INTERIM REPORT

"Algorithmic Procedural Generation of 3D Terrain"

Anna Hayhurst

14th November 2018

ah501@student.le.ac.uk

Aims and Objectives

Within the modern development of graphical systems and video games, manual creation of environments, their terrain, and contained models can be a labour-intensive process taking large amounts of time. In terms of business, this can lead to increased development costs and a longer production timeline.

The **primary aim** of this project is to address these issues by making use of the technique of procedural generation to create a 3D environment. The **secondary aim** is to make this an interactive user experience, where the user can control and customise the final generated scene produced by the system. This is a new aim that wasn't planned for at the beginning of the project. I have introduced it in order to ensure the final product is of higher value to end users, by ensuring it is designed with their inputs in mind.

Procedural generation concerns the use of algorithms to create data, rather than manually producing it on a case-by-case basis. In the context of computer graphics, the process can be used to quickly create a topology for environments in both 2D and 3D, as well as populate the scene with textures and models. Another benefit is the reduction of file sizes when compared with the same amount of manually created content.

This technique has been successfully applied in many situations, most noticeably in video games such as *The Elder Scrolls II: Daggerfall, No Man's Sky* and more generally in the roguelike genre. This forms part of my motivation for this project, as a potential application of my work would be in video game development; the system would be able to produce randomised maps that could be further built on with interactivity and gameplay.

The primary algorithm chosen is Perlin noise, a popular algorithm within procedural generation. It produces a coherent noise map from which varied heights and terrain can be generated for a scene, with low computational complexity in three dimensions.

The software system is being developed in OpenGL 3.3, using C++. Considering this, the **key objectives** are:

- To produce a C++ implementation of Perlin noise,
- To produce a graphical system within OpenGL,
- To link the Perlin noise values to the system such that a procedurally generated scene is created.

Another algorithm to be implemented is Worley noise. This noise will be utilised to procedurally generate texture for the scene; it is ideal to produce simple textures such as stone with very little overhead. It will be clearly observable if this objective is achieved, as you will see a texture that varies slightly with each run.

Expanding on core functionality, there is potential to make use of height ranges to produce a more interesting output scene. The following objectives represent further features that will be built on top of the core system:

- Divide height values from Perlin noise into subsets,
- Create 'biomes' according to the different subsets of values, e.g. mountains, grasslands.

This will add visual variety without first having to load in and place models. Of course, model loading is a natural extension to creating variety. Therefore, this will be another stretch goal, with the objective being to add environmental assets such as trees and rocks at sensible positions in the scene.

A final **key objective**, expanding on the secondary aim, would be to allow a higher level of user interaction by prompting the user to enter a seed value upon application start-up, which would then be used to seed the noise generating the environment. The user will also be able to control parameters such as window dimensions, octave count and persistence.

Considering the written work required for this project, my broad aims remain to provide a good overview of the algorithms I am using, including their origin, their strengths and a description of how they work, as well as why I chose these algorithms.

Survey of Literature

Most of the research undertaken for this project has concerned OpenGL – its syntax, style, and best practices. My primary source for learning the framework has been the website *Learn OpenGL* [1], which provides clear and concise tutorials for key features, grouped by use and complexity. It has proved to be an excellent learning tool that explains the high-level use of OpenGL in graphics programming, though it doesn't go into much low-level detail.

To learn about low-level features, I made use of the *OpenGL Programming Guide* [2]. This book provides good advice on programming style and a comprehensive directory of GL and GLSL values, but

unfortunately only covers OpenGL 2.0. Hence, I also made use of the *OpenGL Wiki* [5] to ensure I knew about modern features.

Further research was required to understand the auxiliary libraries and languages. For this purpose, the eBook *Instant GLEW* [4] was utilised, with *The Book of Shaders* [6] as a source for shader programming in GLSL. Finally, the vector and matrix mathematics required to create 3D graphics effectively were revised from *Mathematics for Computer Graphics* [3].

