does the program react as expected), performance testing (how efficient is the program in terms of memory and CPU usage) and user testing (evaluating any feedback from users). Any issues discovered during testing will also be highlighted. It is also worth noting that constant unit testing of all functions was carried out throughout the development of the application, this will reduce the amount of bugs during the final testing stages, especially hard to track bugs related to incorrect data. Screenshots of the finished application can also be seen in the appendix (see Fig. 55-57).

4.2: Application testing

A common strategy when testing is the use of box testing. Box testing normally falls under two categories; black-box or white-box testing (see fig. 38). These categories depend on the view that the tester takes when designing test cases. In the case of black-box testing, the tester has no knowledge of the internal implementation of the program; they are only aware of what the program is supposed to do, not how it does so (Mohd, E K, 2010). Common black box testing methods include boundary value testing and decision table testing. White-box testing on the other hand, tests the internal structure of the program; the tester chooses specific values to test every code path the system may take when responding to user input (Mohd, E K, 2010). Common white-box testing methods include fault injection (purposely introducing faulty data to functions to see how the program copes) and code coverage (creating tests to cover all statements in the program at least once). Both of these methods will be implemented to reduce the number bugs upon release.

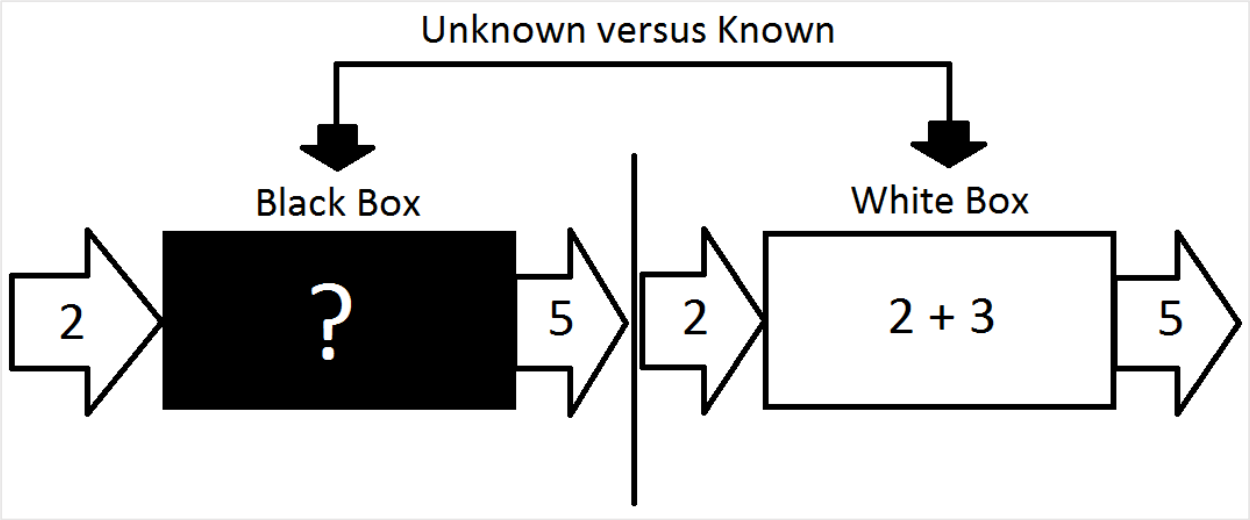

Fig. 38 *Black vs White Box Testing* (2014)

4.2.1 Black-box testing

Terrain generation		
Test case	*Expected result*	*Actual result*
Noise Seed input box: boundary test for the integers entered.	The program should cap the seed number between 0-9999 i.e. user enters -100, program changes this to 0.	As expected.
Corner Seed input box: boundary test for the integers entered.	The program should cap the seed number between 0-9999 i.e. user enters -100, program changes this to 0.	As expected.
Noise Seed input box: seed number affects the heightmap image.	The heightmap image should change depending on the seed number. The image is rendered at the bottom of the tool window.	As expected.
Corner Seed input box: seed number affects the heightmap image.	The heightmap image should change depending on the seed number. The image is rendered	As expected.

	at the bottom of the tool window.	
Tessellation sliders: boundary test.	Both the inner and out tessellation sliders should be capped between 0-500 to avoid tessellation shader from crashing.	As expected.
Tessellation sliders: factors affect the planets mesh.	Altering the inner and outer tessellation sliders should increase/decrease the triangle count of the planets mesh.	As expected.
Smoothing filter: boundary test.	The smoothing slider should be capped relatively low (between 0-5) as this is an extremely expensive operation.	As expected.
Noise algorithm drop down box: decision test.	The program should change its terrain generation algorithm based on the users' choice.	The choice made in the drop down does reflect the programs choice of algorithm however, there are clear artefacts displayed in the planets mesh when choosing "Diamond-Square".
Mountain height slider: boundary test	The mountain slider should cap the mountain height to realistic heights (0-200).	As expected.
Mountain height slider: decision test.	Changing the mountain height should visually alter the landscape.	As expected.
Data size dropdown box: decision testing.	The program should generate different sized heightmaps based on the users choice i.e.	As expected.

	selecting 128 should generate a 128x128 heightmap.	
Randomise button: decision test.	The program should randomize the noise and corner seed box values, the noise algorithm drop down box, smoothing filter and mountain height slider.	Program does not randomise the mountain slider.

Water generation		
Test case	*Expected result*	*Actual result*
Water scale slider: boundary test.	The water scale slider should cap the water levels to realistic heights (0-maximum mountain height).	As expected.
Water scale slider: decision test.	Changing the slider should alter the water levels.	As expected.
Ripple intensity scale slider: boundary test.	The ripple intensity slider should cap the waters ripple intensity to realistic results (0-50).	Upper range ripples too intense and don't look realistic.
Ripple intensity scale slider: decision test.	Changing the slider should alter the water ripple intensity.	As expected.
Transparency scale slider: boundary test.	The transparency slider should cap the transparency of the water between fully transparent (0) and fully opaque (1).	As expected.
Transparency scale slider: decision test.	Changing the slider should alter the transparency of the water.	As expected.
Red, blue and green water colour sliders: boundary tests.	Each of the coloured sliders should cap between no colour (0) to full colour (1).	As expected.

Red, blue and green water colour sliders: decision tests.	Changing these sliders should alter the colour of the water.	Incorrect, the boundary values are wrong in the context of SFML colour values, the colour range is limited. The range should not be between 0-1 but between 0-255.

Command panel		
Test case	*Expected result*	*Actual result*
Update button: decision test.	Upon pressing the update button, the program should generate new terrain taking into account all non-real-time input fields.	As expected.
Import button: decision test.	Upon pressing the import button the program should read an already previously saved project file and update the terrain using values in this file.	As expected.
Export button: decision test.	Upon pressing the export button the program should export both the project settings file to be imported in future as well as an object file which holds the meshes data.	As expected.

4.2.1 White-box testing

Fault injection		
Test case	*Expected result*	*Actual result*

User input boxes: Noise & corner seed boxes.	Entering inappropriate values (strings) into these boxes should be handled correctly internally i.e. if a string is entered the user should be notified of the error.	As expected.
UV Texture calculations.	If a new texture is placed inside the assets folder, the program should successfully calculate the new UV texture coordinates for it.	Incorrect, it became clear when using a more detailed texture that the v-coordinate seems to be incorrectly calculated as banding issues are present.
Testing of all code paths.	When the program is debugged and stepped through, all code paths should be tested and no undefined behaviour should be present.	As expected. (No undefined behaviour found)
Memory usage.	The memory usage of the program should remain relatively constant throughout its life cycle.	The memory usage increases over time – sign of a memory leak.

4.3: Performance testing

To satisfy the objective of analysing the program based on performance, the profiling tool in visual studio was used. This profiler displays information such as CPU and memory usage; using this we should be able to see if any CPU bottlenecks occur and whether any memory leaks are present.

4.3.1: System usage

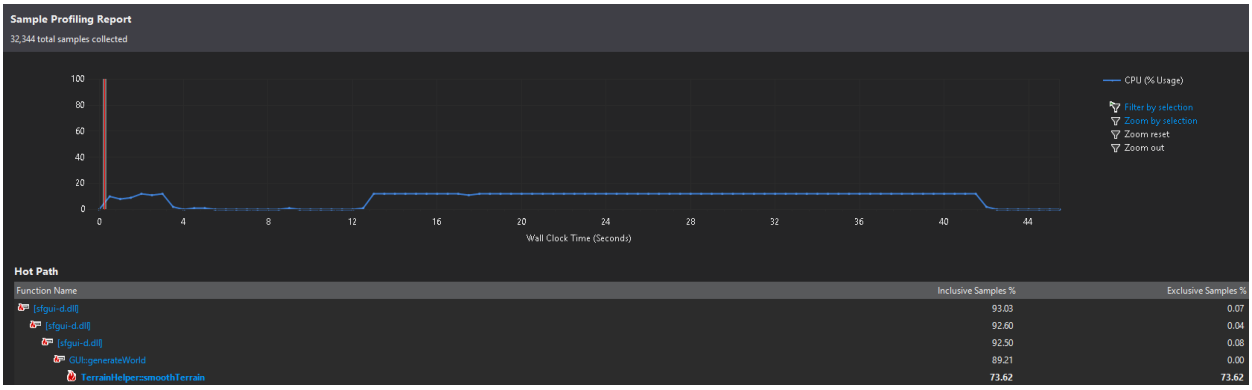

Fig. 39 *Application CPU Usage* (2014)

The report (see fig. 39) shows the systems CPU usage. What was noticeable when testing system performance, were the large CPU inclines when choosing to manually update the planet. The reason behind this is that when manually updating the planet, the terrain generation algorithm needs to be run again; this is done on the CPU which is why it takes longer to update the planet and is why we see CPU spikes. It would be possible to reduce these spikes by unloading most of the processing to the GPU, this would however, require quite drastic changes to the code base.

The biggest cause of CPU spikes is the result of having a high value for the smoothing filer (more than one). This is highlighted in the above diagram as the TerrainHelper::smoothTerrain function is the last function ran before the CPU incline. The reason this happens is because the smoothing algorithm needs to scan line by line the values of the heightmap and perform an average function; the higher the smoothing slider value is set the more times the program needs to loop through this process, stalling the application. Again, with major changes in the program, we could unload the responsibility from the CPU to the GPU by placing this algorithm inside a shader which would reduce the bottleneck. Another way to make sure the user doesn't stall their program for too long is to just simply set the total smoothing loops to a low amount, such as five.

