Algorithmic Procedural Generation of 3D Terrain

University of Leicester - Department of Informatics
CO3015 – *Computer Science Project*
Computer Science BSc.
Dissertation

Anna Hayhurst

1st May 2019

Table of Contents

Declaration

All sentences or passages quoted in this report, or computer code of any form whatsoever used and/or submitted at any stages, which are taken from other people's work have been specifically acknowledged by clear citation of the source, specifying author, work, date and page(s).

Any part of my own written work, or software coding, which is substantially based upon other people's work, is duly accompanied by clear citation of the source, specifying author, work, date and page(s).

I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this module and the degree examination as a whole.

Name: Anna Hayhurst

Signed: 

Date: 24th April 2019

List of abbreviations

- **API –** Application Programming Interface
- **PCG –** Procedural Content Generation
- **GPU** – Graphics Processing Unit
- **CPU** – Central Processing Unit
- **VRAM** – Video RAM
- **GLSL** – OpenGL Shader Language
- **GLEW** – OpenGL Extension Wrangler
- **GLFW** – Graphics Library Framework
- **VAO** – Vertex Array Object
- **VBO** – Vertex Buffer Object
- **IBO** – Index Buffer Object
- **FPS** – Frames Per Second
- **RNG** – Random Number Generator
- **GUI** – Graphical User Interface
- **FOV** – Field of View angle

Glossary of common terms

Term	Defined here as...
Render	Producing a 3D image or images based upon pre-existing data.
Scene	The result of rendering several objects or elements in combination, which can be viewed and navigated through.
Generator	A shortened term for the procedural content generator implemented in this project.
Engine	A group of classes that, when used together, facilitate the rendering of graphics by interacting with the graphical pipeline.

1. Abstract

This project explores the concepts of coherent noise in order to develop a procedural content generator. Procedural generation and its applications are examined, and types of coherent noise are compared and contrasted.

An OpenGL system is built from first principles and is then combined with the Perlin noise algorithm in order to generate pseudorandom 3D terrain. Control over parameters affecting the result is given to the user, such that a potentially infinite set of results can be produced. The result is a program that demonstrates the potential of coherent noise, alongside an engine that can be reused and easily built upon further. Consideration is given to how the procedural generator scores against modern criteria.

2. Introduction

Development of computer graphics, and by extension video games, can be a costly and lengthy process. This is particularly true with regards to the creation of terrain and maps, as the design and manufacture of convincing environments requires both creative vision and a solid graphical toolset to implement this vision.

The primary aim of this project is to utilise OpenGL 3.3 to create an efficient, functional and reusable engine, and from this foundation develop a procedural generator that will greatly simplify and automate the creation of 3D terrain. The generator will be built using Ken Perlin's coherent noise algorithm.

The hope is to provide game designers with a wide variety of results from which they can take inspiration for their own work, while at the same time demonstrating the possibilities of utilising coherent noise. To further this aim, a focus is made on consideration of UX. Through an easy to navigate GUI, the end user will have control over parameters affecting the result. This removes the need to understand the intricacies of procedural generation.

Similarly, the engine itself will be made accessible to users with programming knowledge but little to no experience with graphics APIs. This will be achieved through proper encapsulation of code that works with the low-level graphics pipeline, so that the user never has to make direct calls to OpenGL.

2.1 – Objectives

With the aims of the project in mind, the specific objectives to be achieved are –

- Implement a standalone OpenGL engine, with the capability to render meshes with texture and lighting.
- Adapt and augment Perlin's noise algorithm, adding additional options and functions.
- Design a procedural generator using Perlin noise in tandem with the engine.
- Add further design options, such as biomes.
- Implement an easy to use GUI that allows the user to influence the result of the procedural generator.

3. Background

Here, an overview is provided of the necessary theory and background required to understand the implementation of the project.

3.1 – Procedural Generation

Procedural generation is the process of producing data algorithmically with an element of randomness, as opposed to producing it manually. In this project, we specifically consider procedural content generation (PCG), which encompasses the creation of content usable in video games. [1]

In the context of computer graphics, the principles of PCG can be applied to produce 3D terrain, as is the objective here. One or more coherent noise algorithms (see section 3.2) can be combined to produce a realistic landscape. The benefits of this process are numerous, and not limited to –

- A potentially infinite set of results, achievable with small changes to a few parameters.
- A natural increase in production speed of landscapes and maps due to the ease of producing these results.
- A reduction in file sizes for applications using the technique, as static storage of map data is not required when it can be reproduced from a given set of parameters.

However, there are also drawbacks to automating generation. The reliance on noise means the resulting terrain may lack fine detail or desired landscape features, which may need to be then added manually after the fact. One must also consider the fact that the time and effort required to develop the PCG engine may not always justify the benefits.

PCG techniques have been applied successfully to video games for decades, capitalising on the speed of map production to allow developers to focus more on gameplay aspects. As early as 1980, the game *Rogue* applied the concept to generate 2D dungeon maps, becoming influential enough to spawn an entire genre, 'roguelike', using the same procedures.

In more recent years, perhaps the most successful example of PCG is *Minecraft* (2011), which used coherent noise to create not only terrain but a sprawling underground system of tunnels and caves. [2] Incidentally, it makes use of Perlin noise, as is used in this project.