A significant time investment was also made in researching and understanding the Perlin noise algorithm. The original code itself [7] provides little explanation as to its procedures and mathematics, so I turned to external references. Biagioli's writeup [8] was combined with Gustavson's article [9] to give two perspectives on the implementation.

Current research concerns procedural generation itself. Previously, I made use of Archer's article [10] comparing the efficiency and complexity of different noise types in order to select Perlin noise as the generator. Now, I am using the *PCG Wiki* [11] to perform further research on how procedural systems work best. Furthermore, in Santamaría-Ibirika et al.'s article [12], I have found a comprehensive set of performance metrics for such systems that I hope to use to evaluate my project.

In future, I will need further reading in order to implement additional features, particularly the Worley noise algorithm. I believe my current sources will be sufficient for the algorithm – [6] has an extended shader tutorial for cell noise, [10] contains an overview of the algorithm's procedure, and it is also mentioned as an article within [11].

Architecture, Algorithms and Design of the Software System

i. System architecture

The core of this system is built in OpenGL 3.3. OpenGL provides a software interface through which we can interact with hardware and manipulate the rendering pipeline used by the Graphics Processing Unit (GPU). Version 3.3 was selected as it provides a good balance between newer features and compatibility with most modern hardware. This compatibility allows for better portability.

Figure 3.1 gives an outline of the OpenGL rendering pipeline and how this is reflected in the system, omitting optional steps not implemented for this project.

159015693
ah501@student.le.ac.uk

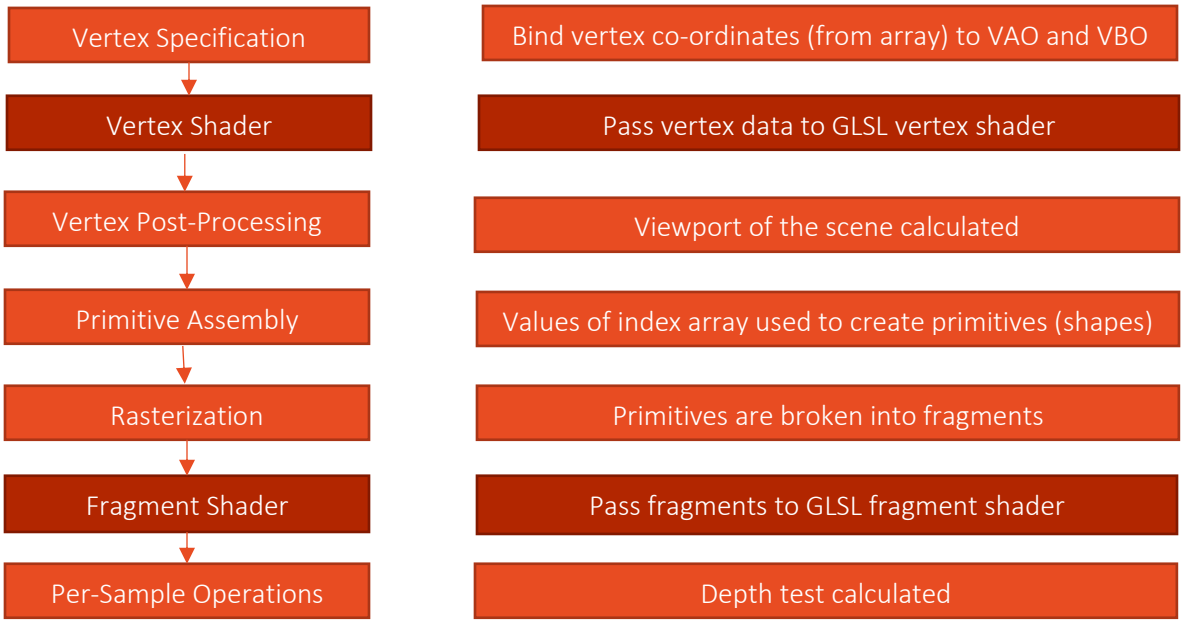

Figure 3.1 – OpenGL pipeline steps and a summary of which elements of these occur in this system

Currently, classes for the creation of OpenGL elements implemented are Window, Camera, Shader, Mesh and Texture. These classes are used within the main executable to follow the flow of the pipeline and render the final scene. In addition to these, classes representing light (with a derived class for directional light), material properties, and three-dimensional models will be present in the final project.