It became noticeable after running the program for a long period of time that the memory usage continued to increase. This was a sign of a memory leak somewhere in the system. To find this, a table was made to see how much the memory requirements increased when pressing each button/slider in the toolbar. It became apparent that when pressing the update button that the memory usage of the application increased and yet never went down. This led to increasing the amount of white box testing of the terrain generation function (which is what the update button executed) until it became clear what was causing this issue. It was discovered that the program didn't delete the newly created Diamond-Square variable

after the manual update which is what was causing the memory usage to increase. By simply deleting this variable before generating new terrain fixed the memory leak (see fig. 40). The programs memory usage now remains constant.

```
// FIXED: Delete heightMap object before creating new
delete heightMap;
...
// Create new heightmap object
heightMap = new DiamondSquare("heightmap", Settings::dataSize,
Settings::noiseSeed, Settings::cornerSeed, 32);
```

Fig. 40 Fixed *Memory Leak in Update Function* (2014)

4.4: User testing

User testing was the last type of testing done on the project as the idea was by this point, all crashing bugs and undefined behaviour should have been caught by application and performance testing. The objective decided upon when starting the project was to evaluate the usability and aesthetics of the program which is quite subjective and difficult to test personally. To do this, randomly selected individuals were chosen (thirteen in total) to try the terrain generation tool. Afterwards, they were asked to fill in a feedback form which included both qualitative and quantitative questions relating to ease of use and how aesthetically pleasing the generated planets are. There was also an optional comments field which asked what other features they would like to see implemented in any future work. The users were also given a list of unfinished features beforehand to avoid these features being taken into account when evaluating the software.

4.4.1: Ease of use

Users were asked to evaluate how easy the application was to use. Below are the results from a set of thirteen users.

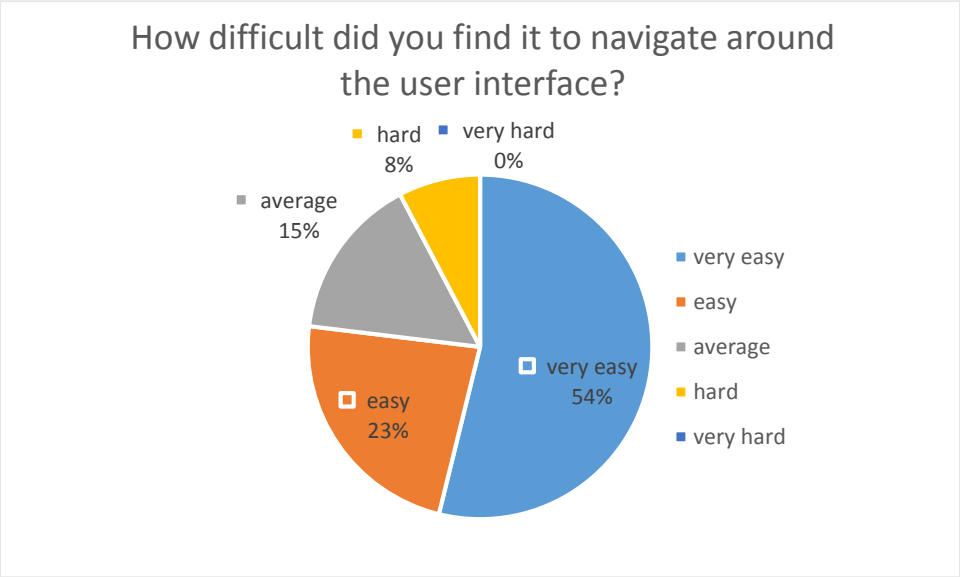

Fig. 41 Navigation (2014)

The majority of users (54%) found the GUI very easy to navigate with only 8% finding the application quite difficult. This reinforces one of the objectives set upon; allowing users to be able to easily manipulate the generated planet through a GUI in real time.

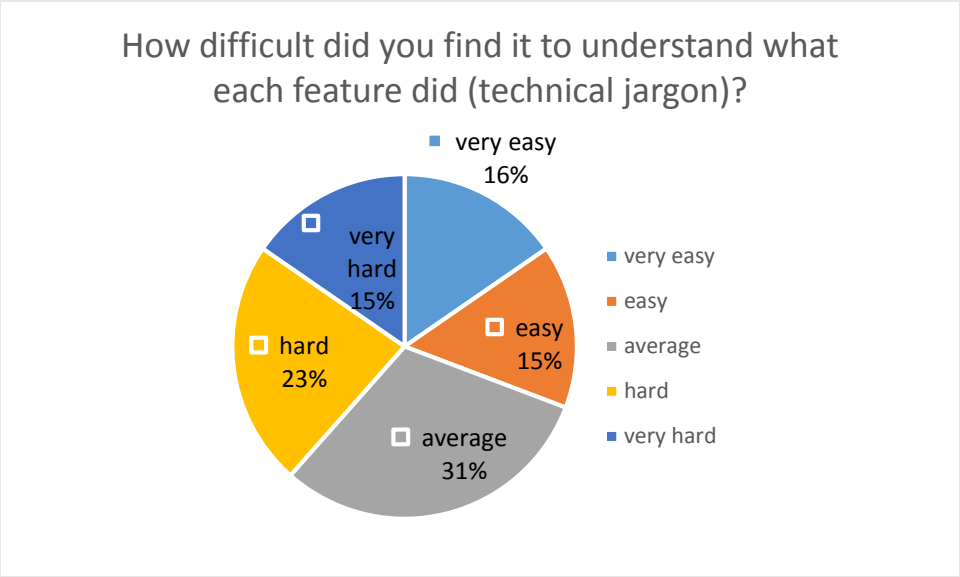

Fig. 42 Jargon (2014)

This question raised mixed views on how difficult people found the technical jargon to be with the majority of users finding the jargon to be about average. This was to be expected as it is difficult to get across what

each feature actually does without using technical terms, especially when it came down to terrain manipulation features such as tessellation and seeding.

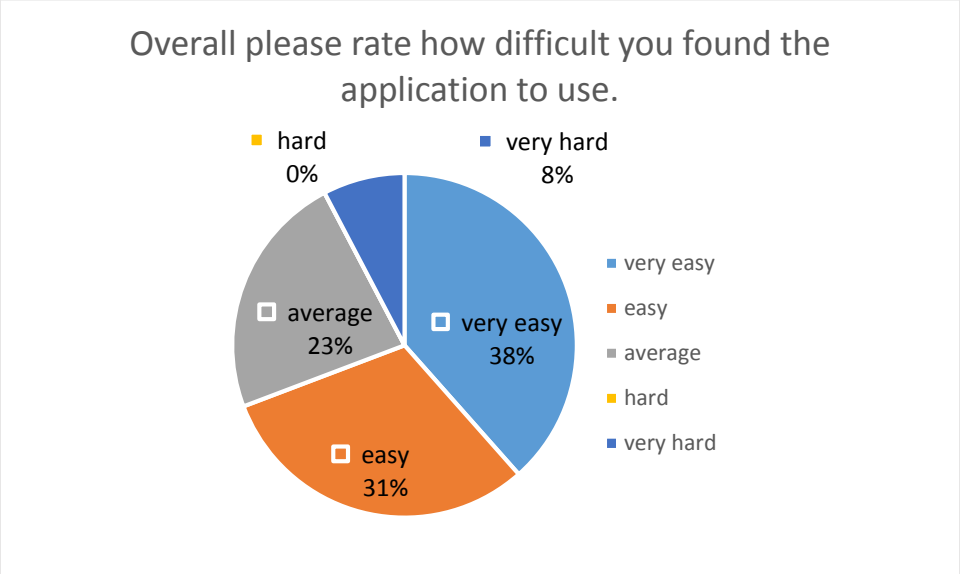

Fig. 43 Overall Ease of Use (2014)

Overall, 91% of users agreed that the application was very easy to average on how difficult it was to use.

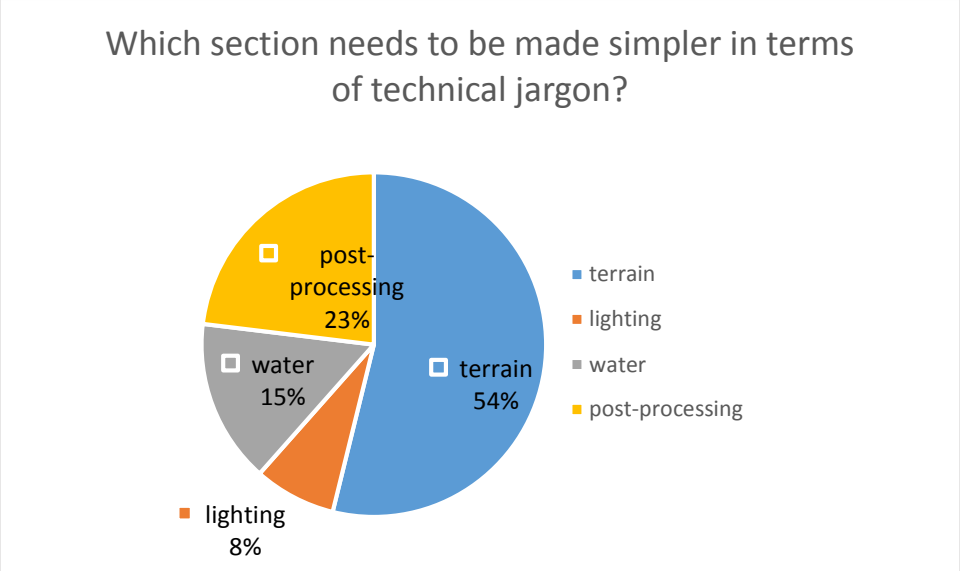

Fig. 44 Jargon Section (2014)

The majority of users found the terrain section in the GUI to be the most difficult to understand in terms of technical jargon. Again this was to be expected as it is the most bloated section with a lot of advanced features for the user to choose from.