The current state of the art of procedural generation is moving towards more complicated implementations of noise algorithms, which reduce the computational complexity as the number of dimensions increase. A prime example of this is simplex noise. Advancements are also being made to combine aspects of artificial intelligence with the results of procedural generation, removing the need

for human intervention to make improvements to randomly generated maps and other content. Specifically, machine learning is being used as a method of examining existing data and generating new, feasible content, as demonstrated in the study of Giacomello et al. [3]

The scope of this project does not cover these recent advancements, but does cover the relatively modern technique of 3D terrain generation as opposed to 2D map layout production.

3.2 – Coherent Noise Algorithms

White noise can be defined as any set of values within the bounds $[-1, 1]$ produced by a random number generator – the overall appearance of the noise, in turn, is also random, with no discernible features. By contrast, coherent noise is pseudorandom – there is a degree of control over the final result, which causes the noise to have areas of similar values, grouped together.

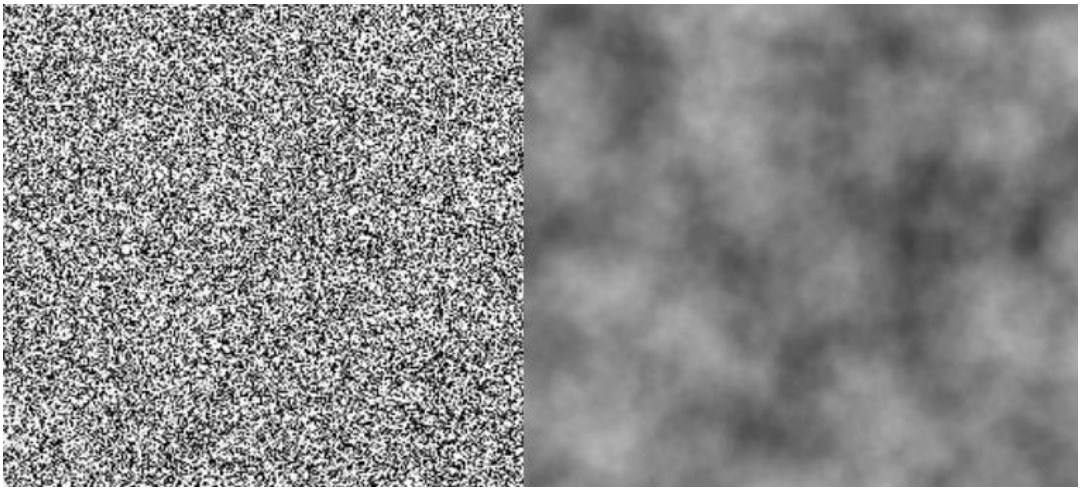


Figure 3.2.1 *– A side-by-side comparison of white noise (static) and coherent noise (produced by a variation on Perlin's algorithm). Both examples were produced using Photoshop tools.*

A coherent noise algorithm is any algorithm that takes one or more numerical values as input and produces a single coherent noise value as a result. Broadly, the output of the algorithms obeys the following principles [4] –

- The same input value(s) will always produce the same output.
- Small variations in input will produce small variations in the output.
- Large variations in input will produce random variations in the output.

Proper application of these concepts ensures that the overall result has areas of visual cohesion – two points that are close together will have similar values, while two points that are arbitrarily far apart will have a random difference in value.

3.2.1 - Perlin Noise

The coherent noise algorithm implemented in this project is Ken Perlin's "Improved Noise", a variation on his original algorithm, released in 2002 [5]. It became an industry standard for a variety of purposes, the most pertinent case being procedural generation. Both 2D and 3D variations have been created for the software system, but 2D noise will be discussed here.

In terms of PCG, this algorithm is considered to be ontogenetic, meaning it emulates the result of natural processes without explicitly showing the intermediate steps [6]. What we see is the final result of this emulation, as opposed to a continuous simulation of the events leading up to it.

Perlin noise applies several key stages to the cartesian co-ordinate input to produce its final output, an overview of which can be seen in Figure 3.2.2 –

Input: x, y

- Take FLOOR values of (x, y) to create UNIT SQUARE -> (X, Y)
- Find RELATIVE position of (x, y) within square
- Apply FADE function to (x, y) -> (u, v)
- HASH the vertices of the unit square
- Use hashed values to select pseudorandom GRADIENT
- Calculate DOT PRODUCT between gradients and (x, y)
- LINEAR INTERPOLATION between (u, v) and dot products -> n

Output: n

Figure 3.2.2 *– The process of calculating Perlin noise, with emphasis given to key points and helper functions.*

Fundamental to the process is the production of the unit square. This is a 1 by 1 structure that surrounds the input values, within which we locate the relative positions of (x, y). The corners of the unit square and these relative positions are key to producing the pseudorandom output.

A fade function is applied to the relative positions, the purpose of which is to bring them closer to the integer values bounding the unit square. Doing so helps to fulfil the requirement that small changes in input produce small changes in output, as it eases the results for similar (x, y) pairs closer together. Perlin's specific fade equation is a fifth-order polynomial, $6t^5 - 14t^4 + 10t^3$.