Upon running the executable, an instance of Window is created. The initialisation process for this window makes calls to the Graphics Library Framework (GLFW) and sets up properties for the OpenGL Extension Wrangler Library (GLEW). The combination of these steps creates an empty frame for OpenGL to perform rendering in.

At this time, the camera is also created with an initial direction vector, pitch and yaw angles, and movement speed. Scene perspective is set by suppling the field of view (FOV) and aspect ratio, along with the minimum and maximum render distances.

Calls are then made to create shaders to be applied to every vertex. This is performed with Shader class functions, which load in OpenGL Shader Language (GLSL) files and attach their data to the OpenGL program. After this, meshes and textures can be created, ready for rendering.

Meshes are created from arrays of vertex and index values, representing their vertex co-ordinates and the order in which these vertices are connected to create primitive shapes. Through Mesh functions, the Vertex Array Object (VAO), Vertex Buffer Object (VBO), and Index Buffer Object (IBO) are bound to the system, and their properties defined.

Textures are created from image files, utilising functions from the stb_image.h library. Image data is bound to the first available texture unit. Settings for how the image is mapped to the mesh are defined, including whether it should be tiled.

The shaders, meshes and textures are all stored in dynamic memory within their own vectors. This prevents the objects from going out of scope and means they can easily be accessed when they are needed for the drawing process. To prevent memory leaks, these objects are deleted explicitly before system exit.

With all objects in place and configured correctly, the main rendering loop can begin. The following are repeated to complete the rendering process:

- Calculate the time elapsed since the previous loop using system time.
- Listen for user interaction (event polling).
- Bind camera's mouse and keyboard controls to the window.
- Render the map – set uniform values for shaders, reset viewport, clear out colour information.
- Render the objects – apply textures to meshes, set transformations, draw meshes.
- Swap the rendered scene with the displayed scene.

This continues until the window is set to close, either manually or triggered through pressing the escape key.

ii. Perlin noise algorithm

Ken Perlin's noise algorithm produces coherent noise values; this is pseudo-random, as opposed to random 'white' noise. The key properties of such noise are that providing the same input value will produce the same output, and that changing input slightly will change output slightly, leading to areas of visual similarity. By contrast, when there is a large change in input, the change in output is random. Perlin's algorithm works to fulfil these requirements.

The algorithm takes an input of cartesian co-ordinates – in our case, x, y, and z for three-dimensional noise – and produces a single output in the range -1.0 to 1.0. It makes use of a fixed collection of values between 0 and 255, named permutations, to generate the pseudo-random result. For a fixed set of permutations, using the same input co-ordinates will always produce the same output value. There is, of course, the option to generate a new set of permutations and thus generate different output values.

Overall, the process consists of several key steps, an overview of which can be seen in Figure 3.2:

```
INPUT x, y, z

   • Create UNIT CUBE of x, y, z
   • Find RELATIVE position of x, y, z in cube
   • FADE the relative values
   • HASH the unit cube values
   • Use hashed values to select random GRADIENT
   • Calculate DOT PRODUCT between gradients and x, y, z
   • Perform LINEAR INTERPOLATION between relative
     values and the dot products

OUTPUT n
```

Figure 3.2 – The steps of calculating Perlin noise

The key step to note with regards to this algorithm is the production of the unit cube. The values are used in most of the subsequent steps, most importantly to select the gradient vector by accessing a permutation and hashing that value.

There are 8 possible gradient vectors when producing the noise. These are three dimensional vectors where values are either -1, 0, or 1. The chosen vector is used in a dot product calculation with x, y, and z, and finally linear interpolation is performed between these values and the relative values within the unit cube. A combination of linear interpolation calculations produces the final output.