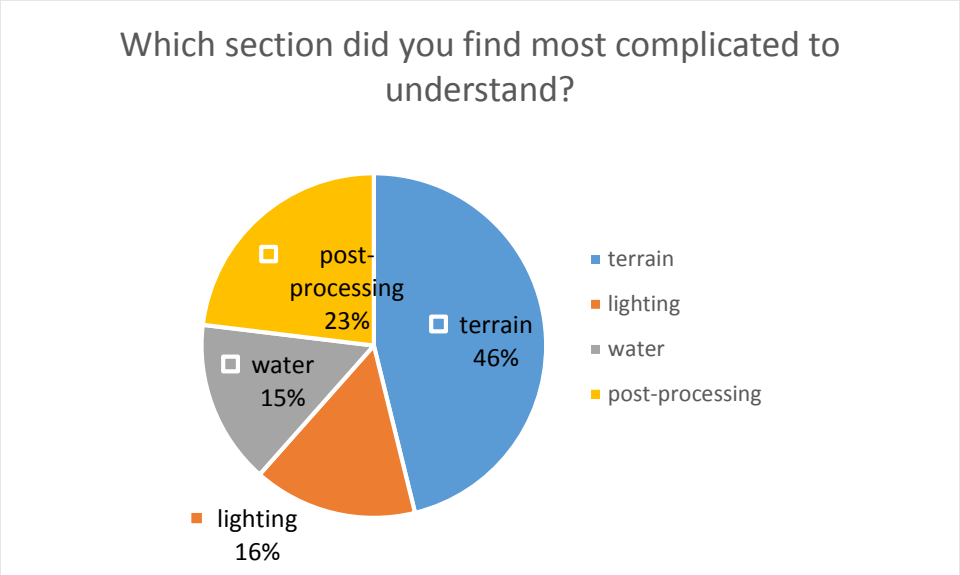

Fig. 45 Most Complicated Section (2014)

The terrain section again was highlighted as being the most complicated to understand with the other sections being around equal to each other.

The users were also asked to leave some written feedback with regards to ease of use. The most common request was that the widgets should have some sort of tooltip to explain in greater detail what they are actually doing under the hood.

4.4.2: Aesthetics

Lastly, the aesthetics of the generated planets were evaluated. This covered evaluating the land, water, lighting and overall look of the terrain.

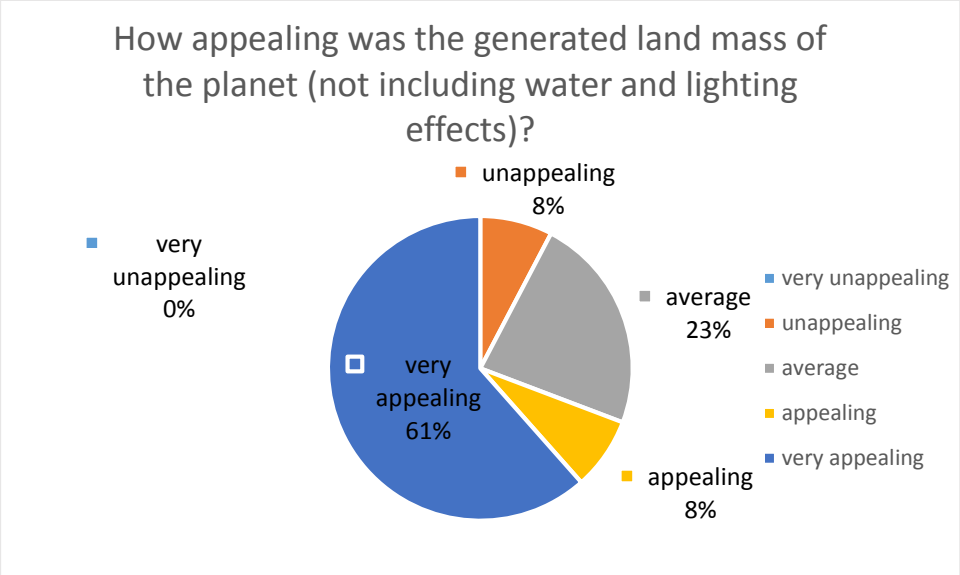

Fig. 46 Land Mass (2014)

The majority of the users found the actual land that could be generated to be aesthetically pleasing with no users rating the land to be very unappealing.

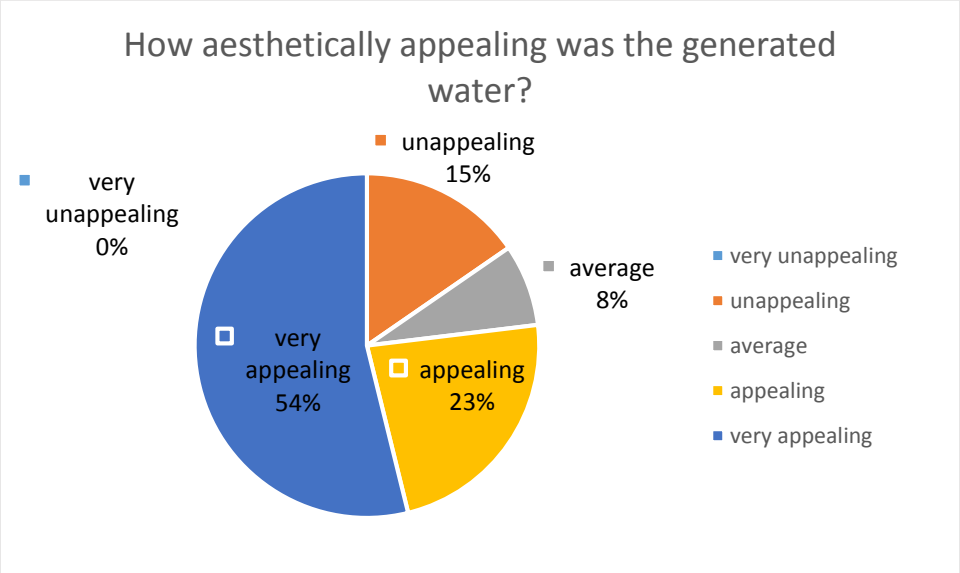

Fig. 47 Water (2014)

The water effects results ended up being slightly lower than the land effects however, they are still very high and don't impact negatively on the projects aesthetics.

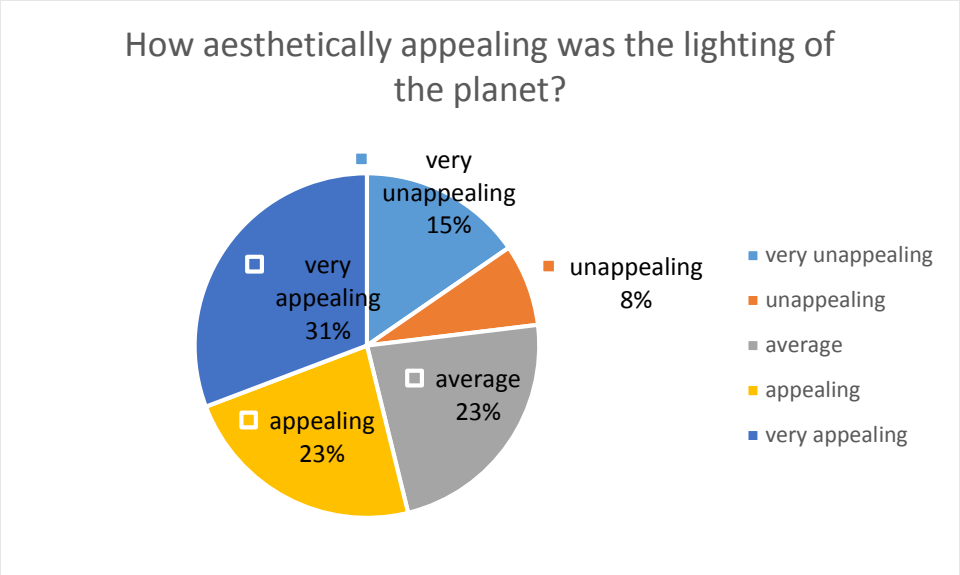

Fig. 48 Lighting (2014)

The lighting effects received mixed results with the majority of users in favour of the lighting effects. This was to be expected as the lighting section was slightly incomplete when user testing was carried out.

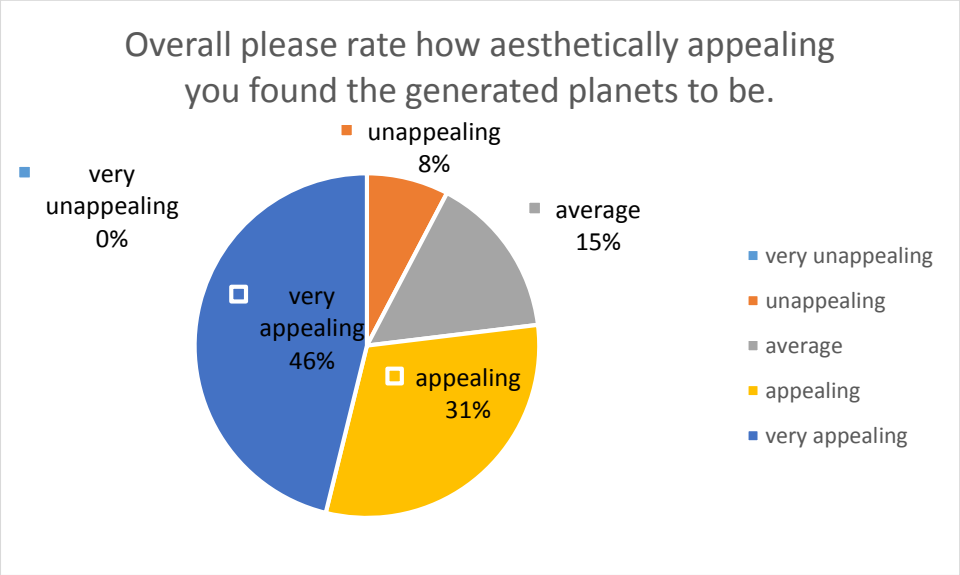

Fig. 49 Overall Aesthetics (2014)

Overall the majority of users found the terrain generation tool to be very appealing with no users thinking that the application was very unappealing.

Similarly to the ease of use section, the users were also asked to leave some written feedback with regards to the applications aesthetics. The most common request was to improve the lighting effects of the application “to not make the planets look so flat”. Another user suggested the inclusion of weather effects on the planet, such as rain, thunder and snow which hadn’t been thought about previously.