The pseudorandom element of the algorithm is introduced when the vertices of the unit square are hashed. This is achieved by using the vertex values to access the corresponding element of a permutation lookup table with the following properties:

- The table contains 512 values. The values are between 0 and 255, inclusive.

- Each individual value appears exactly twice.
- The first 256 values are shuffled in an arbitrary order, and the second 256 values are an exact copy of this first set.

The values retrieved form gradient vectors, which can point in any direction from the given vertex. The gradients produced from two identical inputs will also be identical, as the same permutation value will be accessed on both occasions. Shuffling the values in the permutations array allows different results to be achieved.

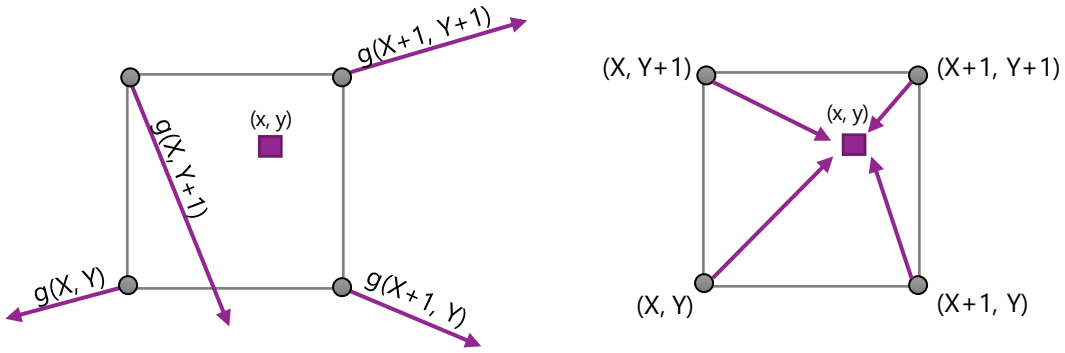


Figure 3.2.3 - *The vectors involved in the dot products, represented by arrows. The gradients on the left-hand side have a dot product calculated with the corresponding vector on the right-hand side. This diagram was adapted from [7].*

The dot product between each gradient and a vector from that vertex to (x, y) is then calculated. Through this, we determine the overall influence of each gradient on the final value – a larger dot product result means the gradient affects the output more. Perlin achieves this using bitwise comparisons between the gradient and direction vector.

The final step of the algorithm is to perform linear interpolation between the faded co-ordinate values and the results of these dot product calculations. From this, a scalar noise value is produced. In the case of 2D noise, one could consider this value to be a z value corresponding to the original (x, y) pair, allowing us to produce a 3D environment from 2D input in the form of a heightmap.

3.2.2 – Perlin in comparison to other algorithms

Perlin noise was chosen over other algorithms for several reasons. A key source in making the decision was Archer's report, '*Procedurally Generating Terrain*' [8], which contrasts and compares the most well-known coherent noise algorithms, evaluating their performance, quality of results, and difficulty to implement.

The algorithm is described in the report as producing realistic results with low memory overhead while being moderately difficult to implement. Its primary drawback is that it is fairly slow to run. Simpler to

implement algorithms, such as value noise and mid-point displacement, run much more quickly, but in turn do not produce convincing results for this purpose.

Conversely, a more recent algorithm, simplex noise – itself a redevelopment and improvement on Perlin noise – produces an even higher quality output, with a faster running time. Whereas Perlin noise has a time complexity of $O(n \times 2^n)$, where n is the number of dimensions, simplex noise is only $O(n^2)$ [8].

The reason simplex noise wasn't chosen over Perlin, despite these advantages, was primarily due to its extreme difficulty to implement in comparison. Archer describes it as "very difficult to understand, much less implement", remarking that it is equally difficult to identify where errors are made. This relative difficulty means there is also much less documentation on its implementation, and thus I evaluated that it was infeasible for the scope of the project.

Overall, Perlin noise combines a balance of convincing results, small overhead, and acceptable difficulty to produce. In 2D, the exponential element of its time complexity is a minor concern, so while it may run more slowly than comparable algorithms, it still runs in an acceptable timespan.

3.3 – OpenGL

OpenGL is an API that allows us to interact with and send instructions to a GPU. It allows a high degree of control over the production of 2D and 3D vector graphics on a computer system, especially when compared to higher level graphical engines such as Unity and Unreal, as we can directly interact with the graphics pipeline on a low level.

With OpenGL, a graphical program is built and developed almost entirely manually, with the use of some key open source libraries for maximising compatibility and mathematical functions. It was chosen for the project for this reason – each element can be customised to purpose without unnecessary overhead. In addition, it is naturally cross-platform, as it remains the industry standard and most widely adopted graphics API [9].

3.3.1 – Version 3.3

The specific version of the API chosen is OpenGL 3.3, released in March 2010. This is the third edition of OpenGL 3, the major version considered to be the beginning of the 'modern' form of the API. Version 3 was the first edition to allow context creation to call a specific version of OpenGL, as opposed to having the highest version available to the GPU automatically selected. This change allowed an overhaul of features and the deprecation of many outdated functions.

The advantage of using version 3.3 specifically comes from its lasting prevalence in contemporary software and the comprehensive documentation surrounding it as a result. Its age also provides a wider compatibility for the project – it is highly unlikely that a system will be incompatible, as any relatively modern GPU will be running the later editions of version 3 and above.