The algorithm was chosen for numerous reasons. For one, according to Archer's report (2011), Perlin noise is described as having a good balance of computational speed, memory usage and output quality, without being as complex as similar algorithms – namely Simplex noise. He also states that, at the time, it was still considered the industry standard.

In terms of procedural generation itself, Perlin noise is classified as an ontogenetic algorithm, meaning it emulates the result of natural processes without showing intermediate steps (*Ontogenetic* 2009). This is

a contrast to teleological algorithms, which behave more like simulations. The focus on the result makes Perlin noise ideal for a real-time system, and keeps hardware requirements low, leading to increased compatibility.

iii. Software design techniques

Until now, I have primarily followed a Waterfall style approach of development, where features are completed in a linear fashion, according to the layout of the Gantt chart and the dependencies needed for each class. While this hasn't caused issues in the development process thus far, it does mean that elements required for features to be considered fully complete are missing, such as unit testing.

To improve, my aim is to adopt a more Agile method of development, where features are fully complete and deliverable before moving on to the next. For the rest of this semester, I will complete the remaining planned features as normal and implement the missing test cases. Then, with the beginning of the new semester, I will begin to work in this new style. By doing so I hope to improve the stability of my code and make faster progress, while gaining more experience with the methodology.

Prototype Description

The produced prototype of the software system covers two key elements – the fundamental OpenGL system from which the eventual final scene will be produced, and the implementation and output testing of Perlin noise. The current state of the software displays key elements of the final system as well as a minor combination of graphics and the primary algorithm, providing a solid platform to further combine the two as development progresses.

i. Perlin noise prototype

In this prototype, the C++ implementation of Perlin noise is fully implemented. The original algorithm with Perlin's permutation values is correct and functioning, and in addition to this it is possible to seed a new version of permutations using an integer value. This fulfils several of the key planned requirements.

On top of the original functionality of the noise function, another adapted method of generating noise using octave and persistence values was implemented. This additional function will enable the end system to produce more visually interesting maps from noise with a greater degree of variety. As stated in the code itself, this was implemented with the assistance of [3].

The first challenge presented in implementing the algorithm was the difference in flexibility between arrays in Java and C++. For the purpose of producing the standard collection of permutations, a static C++ array would be fit for purpose. However, when producing a randomized permutations array, the requirement of having a fixed size known at compile time presented a major issue – how could I produce a 512-value collection, where the latter 256 values are a repeat of the first, without overcomplicating the code with this restriction?

The solution I chose was to use a vector to store the values instead. By sacrificing a small amount of speed, vectors allowed for easy resizing, in turn simplifying the overall code and making it easier to use the `std::default_random_engine`, which can target the beginning and end of vectors to inject its shuffled values.

A further challenge was truly understanding the algorithm itself. I didn't want to lift the implementation from Perlin's work and reproduce it verbatim, as this would lead to no learning and make any potential errors difficult to identify. As mentioned in the literature review, I utilised several sources to learn the background mathematics and structure of the algorithm. I could then apply this knowledge to understand the steps of the algorithm and separate them out where appropriate to aid code comprehension.