4.5: Issue resolution

During the testing stages a few bugs came to surface, some of which were easier to solve than others. Below is an overview of the actions taken to resolve these issues; it is also important to point out that after fixing these issues, the whole application was tested again using the above methods (apart from user testing) just in case the fixes broke something else unintentionally.

4.5.1: Artefacts present on planet mesh after Diamond-Square algorithm runs.

This was a fairly tricky bug to fix, the screen shot below (see fig. 50) shows the problem more clearly:

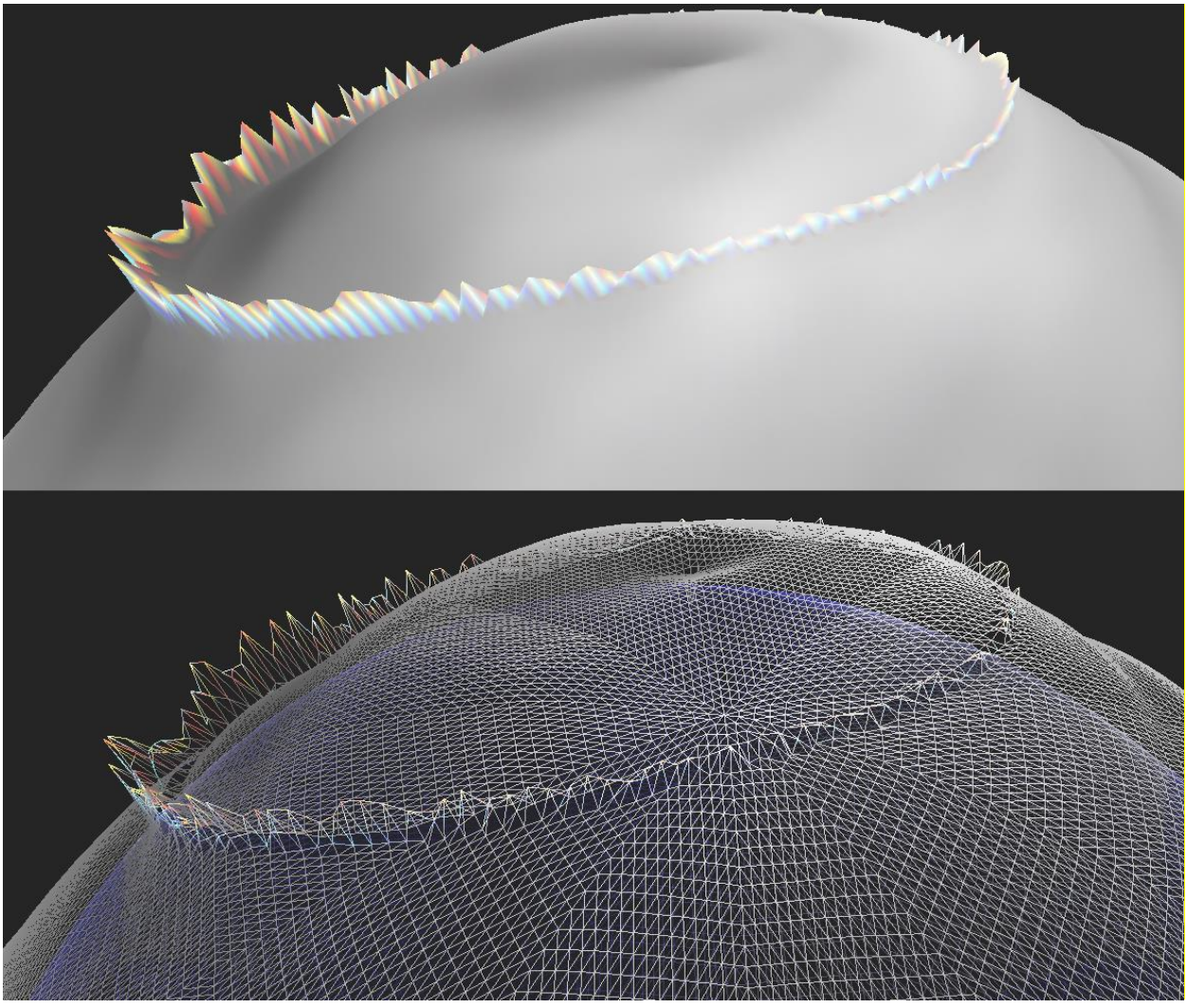

Fig. 50 Mesh Ring Artefact (2014)

As you can see there is a thin, circular ring forming around the sphere. I suspected the heightmap might be incorrect so the first step was to analyse the texture generated by the noise algorithm. By looking closely at the bottom and right hand side of the image it was clear that the pixels of the heightmap generated were off by one row and one column.

This led to analysing the ‘for loop’ code inside the save to bitmap function. As suspected there was indeed an ‘off by one error’ inside the ‘for loop’ which saved the heightmap. Fixing this solved the ring artefact bug.

4.5.1: Texture banding on terrain

Fig. 51 Texture Banding Bug (2014)

This was a fairly simple bug to track down; from looking at the planets' surface (see fig. 51) you can see that the texture has lines running horizontally through it. After investigating the tessellation evaluation shader (where the UV coordinates are generated) it became obvious that the v-coordinate was being incorrectly calculated. Fixing this coordinate calculation solved the banding issue (see fig. 52).

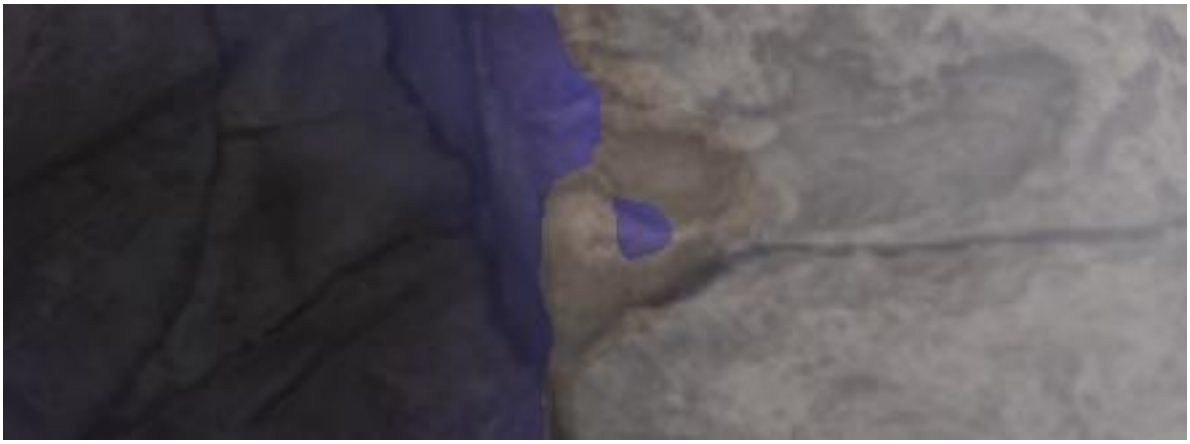

Fig. 52 Fixed Texture Banding Bug (2014)

4.5.2: Mountain slider widget not randomized

The mountain slider widget was not being included when the randomise button was executed. This was simply a case of forgetting to include that specific widget in the randomise function. By simply allowing the mountain slider widget to be included in this randomise function solved this bug.

4.5.3: Water ripple intensity too high

This was simply a case of the upper limit of the ripple variable being set too high which resulted in unrealistic looking water. By simply lowering the upper limit of this variable resolved this issue.

4.5.3: Incorrect results for water colour sliders

This bug came about due to conflicting definitions of the upper and lower colour boundaries of the two APIs used by the application, OpenGL and SFML. OpenGL defines a colour to be within the range of 0-1 whereas SFML defines a colour to be between 0-255. By simply changing the range to be that of 0-1 instead of 0-255 it resolved the bug and now allows the user to choose from a full spectrum of colours.

4.6: Summary

In summary, this chapter covered the full range of testing which the application went through, from simple black-box testing right through to user testing. Crucial bugs were discovered, such as the Diamond-Square off by one bug, which were resolved due to this testing. The feedback that was gathered from user testing will be extremely valuable when evaluating the application as a whole in the next section.

Chapter 5: Evaluation

5.1: Overview

This chapter will evaluate the project as a whole, from how well the initial time plan was kept to, right through to the remaining work left to do. It will also highlight how successful the project has been in relation to the satisfaction of the initial aim, objectives and requirements decided upon at the start of the project.

5.2: Time plan

The time plan was split into three main sections: research, implementation and evaluation. The time put aside for research was adequate enough to gain valuable information relating to terrain generation and rendering techniques. There was also adequate time to discover third-party libraries to aid in development whilst also fulfilling requirements i.e. finding libraries which are cross-platform to satisfy the portability requirement. Most of the technologies and techniques that were researched in the beginning went on to be used in the implementation stage.

The implementation stage was split into core, render and GUI programming stages. The core programming stage had some flaws due to underestimating particular tasks. For example four days were put aside to implement a level of detail algorithm to improve the efficiency of the application. Unfortunately this task was quite difficult due to the way the terrain, by that point, was being generated which conflicted with the methods that were researched previously; this led to the LOD algorithm being unimplemented as it was starting to push back the other milestones.

A similar issue also came about during the render programming stage. The time set aside to implement atmospheric scattering was underestimated; this feature was quite mathematically intensive and the time needed to fully understand this was more than what was put aside. Again, this led to the feature going unimplemented.

The GUI programming stage went according to plan, enough time was put aside to expose as many variables as possible and then implement controls such as sliders and text boxes to manipulate them. This was due to careful preparation of the solution which made adding new controls easy.

A testing section was added at the end of each stage; this was a great decision as it led to no bugs coming to surface during user testing at the end of the project. As mentioned in the testing chapter, various testing methods such as black and white box testing were used to make sure each stage was complete before moving on to the next.

5.3: Satisfaction of aim & objectives

5.3.1: Aim

The initial aim that was decided upon at the start of this project was to create a tool which allows the user to easily generate, modify and export realistic large scale alien planets that are customisable to the users' needs. From the results gathered from user testing as well as the satisfaction of the three objectives and requirements set at the start (discussed below), it is safe to say this aim has been met.