3.3.2 – The Rendering Pipeline

A rendering pipeline is a model describing the steps that must be taken for graphical output to be produced on a computer system. OpenGL implements its own specific version that gives clear steps as to what input is required and how it is processed in order to give a final scene. An overview of this pipeline can be seen in Figure 3.2.1, although it must be noted that two optional steps – Tessellation and the Geometry Shader – have been excluded as they do not pertain to this project.

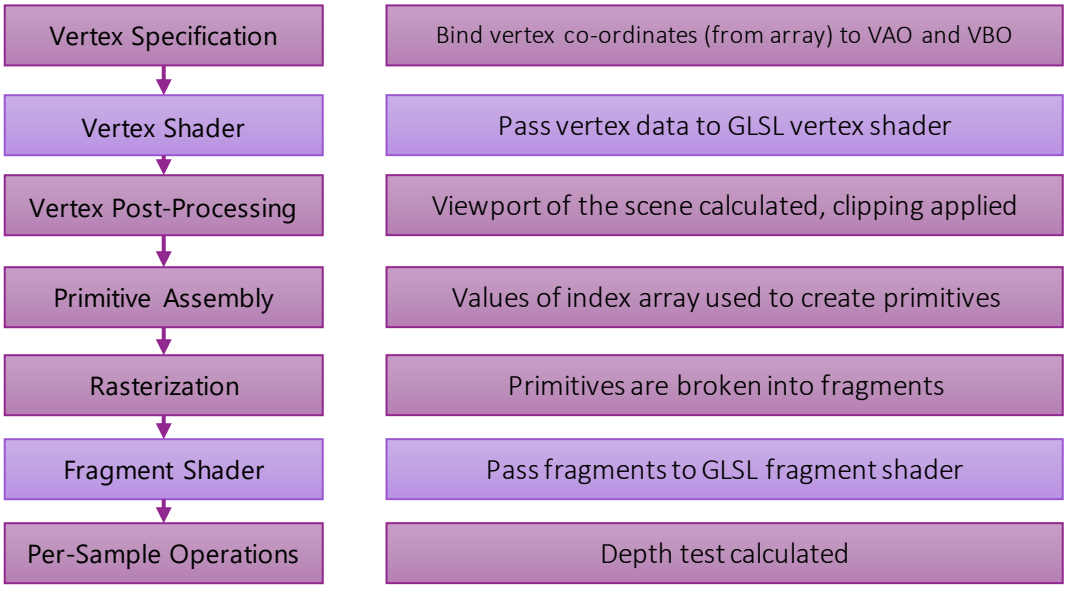


Figure 3.3.1 *– The primary steps of the OpenGL pipeline (omitting optional steps not implemented for this project) and a summary of how they are applied in the system.*

As mentioned previously, it is the direct, low level access to each step of the pipeline that gives programming in OpenGL more flexibility and control over the final rendering result. Each step must be explicitly accounted for in the system architecture – for example, in the version of OpenGL used, the vertex shader and fragment shader must both be written in GLSL, loaded, and attached to the current context to process incoming data.

The pipeline as implemented can be broadly divided into two stages – the vertex processing stage and fragment processing stage [10]. The former takes vertex information in array format from the VAO and VBO, handling each vertex independently. This data is passed to and processed by the vertex

shader, and will be output to be processed into primitives, a term meaning fundamentally simple shapes compiled from vertex and index data.

The latter stage processes these primitives into fragments in a process called rasterization. Their final colour and depth values are calculated, ready to be output. This is achieved within the fragment shader. Unlike vertex processing, this step of the pipeline is optional to implement – however, for the purpose of this project, it is vital to handle fragments in order to implement accurate and natural looking colour and lighting.

3.3.3 – Co-ordinate Systems and Projections

Throughout the pipeline process, several co-ordinate systems are used and converted between. These systems are categorised into 'spaces', which are labels that identify the current conversion step and give a more useful label as to what the raw values represent. One space is converted to another by means of matrix multiplication (with a 4x4 matrix) or transformation.

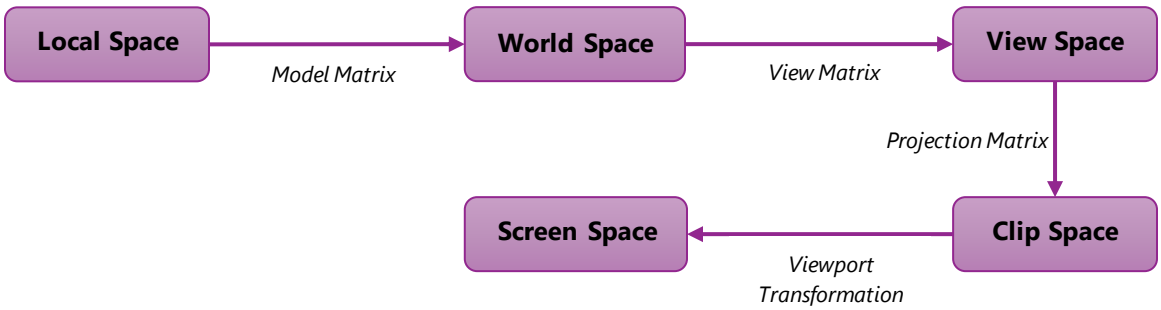


Figure 3.3.2 *– A visual representation of OpenGL spaces and how we convert from one to the next.*

Rendering begins in local space, concerning the original x, y, and z co-ordinates relative to a local origin, which are designated during the vertex specification step. A fourth w co-ordinate with a value 1.0 is also included at this step. This is required to facilitate multiplication between the co-ordinate vector and the model matrix, which converts co-ordinates into world space.