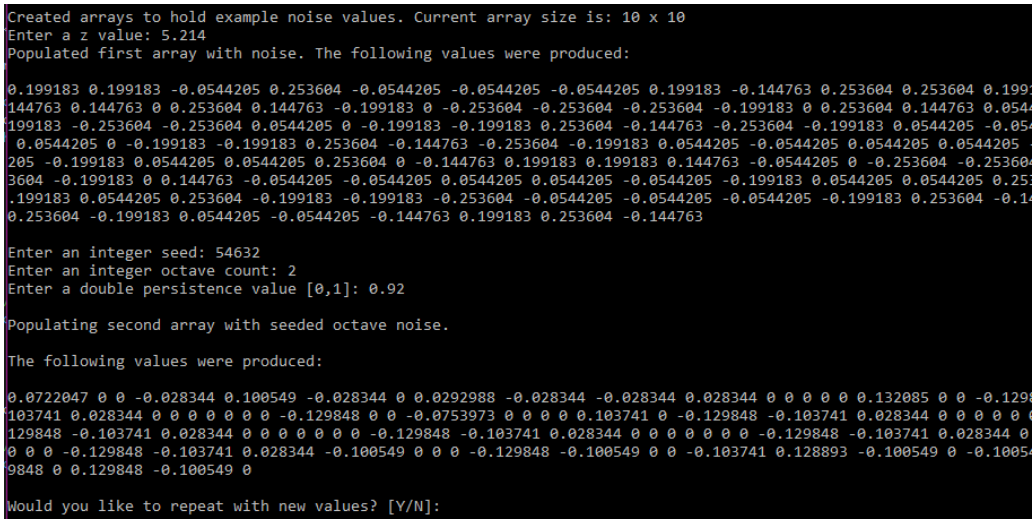

Figure 4.1 - Example output from helper

For the purpose of demonstrating the prototype, a helper class was produced to display various outputs of Perlin noise, with the ability to input z values, seed values as well as the octave count and persistence. This is displayed in the console and can be accessed (and looped through for as long as desired) once the

OpenGL window is set to close. Some simple input validation is performed to ensure the noise function calculates successfully.

Though automated testing remains to be written, manual desk testing has been completed. Values were compared against those from the original Java implementation and consistently matched.

ii. OpenGL system prototype

The current system implements the core elements of an OpenGL system, namely the window used to display the rendered scene, a camera, and classes implementing meshes and shaders. Textures are also fully implemented and can handle both JPG and PNG file extensions.

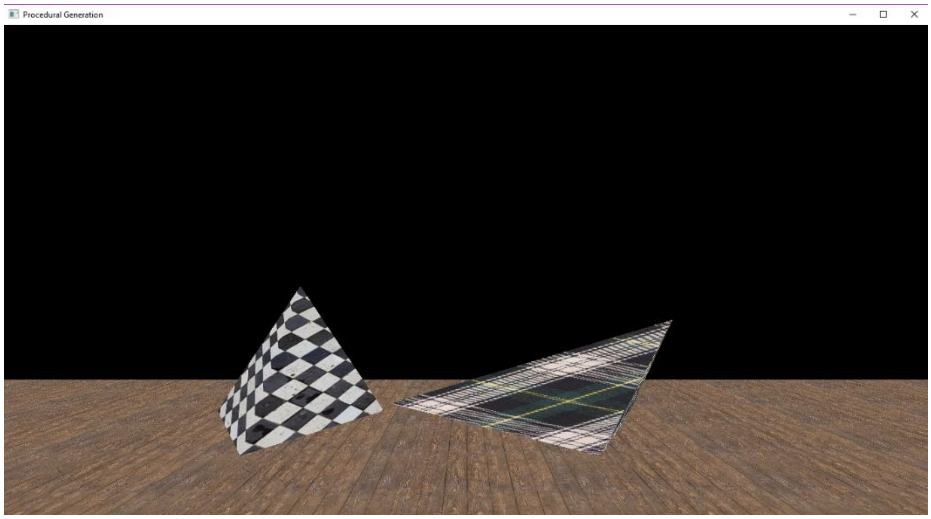

Figure 4.2 – Example run of 3D scene. The right-hand mesh has been altered by noise values.

To best display all current features, a simple scene has been rendered. Three textured meshes have been generated, two of which are static and appear identical each run, with the third involving Perlin noise. The static meshes have vertices made from fixed float values, whereas the noise adjusted mesh has the result of `Perlin::noise()` added, as calculated from its original x, y, and z values. This is seeded by the current system time and thus produces a different mesh upon each run. A slow real-time rotation transformation was included to properly show the adjustments made.

There is currently a single camera that responds to both mouse and keyboard input. To assist with controlling the camera, the cursor has been locked to the window and disabled. The user can rotate the camera by moving the mouse, and move the camera with either W, A, S and D or the arrow keys. The speed factors for the camera have been set to values that allow navigation at a sensible pace.