When users were asked how difficult the user interface was to navigate around, 77% found the interface to be easy or very easy to use. On top of this 69% of users found the application 'easy' or 'very easy' to use overall. This satisfies the aim of allowing users to easily generate large scale planets.

The tool was then assessed based on its aesthetics as it was in the projects aim to create "realistic" planets. This was assessed by asking users to judge each part of the program, such as the water and land effects, as well as the overall look of the program. In general the majority of users rated the overall look of the generated planets to be 'appealing' or above (77%). However, the area which caused mixed results was the lightning effects on the planets. This was to be expected as only basic lighting had been implemented due to time constraints; the lighting wasn't as advanced as what was originally planned for. Still, the majority of users rated the lighting effects to be at least average or above. The land and water effects were also voted to be aesthetically pleasing which is why the aim of creating "realistic" looking planets has been met.

The exporting and importing features were also implemented in the tool which allows users to save their current session and to then also load these setting in the future. The export button also creates an OBJ file which includes vertex, normal and texture coordinate data so the planets may be used by other programs. Due to these features the aim of allowing users to export and import planets has been met.

5.3.2: Objectives

Objective one: Investigate and implement modern algorithms for generating random terrain, optimizing the terrain and generating realistic atmospheres.

This objective was partially met. The objective specified three areas to research and implement; terrain generation techniques, optimization of that terrain and generating atmospheres. The research that went into all three of these areas was more than enough; I felt each of these areas had been researched thoroughly before starting the project. The application uses a well implemented Diamond-Square algorithm to generate the terrain which can be further enhanced through thermal erosion and smoothing. The terrain can then be manipulated by adjusting the maximum and minimum mountain height.

The terrain can also be partially optimized; the user is able to adjust the tessellation levels of the terrain to reduce the processing power needed to generated the planets however, due to time constraints a LOD algorithm to section off the patches of terrain based on distance was not implemented.

The generated atmosphere was also not advanced enough to fully satisfy this objective. Currently the atmosphere uses a simple noise texture wrapped around the planets orbit to simulate clouds however, features such as atmospheric scattering of the suns' rays had not been implemented, again due to time constraints.

Due to the simple optimization of the terrain and atmosphere this objective has only been partially met.

Objective two: Allow the user to be able to easily manipulate the generated planet through a GUI in real time.

This objective was fully met. The GUI that was created interfaces with the core program quite well; the user is easily able to alter the terrain, water and lighting effects in real-time. The program has also been built to allow any further features to be easily added to the GUI. As shown when discussing the interfaces ease of use (see fig. 41), 54% voted for the interface to be very easy to use.

Objective three: Evaluate the tool based on performance, aesthetics and ease of use by analysing feedback from both application and user testing.

This objective was fully met. The testing stages after each stage of development included application testing (does the program work correctly), performance testing (how efficient is it when doing those tasks) as well as user testing. Application testing highlighted some crucial bugs (such as the Diamond Square artefact bug), which were fixed because of the extensive testing. The program is also optimised better in terms of CPU and memory usage due to constant performance testing throughout development. User testing was based on how easy the application was to use as well as how realistic it looked (the planets aesthetics). From this feedback (see fig.41-49) we were able to highlight the programs strengths (such as the interfaces ease of use) and weaknesses (weak lighting effects).

5.4: Satisfaction of requirements

5.4.1: Functional requirements

Basic manipulation of the terrain

This requirement has been met. The user is able to manipulate the terrain in the following ways: generate new terrain based on the chosen noise algorithm, alter the maximum and minimum height of the mountains, enable thermal erosion of the terrain and increase or decrease the tessellation levels to improve realism. From user testing it was shown that the majority of users (77% see fig. 41) also found these features to be easy to manipulate using the GUI and also at least 'average' to understand the technical jargon (see fig. 42).

Importing and Exporting

This requirement has been met. The user is able to import settings from a previous session as well as export the planet as an OBJ file to be later used in other graphical projects.

Basic control over rendering

The requirement has been met. The user is able to adjust the texture levels of the terrain, adjust the colour of both the water and light as well as adjusting the waters height, transparency and ripple intensity. To further strengthen the satisfaction of this requirement 46% of users found the general aesthetics of the application to be 'very appealing' with only 8% of users finding the application to be 'very unappealing' (see fig. 49).

5.4.2: Non-functional requirements

Efficiency

This requirement has been partially met. The application was tested for performance which highlighted some CPU spikes when carrying out intensive operations, such as smoothing the newly generated terrain. What could have reduced the applications CPU usage further is by unloading the responsibility of the intensive operations onto the GPU, unfortunately this was not able to be done due to time constraints; the design of the application needed large changes for this to happen. During performance testing a memory leak was found and resolved, the memory usage was monitored afterwards and now stays constant during long periods of activity.

Effectiveness

This requirement was partially met. The algorithms used to generate the heightmap should have been complex enough to spawn a wide range of different patterns. Although the Diamond-Square algorithm was implemented well, it is the only noise generator in the application at present as other algorithms such as Perlin were not implemented due to time constraints. This limits the range of different terrain patterns that the user can generate.

Fault tolerance

This requirement has been met. When the program is active, if the user makes a manual update or there hasn't been a backup made for a while, the program saves its current state. If the program crashes these settings can be loaded so the user can carry on where they left off. To reduce the number of crashes the program has also been thoroughly tested, especially when it comes down to filtering data that comes from user input boxes.

Portability

This requirement has been met. During the application testing stage, the program was ran on multiple platforms (Windows, Mac and Linux) without any problems arising. The reason behind this is due to the technologies involved; every third-party library that was used, such as EasyBMP and SFML, were checked for portability before integrating them into the system. Credit for requirement being met goes to the well conducted research, prior to implementation.

Testability

This requirement has been met. Throughout the development process each new feature was given a separate class, such as the terrain generation function or the terrain smoothing function. Due to this modular approach, it was easy when it came to unit testing; each class could be tested as a single unit without affecting others.

5.5: Remaining work

Due to issues with time management, highlighted above, there was some remaining work left to do at the end of the project. Below is a list of these features as well as a note on how much more work is needed to finish them.

5.5.1: Terrain

The application should have presented the user with an array of different noise algorithms to choose from. At the moment the only choice they have is the Diamond-Square algorithm as there was no time left at the end of the project to incorporate Perlin noise. Even though the results generated from Diamond-Square are realistic enough, adding further noise algorithms to choose from would have given the user more control over the terrain and for this reason, is the most important feature left to implement. I estimate that creating this algorithm, along with integrating it into the solution and testing it, would be around five more days of work.

The project failed to implement a level of detail algorithm to reduce the processing power needed to run the application. Although this is an incredibly useful feature to have, it is a feature which the user can't really tell is missing as it effects the application under the hood. The final program is also quite light with respect to CPU usage so a LOD algorithm is not technically needed yet however, if more advanced features are implemented in the future, the addition of this feature would be more or a priority. I estimate that implementing a LOD algorithm into the solution would require around five more days of work.

5.5.2: Lighting

Atmospheric scattering was not implemented as the time set aside for this was too little; the amount of work was underestimated. Unfortunately, the addition of this feature would have drastically improved both the realism of the atmosphere and planet in general and the feedback from user testing. For this reason it should be given a high priority in the remaining work left to do. As this feature wasn't researched enough it would be hard to say how long it would take to implement.

5.6: Summary

During this chapter we have evaluated the time plan that was set out at the start of this project, it was decided that the time plan overall was a success however, certain elements during the implementation stage could have been improved. The satisfaction of the projects main aim and objectives were also evaluated. It was decided that the main aim had been met as none of the objectives had failed however; objective one could have been improved by implementing more advanced terrain optimisation. Similarly, the satisfaction of the requirements was evaluated and any requirements that had only been partially met were discussed further to investigate why this happened.

Finally, the evaluation concluded with a list of remaining work yet to do; realistic time scales were also provided for these tasks.

Chapter 6: Conclusion

6.1: Overview

This chapter will cover the technical and non-technical lessons learnt throughout this project. It will also cover the specific areas that went well and which parts could have been done better. It concludes by discussing the continuation of the application after this dissertation is completed.

6.2: What has been learnt

6.2.1: Technical

Overall, this project has vastly strengthened my 3D mathematics skills due to the amount of vector and matrix transformations which were computed in various shaders when rendering the planets mesh. I also now fully appreciate and can apply concepts such as normal vectors, due to how useful these are when dealing with areas such as lighting. I feel I now have a firmer understanding of this area of mathematics and I am now more comfortable when faced with these sorts of challenges in the future.

The area that I found to be incredibly interested in during this project was that of noise generation algorithms and its many applications outside that of terrain generation. Although the only noise generation algorithm that I managed to implement during this project was the Diamond-Square algorithm due to time constraints, I'm extremely interested in implementing other types of noise such as Perlin or Simplex noise in the future as a hobby. I would also like to explore using noise in other areas such as water manipulation and the generation of realistic clouds.

The project relied on many third-party libraries such as SFML, EasyBMP and JsonCPP so naturally I have become much more comfortable integrating and linking other libraries into my project, this is a skill which I'm sure will come in useful in industry.

Lastly, it is safe to say, due to how programming intensive this project was, that my overall C++ and OpenGL programming skills have vastly improved taking me from a beginner to an advanced user of these technologies. I am now much more confident when having to debug other projects that use these technologies.

6.2.2: Non-technical

What I have learnt the most during this project, from a non-technical point of view, is judging the amount of time needed for specific programming tasks. I am now a more competent C++/OpenGL programmer because of this project and I find it a lot easier to sum up how long tasks relating to these technologies will

be. This is a valuable skill to have acquired as I'm sure this will come in handy when I continue to develop in industry.

Another important soft skill I have acquired is realising the importance of using the right kind of testing at the right point in the development lifecycle. The project was planned out to include testing stages after each development iteration, during this period both black and white box testing was carried out, including system performance. User testing was left until the project had been fully finished and tested. This led to no crashing bugs or undefined behaviour coming to surface when users were testing the application. It also meant that I had an easier time at the end of the project as I had less to test and was confident in the tolerance of the system due to constant testing throughout.

I have also gained an appreciation of what great tools can bring to development teams. Obviously this project was focused on the creation of a planet generation tool for the sole purpose of letting users create random planets. However, if this project was to create a game which involved users exploring the universe to discover random planets, then using this tool would greatly speed up production times and ease the pressure of the art team. This relates back to what the overall purpose of this project was; creating a tool to reduce the work load of the art team.