World space uses the model matrix to create a 3D model of objects as they will be positioned in the final scene, relative to the world origin as opposed to the local origin of each individual object. The model accounts for transformations applied to the object, namely any translation, rotation, or scaling.

The next conversion is to view space, which can be considered the eye's view of the scene – or alternatively, the camera's view. This is achieved with the view matrix, containing appropriate transformations to convert the perspective taken on the scene to that represented by the camera's front facing vector.

Clip space, converted to using the projection matrix, is an optimised version of view space that determines which vertices should be culled depending on whether they lie outside the boundaries of our view of the scene.

The projection provides an angle of view, as well a minimum and maximum draw distance – this combination, called a frustum, defines the space within which polygons will be drawn. The projection type used here is perspective based, meaning it imitates the perception of distance taken on real objects. Figure 3.3.3 provides an illustrated view of the frustum and its components.

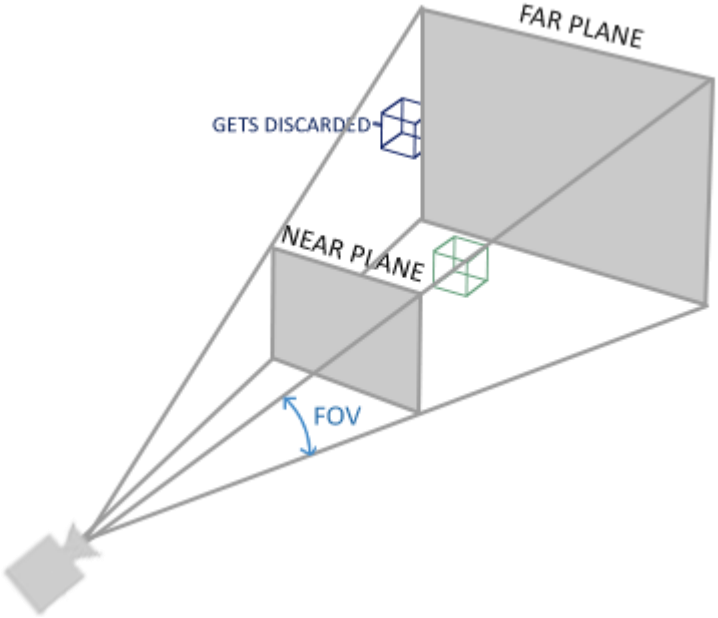


Figure 3.3.3 *– An illustration of clip space as designated by a frustum between the near and far plane. Any polygon with at least one vertex within the frustum will be rendered. Diagram taken from Learn OpenGL [11].*

In this project, these three preliminary conversions are made in the vertex shader. The built-in property `gl_Position` has its value assigned using the following multiplication:

$$gl\_Position = projection \times view \times model \times position$$

Where '`position`' is the local space co-ordinate of the current vertex. Due to the non-commutative nature of matrix multiplication, each transformation is applied in reverse of the order of steps.

The final transition to screen space is made within the code itself as opposed to the shader, with the viewport transformation being applied by the built-in function `glViewport(…).` The transformation specifies where the lower left corner of the user's view will be (set by default to the origin) and the dimensions of the window. Screen space forms the final view of a fully rendered scene, exactly as the user will see it displayed.

3.3.4 – External Libraries

Three key external libraries are used here to support OpenGL's core functionality – GLEW, GLFW and GLM. These three specifically were chosen for their balance of good documentation, modern feature support and regular updates. They also offer comparatively smaller added overhead to other modern libraries.

Each library provides a distinct set of features. GLFW facilitates the creation of windows (or more generally, OpenGL contexts) within which we can display rendered graphics and handles event polling for user input. Without this functionality, it may be possible to produce graphical output, but there would be no means with which to view it.

GLEW's function is to facilitate consistent cross-version and cross-platform compatibility for OpenGL. Specific features and extensions available for use on a given system vary widely according to the model and manufacturer of its GPU, in addition to other factors such as installed drivers. Accounting for this manually is laborious and in the scope of this project, unfeasible. GLEW ensures all common OpenGL functionality is available for use regardless of platform; this includes fundamentals like GLSL.

GLM provides a set of classes representing important mathematical structures for graphics production, such as vectors and matrices, as well as having built-in functions implementing complex vector mathematics. Its classes are used frequently throughout the project and allow more straightforward development of further logic.

4. Software Overview

Here, a high-level overview of the final software system is provided, highlighting the two key displays and the range of possible user inputs with some examples of their results.