The primary challenge faced while producing the core classes has been the difficulty of debugging OpenGL processes. When an error occurs in the OpenGL pipeline, it can be difficult to track, even with added validation. Often the outcome is that the scene fails to render with no console feedback, leading to lengthy manual debugging. I have attempted to minimise this by properly encapsulating methods and testing output frequently to isolate the source of issues as they arise.

As my experience with the framework grows, it is becoming easier to identify what is causing rendering issues. However, I recognise that manually debugging is both time consuming and inefficient; hence, moving forward, my aim is to introduce methods such as unit testing and assertions in order to more quickly identify problem areas and create a more stable system.

Planning and Timescales

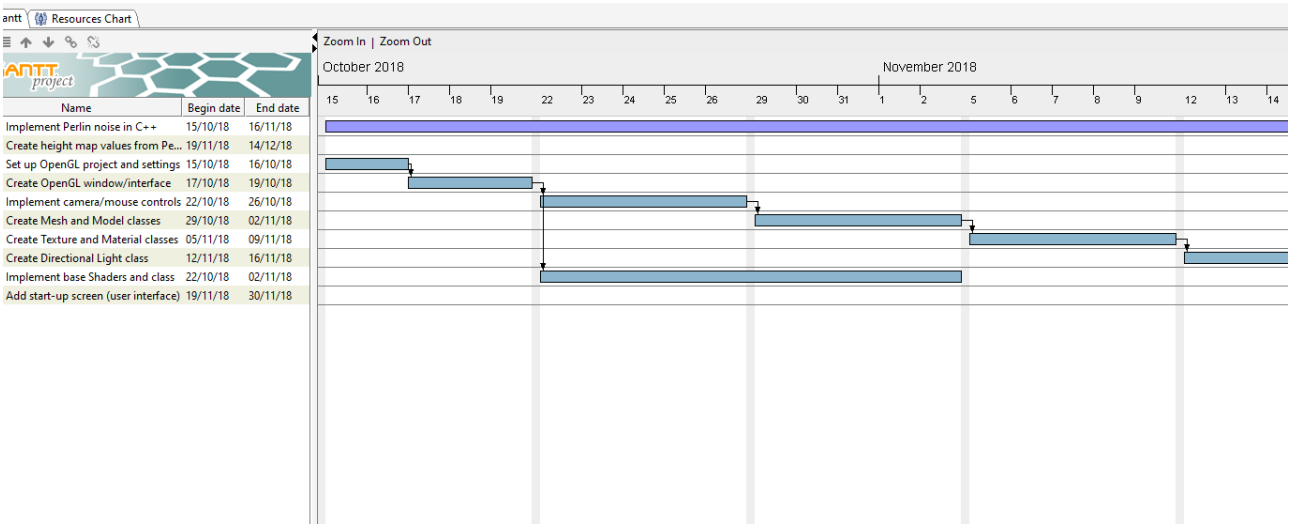

Figure 5.1 – Partial view of the initial Gantt chart (can be found alongside report in /docs)

My main goal for this semester has been to implement Perlin noise fully and accurately, and then to take this implementation and be able to generate an appropriate height map. As can be seen in the diagram above, I had allocated a month to implementing the noise algorithm. A similar amount of time was assigned to creating the height map, which took the timescale to mid-December.

The implementation of Perlin noise did take about as long as was allocated, although there was still some final debugging that had to be completed after the end date. I had allocated such a large amount of time as the algorithm is complex to implement and required extensive research in order to truly understand. Bearing in mind the difficulty I've experienced implementing this element, I have reworked the objective of producing 3D maps using noise into smaller sub-objectives that better reflect the

direction of development in the current system. To reflect this an updated Gantt chart can be found alongside this report.