6.3: What went well

I believe the time that was put into the research stage at the start of this project paid off when it came to actually implementing the application. The third-party libraries which were chosen integrated flawlessly into the project due to careful consideration. The research was also extensive enough to cover all the techniques and algorithms needed to create fully customizable planets.

In general, the majority of the implementation stage went well; the creation of the sphere by tessellating an icosahedron followed by texturing and creating planes of water went smoothly. The creation of the GUI went even smoother and took less time than was initially planned for. The GUI turned out great and offers a great way for users to interact with the generation of the planet; it also leaves room for expansion that may be implemented in any future work.

The testing that was done throughout the project at the end of each stage went incredibly well; at the end of each stage many bugs were discovered due to this testing which ranged from the system crashing from trying to parse strings as integers, to incorrect boundaries set on the GUI widgets. This led to very few bugs making it through to the end of the project, with no bugs being found during user testing. When developing similar software in the future I will take this approach due to how successful it was.

The feedback that was gained through user testing went quite well. It allowed me to see how well my application was in terms of ease of use and general aesthetics from others peoples point of view; it is sometimes hard to see flaws in your own system as you've worked on the project for so long. It can also be easy to forget that the majority of users who will use your software may not be very tech savvy; so what comes across obvious to the developer may not be to the user. This is why questions such as "how difficult did you find the technical jargon throughout the application" was asked and analysed during user testing so changes like these could be made in any future work.

6.4: What could have been done better

Due to underestimating the task of creating realistic lighting, advanced lighting features such as atmospheric scattering were left unimplemented. This was a shame as this feature would have drastically improved the overall look and feel of the generated planets. The lesson to learn here is to allow myself more time in future when creating the time plan for tasks which are completely new to me.

I could also have made more use of TFS' source control features. The code was rarely checked in which meant that if anything had went wrong locally, I would have lost quite a bit of work which would have severely pushed back my time schedule. I could also have taken advantage more of the branching features in TFS. For example, as the project was very experimental I ended up breaking a lot of the builds when testing new functionality; what I should have done is branch from release to create a "test" branch. Then I could have tested new functionality inside this branch without worrying about breaking the main build. This is common practice in industry and a habit I should have adopted throughout the project.

6.5: Continuation of application

6.5.1: Completion of unfinished work

As highlighted in the evaluation section, a small portion of work still remains to be done. These include implementing a LOD algorithm to increase the efficiency of the program as well as the inclusion of atmospheric scattering to improve the realism of the generated planets. This unfinished work should take priority over the below features as they will make the biggest difference overall to the application.

6.5.2: Advanced water effects

At the moment the water effects are achieved simply by animating the UV coordinates of the water texture using a simple sombrero function. Although this was enough to achieve a basic level of realism, other techniques could have been used to make the water more believable. Another technique that could have been used involves tessellating the water sphere and then constantly pushing the vertices back and forth along their normal by an amount relative to the sombrero functions result (Conrod, J, 2011). This would give an actual 3D ripple effect rather than a fake 2D effect that is currently used in the application. This would increase the overall realism of the generated planets (see fig. 53).

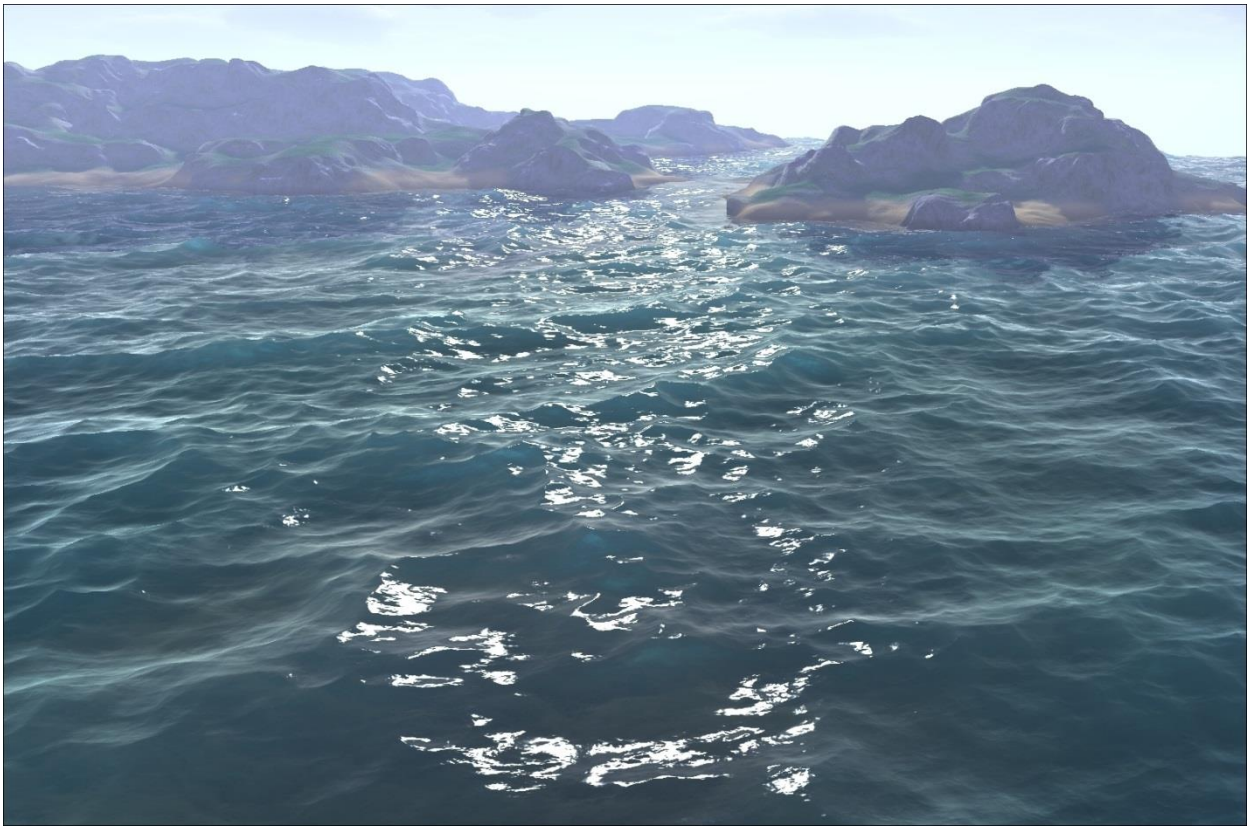

Fig. 53 *Tessellated Water* (2010)

6.5.3: Advanced rendering

To further enhance the visuals of the planets' surface and water, a special type of lighting effect, called bump mapping, could be implemented. Bump mapping is a technique for simulating bumps and wrinkles on the surface of an object (Blinn, J F, 1978). This is achieved by perturbing the surface normal of the object using the perturbed normal during lighting calculations (Blinn, J F, 1978). This results in the surface appearing bumpy although the actual underlying mesh has not been changed. Below shows an image of bump mapping in practice, as you can see this results in much more realistic graphics (see fig. 54).

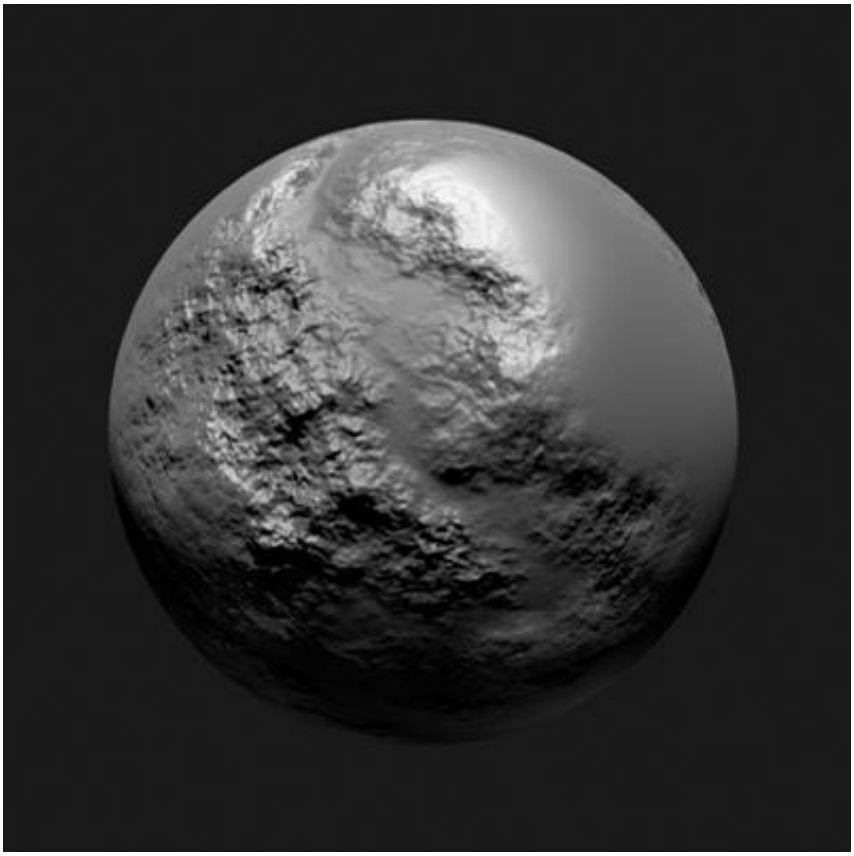

Fig. 54 *Bump Mapping* (2009)

6.5.4: Utilities

When analysing user feedback it came to my attention that a lot of users found it difficult to fully understand what each widget in the GUI actually done i.e. what does enabling "thermal erosion" actually do? To elevate this it would be nice to implement some sort of tooltip functionality that provides more information on what the highlighted feature actually does.

6.6: Summary

In summary, this project has been a success. The initial aim and objectives, alongside the agreed upon functional and non-functional requirements, have been met or partially met. The project is at a very exciting place right now due to the amount of extra features that could potentially be added in the future. Many technical and non-technical skills have been learnt during the course of this dissertation which will no doubt be useful throughout my future career.

References

Archer T. (2009). Procedurally Generating Terrain. Procedurally Generating Terrain. 1 (1), 1-15.