4.1 – The PCG Menu

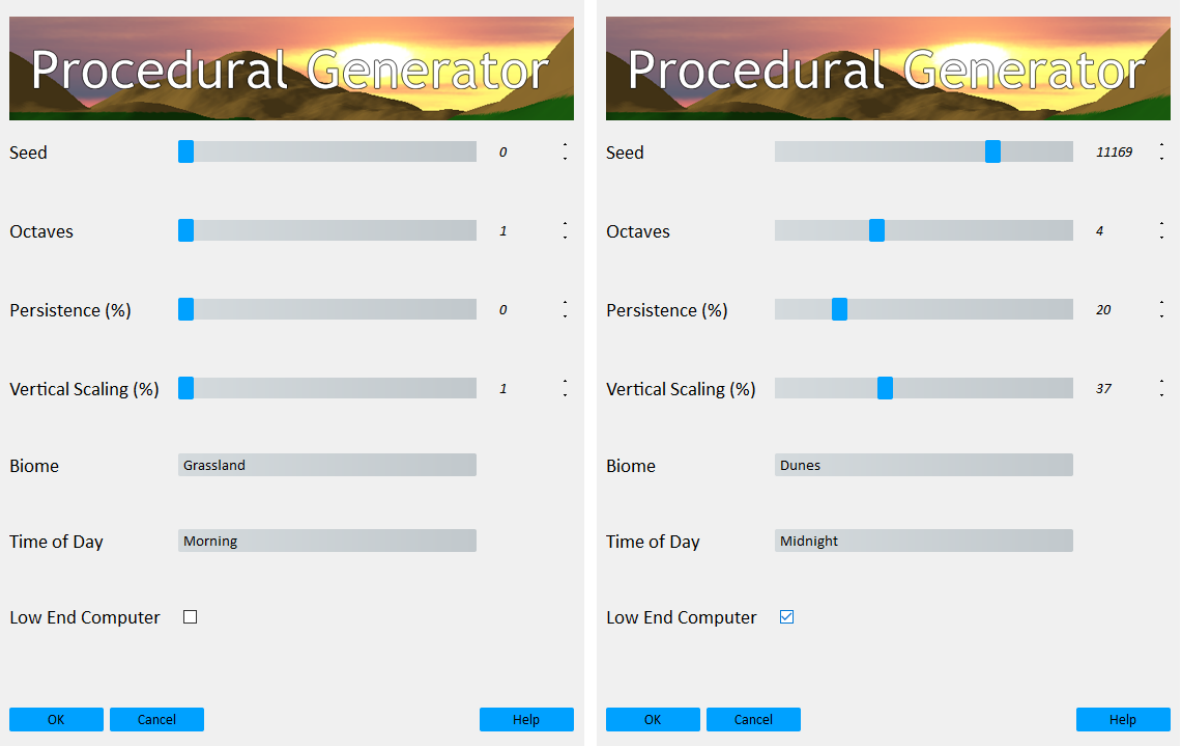


Figure 4.1.1 *– The main interface, comparing the initial view seen with default parameters and a form filled out by a user*

Upon execution of the main program, the user is presented with a clearly laid out form with default values. They have the option to customise each of the parameters before submitting it – most are set through sliders with minimum and maximum values, which may also be controlled from the corresponding spin box. The choice of environment is made through combo boxes, where possible options will appear in a drop-down menu.

It is also possible to specify that your PC hardware is low end, allowing compatibility options to be set within the program to accommodate lower rendering power.

If the form is submitted, the main rendering window will open. The user may also press the 'Help' button, which takes them to the following menu –