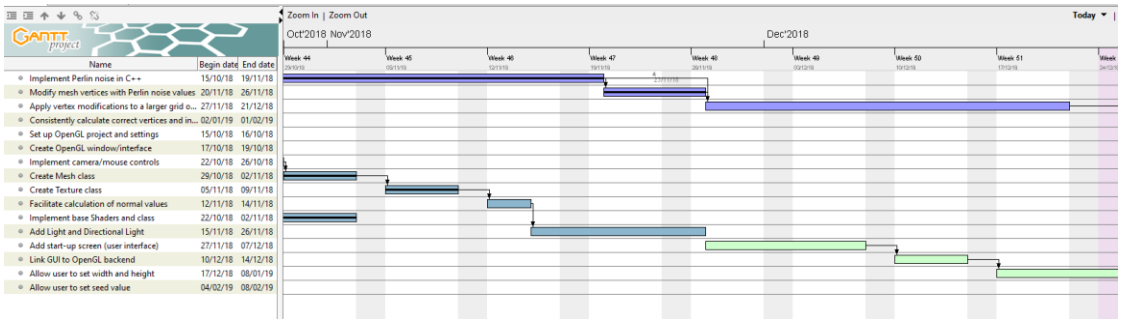

Figure 5.2 – Overview of updated Gantt chart (found in /docs/2_interim_report)

While implementing the algorithm I have been laying the groundwork for the OpenGL system so that it is prepared for when I start combining it with the eventual height map. I had planned to have more elements of the system complete by this point in the project. The implementation of the fundamentals took longer to correctly design and code than I accounted for.

Examining the initial chart, I have completed about 80% of the workload planned to this point. Several requirements as specified in the plan have been fulfilled, namely 1, 1.1, 1.1.1 and 1.3 concerning Perlin noise as well as 3.1 and 3.2 concerning scene display. Several others are partially fulfilled.

Considering all this, to improve future workflow I have extended and reordered the timescales for the remaining key elements. During this development period I have realised it was unreasonable to expect to write this amount of software to a high standard simultaneously. Importantly, the layering of three critical tasks in the same timeframe created stress and indecisiveness about what to complete first, causing pace and quality to suffer.

In the second semester, the plan remains to create the procedurally generated scene and augment it with additional features. In the case that the base procedure for producing the scene is not complete by the beginning of second term, further time will naturally be allocated for its development. The process of creating a quality 3D map will be the largest and most complex section of the project, with the highest potential for delay and error. Hence during this semester, as an aside, I am preparing and researching for this future work.

Bibliography

[1] de Vries, J. *Learn OpenGL*. Available at: https://learnopengl.com/ (Accessed: June 30th 2018).

[2] Shreiner, D., Woo, M., Neider, J., Davis, T. (2006), *OpenGL programming guide: the official guide to learning OpenGL, version 2.* 5th edn. Upper Saddle River, N.J.; London: Addison-Wesley.

[3] Vince, J. (2006) *Mathematics for Computer Graphics*. 2nd edn. London: London: Springer London.

[4] Nanjappa, A. (2013) *Instant GLEW.* Olton: Packt Publishing Ltd.

[5] *OpenGL Wiki*. Available at: https://www.khronos.org/opengl/wiki/Main_Page (Accessed: 1st October 2018).

[6] Gonzalez Vivo, P. & Lowe, J. *The Book of Shaders*. Available at: https://thebookofshaders.com/ (Accessed: 28th October 2018).

[7] Perlin, K. (2002). *Improved Noise reference implementation*. Available at: https://mrl.nyu.edu/~perlin/noise/ (Accessed 1st Oct. 2018).

[8] Biagioli, A. (2014). *Understanding Perlin Noise*. Available at: https://flafla2.github.io/2014/08/09/perlinnoise.html (Accessed 10th Oct. 2018).

[9] Gustavson, S. (2005) *Simplex noise demystified*, Linköpig University.

[10] Archer, T. (2011) *'Procedurally Generating Terrain'*, MIC Symposium.

[11] *Ontogenetic.* (2009) Available at: http://pcg.wikidot.com/pcg-algorithm:ontogenetic (Accessed: 2nd December 2018).

[12] Santamaría-Ibirika, A., Cantero, X., Salazar, M., Devesa, J., Santos, I., Huerta, S. and Bringas, P. (2014) *'Procedural approach to volumetric terrain generation'*, The Visual Computer; International Journal of Computer Graphics, 30(9), pp. 997-1007