Beard, D. (2010). 3x3 Box Filter [PNG]. Available: http://danielbeard.files.wordpress.com/2010/08/filter.png. Last accessed 13th May 2014.

Beard, D. (2010). Terrain with no smoothing [PNG]. Available: http://danielbeard.files.wordpress.com/2010/08/terrain-simple.png. Last accessed 13th May 2014.

Beard, D. (2010). Smoothed Terrain with a box size of 9x9 [PNG]. Available: http://danielbeard.files.wordpress.com/2010/08/smoothterrain2.png. Last accessed 13th May 2014.

Beard, D. (2010). Box size of 9x9, filtering done 50x [PNG]. Available: http://danielbeard.files.wordpress.com/2010/08/really-smooth.png. Last accessed 13th May 2014.

Beard, D. (2010). Diamond Square Algorithm [PNG]. Available: http://danielbeard.files.wordpress.com/2010/08/diamond-square-algorithm.png. Last accessed 13th May 2014.

Beard, D. (2010). Smoothing Terrain. Available: http://danielbeard.wordpress.com/2010/08/07/smoothing-terrain/. Last accessed 1st Dec 2013.

Beard, D. (2010). Terrain Generation – Diamond Square Algorithm. Available: http://danielbeard.wordpress.com/2010/08/07/terrain-generation-and-smoothing/. Last accessed 1st Dec 2013.

Blinn, J F. (1978). Simulation of wrinkled surfaces. *Computer graphics and interactive technique*. 12 (3), p286-292.

Boesch, F. (2011). Eroded [JPG]. Available: http://codeflow.org/entries/2011/nov/10/webgl-gpu-landscaping-and-erosion/eroded.jpg. Last accessed 13th May 2014.

Cepero, M. (2011). The case for procedural generation. Available: http://procworld.blogspot.co.uk/2011/08/case-for-procedural-generation.html. Last accessed 11/12/2013.

Conrod, J. (2011). Water simulation in GLSL. Available: http://www.jayconrod.com/posts/34/water-simulation-in-glsl. Last accessed 1st Dec 2013.

Devnull. (2008). Oblivion Landscape [JPG]. Available: http://i28.photobucket.com/albums/c209/dev_akm/oblivion/oto/noise/QTP3_vanilla_01.jpg. Last accessed 13th May 2014.

Dexter, J. (2005). Ground Types [JPG]. Available: http://images.gamedev.net/features/programming/texturingheightmaps/img4a.jpg. Last accessed 13th May 2014.

Dexter, J. (2005). Ground Types Blended [JPG]. Available: http://images.gamedev.net/features/programming/texturingheightmaps/img5a.jpg. Last accessed 13th May 2014.

Dexter, J. (2005). Texturing Heightmaps. Available: http://www.gamedev.net/page/resources/_/technical/graphics-programming-andtheory/texturing-heightmaps-r2246. Last accessed 1st Dec 2013.

Dunlop, R. (2005). Spherical Mapping with Normals. Available: https://www.mvps.org/directx/articles/spheremap.htm. Last accessed 13th May 2014.

Dunlop, R. (2005). Sphere U [JPG]. Available: http://www.mvps.org/directx/articles/images/sphtu.jpg. Last accessed 13th May 2014.

Dunlop, R. (2005). Sphere V [JPG]. Available: http://www.mvps.org/directx/articles/images/sptv.jpg. Last accessed 13th May 2014.

Ebert, D S. (2003). Texturing & Modeling: A Procedural Approach. 3rd ed. San Francisco: Morgan Kaufmann Publishers. 489-505.

Fernandes, A R. (2012). Computing Normals. Available: http://www.lighthouse3d.com/opengl/terrain/index.php3?normals. Last accessed 1st Dec 2013.

Geiss, R. (2007). Terrain Created Entirely on the GPU [JPG]. Available: http://http.developer.nvidia.com/GPUGems3/elementLinks/01fig01.jpg. Last accessed 13th May 2014.

Geographic Information Systems. (2012). Perlin Noise [JPG]. Available: http://i.stack.imgur.com/IV2fC.jpg. Last accessed 13th May 2014.

Gomila, L. (2013). Simple and Fast Multimedia Library. Available: http://sfml-dev.org/. Last accessed 13th May 2014.

Iteem. (2013). *SFGUI.* Available: http://sfgui.sfml-dev.de/. Last accessed 13th May 2014.

IT Info. (2005). Software Development Methodologies. Available: http://www.itinfo.am/eng/software-development-methodologies/. Last accessed 13th May 2014.

Janszpilewski. (2014). WhiteStarUML. Available: http://sourceforge.net/projects/whitestaruml/?source=navbar. Last accessed 13th May 2014.

JISC Digital Media. (2000). Graphical User Interface Design: Developing Usable and Accessible Collections. Available: http://www.jiscdigitalmedia.ac.uk/guide/graphical-user-interface-design-developing-usable-and-accessible-collection. Last accessed 13th May 2014.

Kessenich, K. (2014). The OpenGL® Shading Language. Available: http://www.opengl.org/registry/doc/GLSLangSpec.4.40.pdf. Last accessed 13th May 2014.

Lepilleur, B. (2000). JsonCpp Documentation. Available: http://jsoncpp.sourceforge.net/. Last accessed 13th May 2014.

Lighthouse3D. (2011). OpenGL 4.0 Pipeline [JPG]. Available http://www.lighthouse3d.com/wp-content/uploads/2011/03/pipeline4.png. Last accessed 13th May 2014.

Macklin, P. (2011). EasyBMP Cross-Platform Windows Bitmap Library.Available: http://easybmp.sourceforge.net/. Last accessed 13th May 2014.

Microsoft. (2013). Team Foundation Server. Available: http://msdn.microsoft.com/en-us/library/ff637362.aspx. Last accessed 13th May 2014.

Mohd, E K. (2010). Different Forms of Software Testing Techniques for Finding Errors. International Journal of Computer Science. 7 (1), p11-16.

Orbiter Forum. (2009). ROAM System [JPG]. Available: http://orbides.1gb.ru/orbf/sw-090428-4.jpg. Last accessed 13th May 2014.

Perlin, K. (1999). Making Noise. Available: http://www.noisemachine.com/talk1/32.html. Last accessed 13th May 2014.

Rege, A. (2010). Tessellated Water [JPG]. Available: http://blogs.nvidia.com/wp-content/uploads/2010/03/6a00d834515fca69e20120a94420be970b.jpg. Last accessed 13th May 2014.

Rideout, P. (2010). Icosahedron Tessellation [JPG]. Available: http://prideout.net/blog/?p=48. Last accessed 13th May 2014.

Rideout, P. (2010). Triangle Tessellation with OpenGL 4.0. Available: http://prideout.net/blog/?p=48. Last accessed 13th May 2014.

RPGamer. (2006). RPGamer Feature - The Elder Scrolls IV: Oblivion Interview with Gavin Carter. Available: http://www.rpgamer.com/games/elderscrolls/elder4/elder4interview.html. Last accessed 13th May 2014.

Shreef. (2010). Source Control Branching [PNG]. Available: http://shreef.com/wp-content/uploads/2010/11/barnching-in-source-control-systems.png. Last accessed 13th May 2014.

Smelik R M. (2009). A survey of procedural methods for terrain modelling. Proceedings of the CASA Workshop on 3D Advanced Media in Gaming and Simulation (3AMIGAS). 1(1), 1-10.

Solid Angle. (2009). Bump Mapping [JPG]. Available: https://support.solidangle.com/download/attachments/1083474/feature_overrides_ignore_bump_off%20(1).jpg?api=v2. Last accessed 13th May 2014.

Sparx Systems. (2000). UML Tutorial. Available: http://www.sparxsystems.com/uml-tutorial.html. Last accessed 13th May 2014.

Turner, B. (2003). Real-Time Dynamic Level of Detail Terrain Rendering with ROAM. Available: http://www.gamasutra.com/view/feature/3188/. Last accessed 1st Dec 2013.

White, M. (2008). Real-Time Optimally Adapting Meshes: Terrain Visualization in Games. Available: http://www.hindawi.com/journals/ijcgt/2008/753584/. Last accessed 13th May 2014.

Wikibooks. (2009). Waterfall vs Iterative [JPG]. Available: http://upload.wikimedia.org/wikipedia/commons/d/d6/Waterfall_vs_iterative.JPG. Last accessed 13th May 2014.

Wikimedia. (2006). Heightmap [PNG]. Available: http://en.wikipedia.org/wiki/File:Heightmap.png. Last accessed 13th May 2014.

Wikimedia. (2006). Heightmap Rendered [PNG]. Available: http://en.wikipedia.org/wiki/File:Heightmap_rendered.png. Last accessed 13th May 2014.

Zucker, M. (2001). The Perlin noise math FAQ. Available: http://webstaff.itn.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html. Last accessed 13th May 2014.

Bibliography

Gerdelan, A. (2013). Cube Maps: Sky Boxes and Environment Mapping. Available: http://antongerdelan.net/opengl/cubemaps.html. Last accessed 1st Dec 2013.

Meiri, E. (2013). Tutorial 25 - SkyBox. Available: http://ogldev.atspace.co.uk/www/tutorial25/tutorial25.html. Last accessed 1st Dec 2013.

Nguyen, H. (2007). Chapter 1. Generating Complex Procedural Terrains Using the GPU. In: Cyril Zeller, Evan Hart, Ignacio Castaño Aguado, Kevin Bjorke, Kevin Myers, and Nolan Goodnight GPU Gems 3. London: Addison-Wesley. p56-59.

Nguyen, H. (2007). Chapter 16. Accurate Atmospheric Scattering. In: Cyril Zeller, Evan Hart, Ignacio Castaño Aguado, Kevin Bjorke, Kevin Myers, and Nolan Goodnight GPU Gems 3. London: Addison-Wesley. p300-350.

Planetside. (2013). Planetside. Available: http://planetside.co.uk/. Last accessed 1st Dec 2013.

Ulrich T. (2002). Rendering massive terrains using chunked level of detail control. SIGGRAPH Course Notes. 3 (5), 1-14.