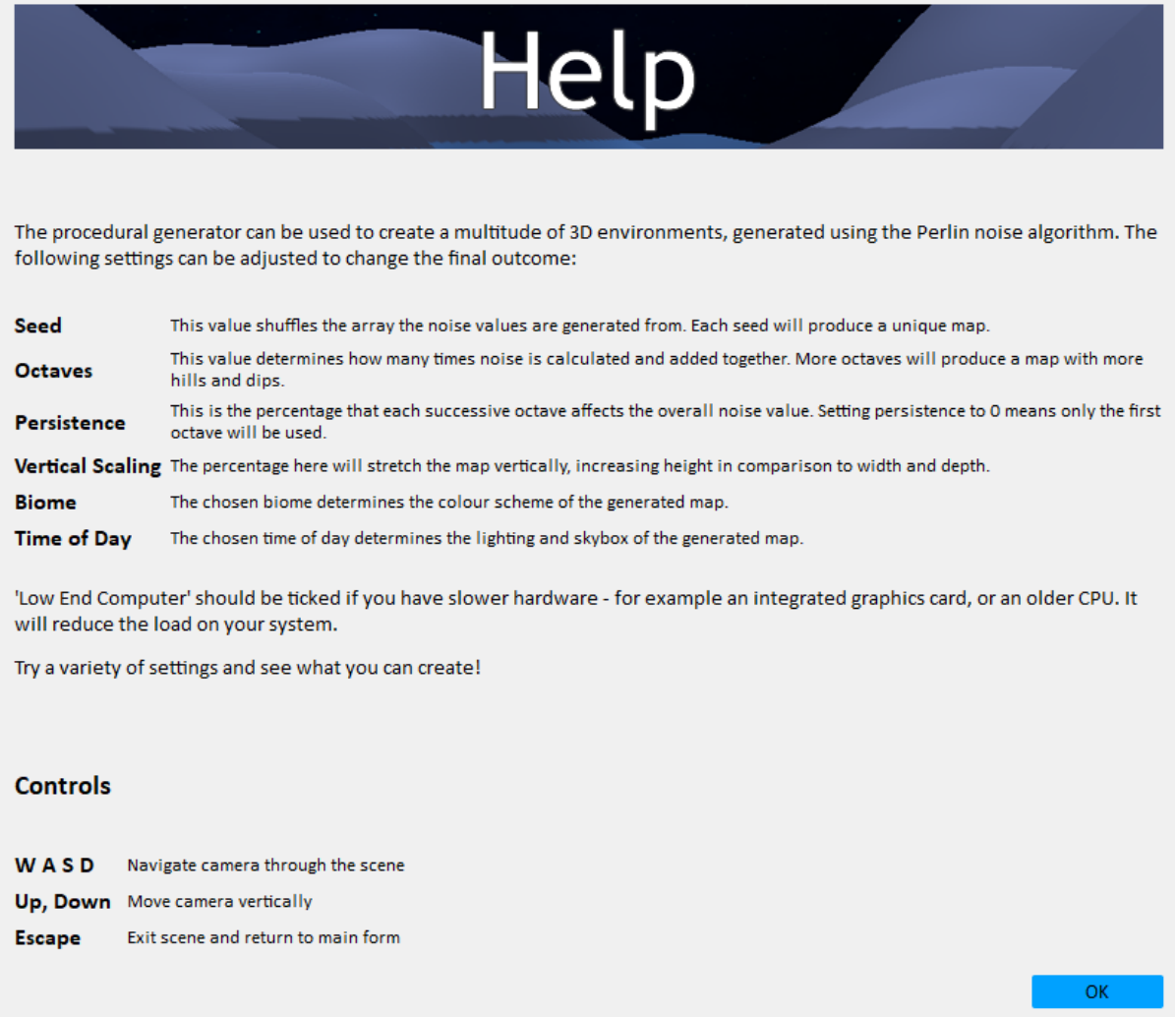


Figure 4.1.2 *– The help menu.*

This dialog is designed to give an overview of the different parameters, and the effect of changing their values. It also gives some general advice pertaining to the use of the 'low end computer' option, and how to navigate the scene produced. The dialog is modal, meaning input to the form is blocked while it is open – the user may return to the form by either pressing OK or manually closing the window.

4.2 – The OpenGL Rendering Window

The main rendering window displays a 3D OpenGL scene according to the parameters set in the form. The user has control over the settings for the noise generator, the skybox and lighting used, and the colour scheme applied to the final generated terrain.

There are potentially infinite permutations of such terrain with the settings available, a small sample of which are displayed in Figure 4.2.1. When the window is closed, the form opens again with the same

parameters as previously submitted already displayed for convenience. This allows the user to make small tweaks to values and compare the result.

Due to the nature of the engine created (see section 5.1), another programmer could use an instance of this rendering window to create their own custom scene as well, with or without the procedural generation aspect.

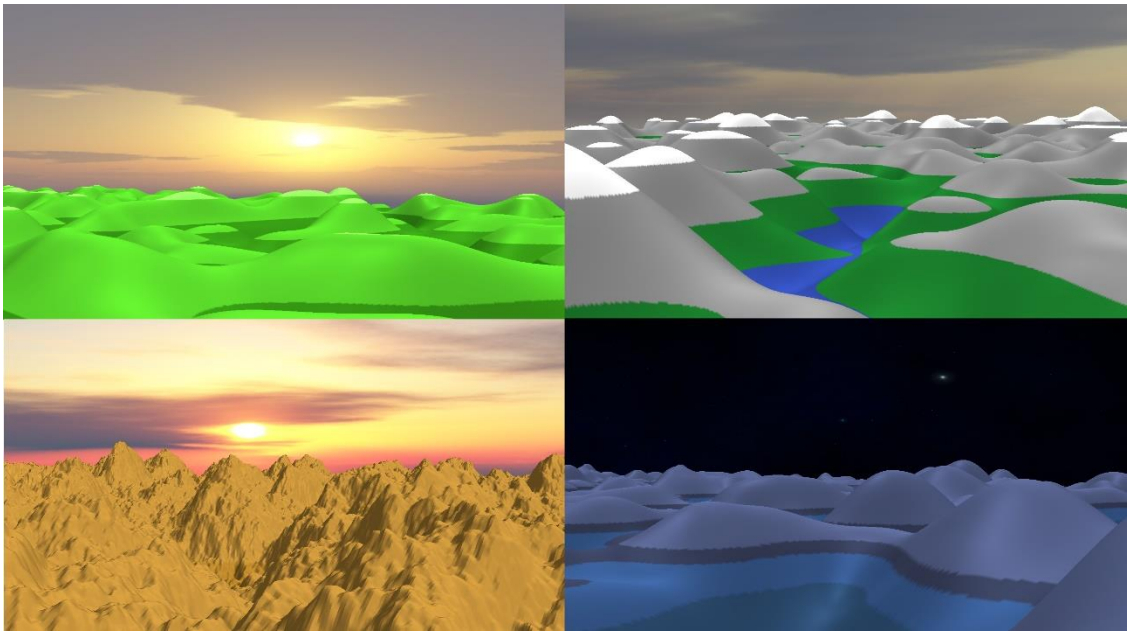


Figure 4.2.1 *– Some possible scenes, displaying all possible skyboxes and lighting, and all but one biome type.*

5. Software Implementation

A detailed description of the design and implementation of the software system is given, focussing on its key components – the OpenGL engine, the noise generator, the grid system and the GUI.