Appendices

Screenshots of finished application

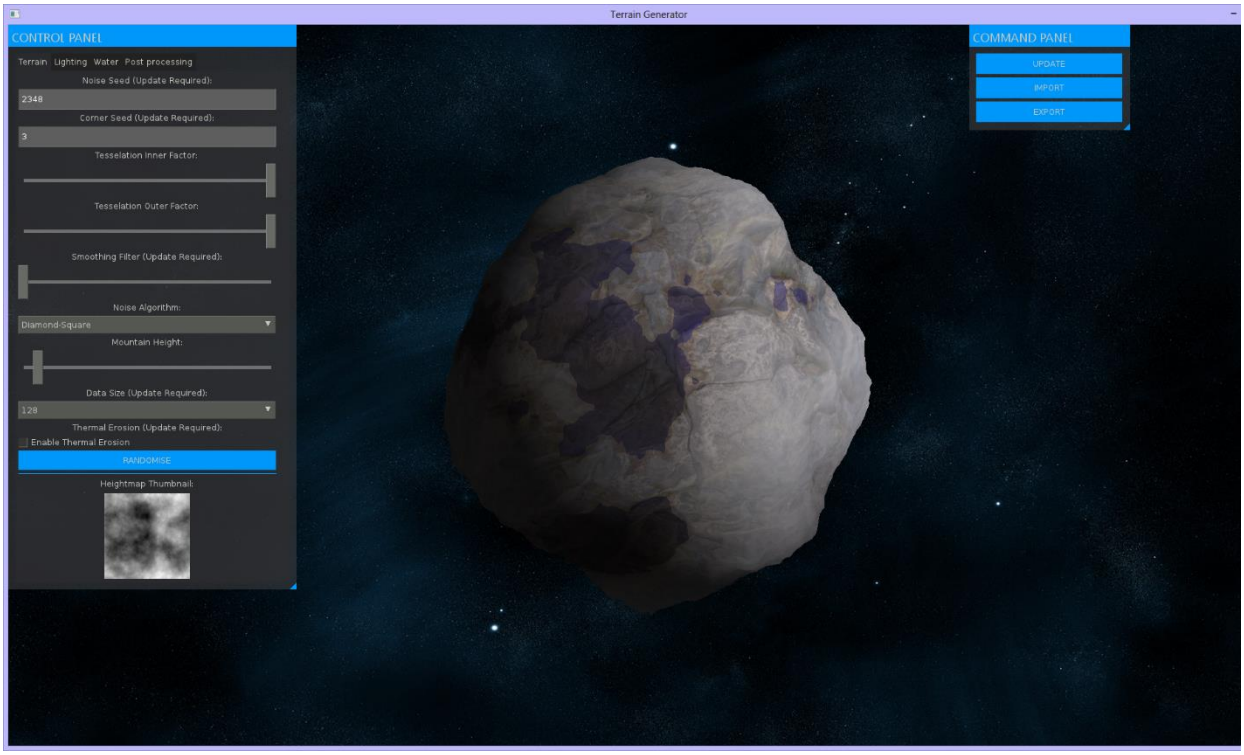

Fig. 55 *Complete Application* (2014)

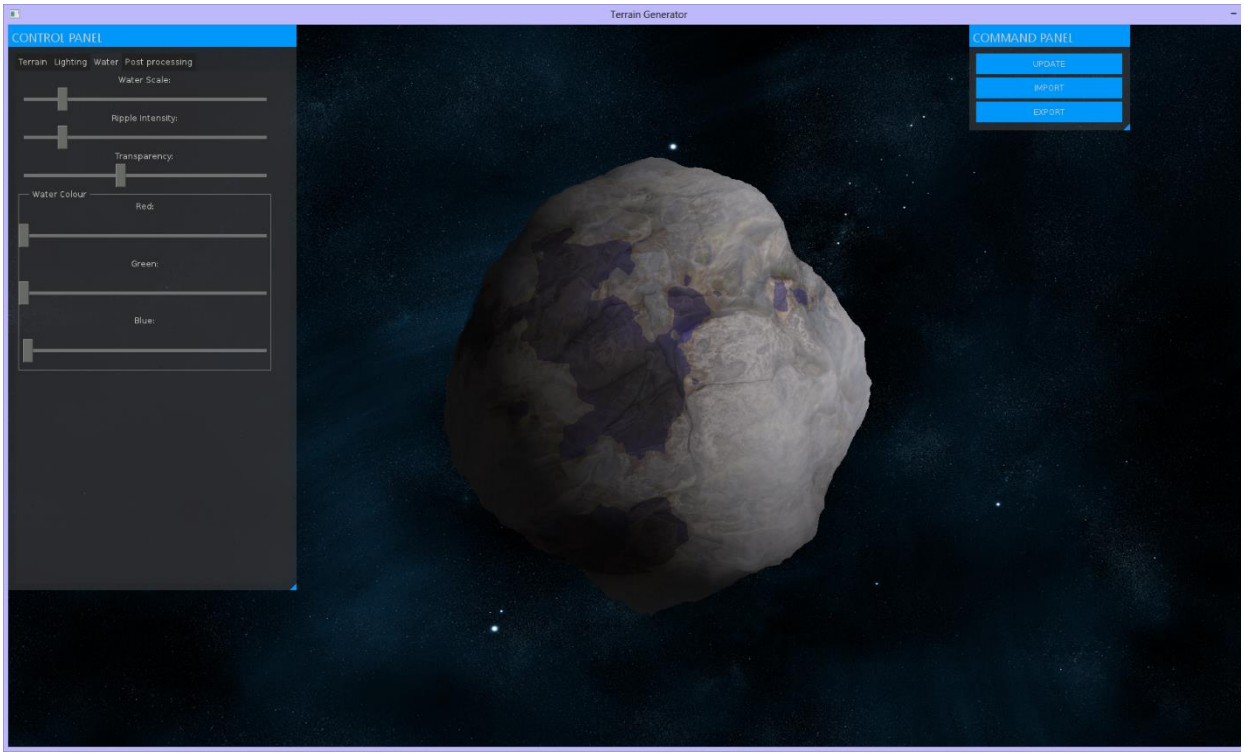

Fig. 56 *Water Toolbar* (2014)

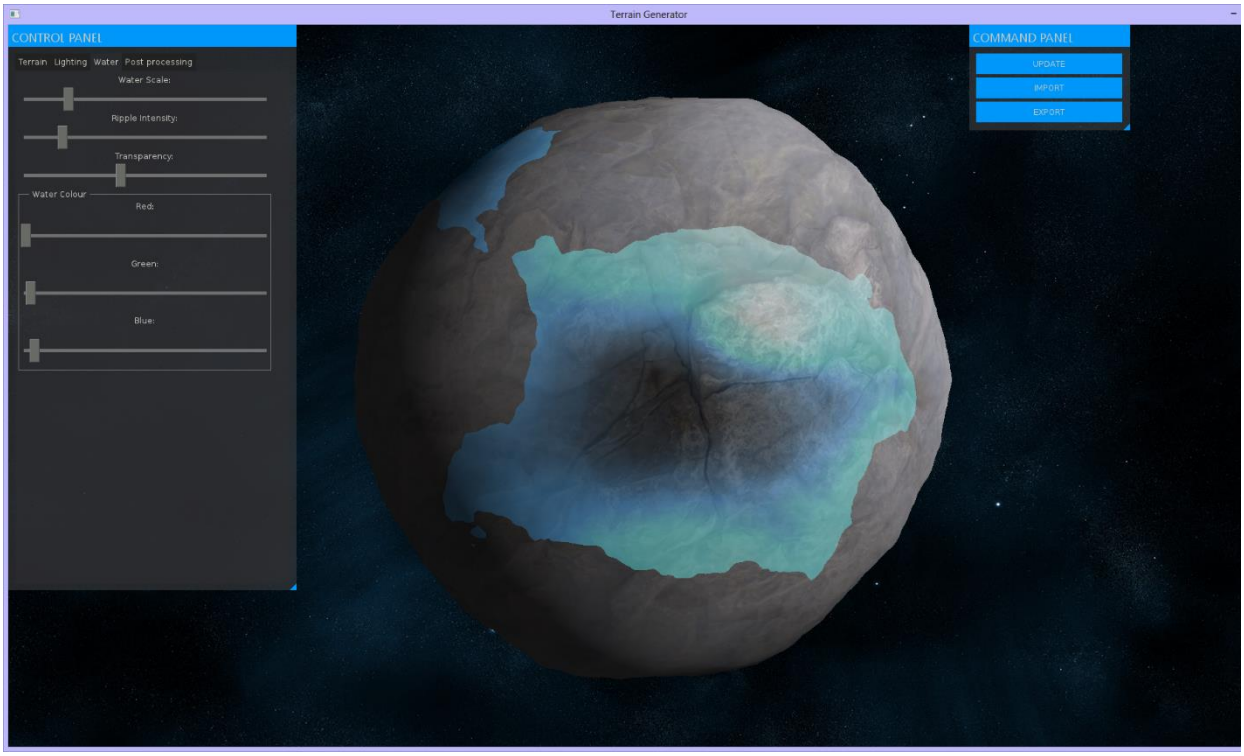

Fig. 57 *Water Colour Changing* (2014)

Questionnaire

Ease of use

Please circle were appropriate.

How difficult did you find it to navigate around the user interface?
Very easy | Easy | Average | Hard | Very hard

How difficult did you find it to understand what each feature did (technical jargon)?
Very easy | Easy | Average | Hard | Very hard

Overall please rate how difficult you found the application to use.
Very easy | Easy | Average | Hard | Very hard

Which section needs to be made simpler in terms of technical jargon?
Terrain | Lighting | Water | Post-processing

Which section needs to be made simpler in terms what the features do?
Terrain | Lighting | Water | Post-processing

Any other comments related to the applications ease of use?

Aesthetics

Please circle were appropriate.

How aesthetically appealing was the generated land mass of the planet (not including water and lighting effects)?
Very unappealing | Unappealing | Average | Appealing | Very appealing

How aesthetically appealing was the generated water?
Very unappealing | Unappealing | Average | Appealing | Very appealing

How aesthetically appealing was the lighting of the planet?
Very unappealing | Unappealing | Average | Appealing | Very appealing

Overall please rate how aesthetically appealing you found the generated planets to be.
Very unappealing | Unappealing | Average | Appealing | Very appealing

Any other comments related to the applications aesthetics?

Fig. 58 *User Testing Questionnaire (*2014)