5.1 – The OpenGL Engine

The engine for this system provides a means of rendering fully lit meshes, both textured and untextured. A camera is used to facilitate navigation through the resulting scene. An additional class, Scene, has been implemented to provide an interface between a potential future user and the backend engine, allowing them to use it without needing to understand the theory and usage of OpenGL itself.

The classes have been grouped into a static library, `EngineLib.lib`, with a namespace `Engine` in order to improve reusability and portability.

5.1.1 – Window

The Window class serves as a container for the OpenGL context. An instance of Window can be used to display what is rendered by the other backend classes to the user; without it, calls to the OpenGL pipeline will not produce output. Window also handles the initialisation of the GLEW and GLFW libraries for convenience.

Window handles the logic of event listening, namely keeping track of the user's mouse and keyboard inputs and the length of time they are held down for. This information can be fed into the Camera class in order to determine camera movement and rotation.

Keyboard input capture is achieved using a size 1024 boolean array, with each entry corresponding with a GLFW enum entry representing a key. When a given key is pressed, the appropriate boolean flag is set to true, and is returned to false once depressed.

For mouse input, a flag within the GLFW window object itself is used to detect movement. Once movement is detected, the change in horizontal and vertical positions (`dX` and `dY`) are calculated from the difference between the current and last known positions.

The primary function for the class is `init()`, which contains the necessary logic to perform the initialisation process and create the window itself with the correct parameters. The process can be summarised by the following steps:

- Initialise GLFW and check for errors
- Set window properties and compatibility
- Create window using glfwCreateWindow() and check for errors
- Make the window the current context for OpenGL
- Assign key call-backs to handle event listening
- Disable the cursor and lock it to the window
- Initialise GLEW and check for errors
- Enable depth test and create viewport
- Assert that this instance of Window owns the GLFW window

Figure 5.1.1 *– The process of initialising and creating a window*

This function is vital for the use of the rest of the engine as it allows use of GLFW and GLEW functions. Hence, window creation is handled first during execution of any program using this engine.

5.1.2 – Camera

The Camera class allows the user to navigate the scene. A single instance should be created for any instance of Window. Altering the position and angle of the Camera object changes the displayed view; this is achieved by utilising the collected data from Window's event listening. The amount of movement depends on the time elapsed since the last movement trigger - this reliance on the change in time ensures that the speed of movement is independent of CPU or GPU speed, meaning camera movement is consistent across all systems.

The class consists primarily of functions that provide logic for updates to the three-dimensional position vector, direction vectors (front, up, right), its pitch (y-axis rotation) and yaw (z-axis rotation) of the camera. A default 'world' up vector also exists and represents which direction up is in context of the world space co-ordinate system. This is almost always `(0.0, 1.0, 0.0)`.

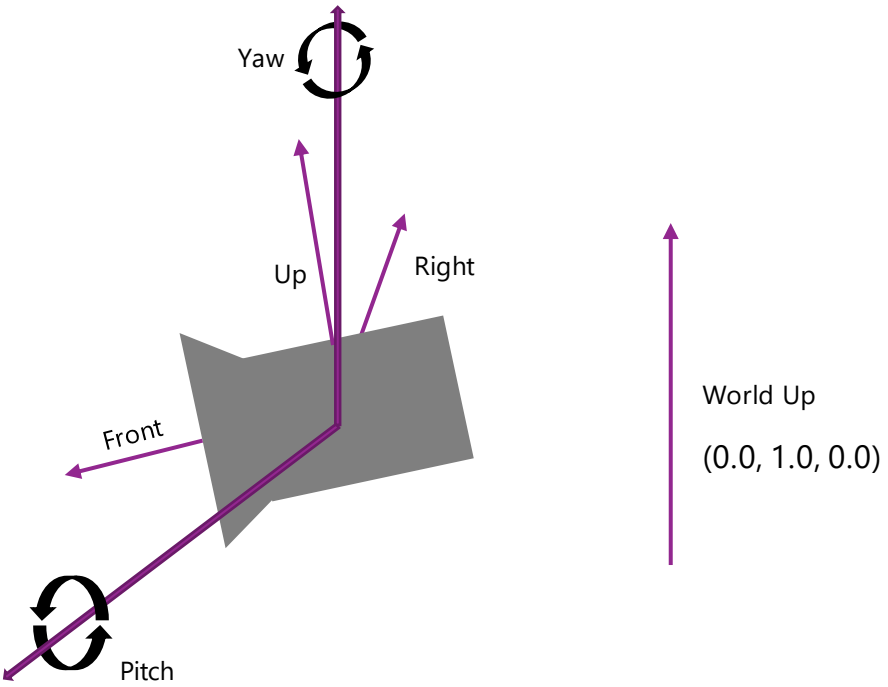


Figure 5.1.2 *– A visual representation of the camera's vectors (front, right and up) used to represent its direction, as well as its rotational axes, while it is tilted. World up is displayed to show how it differs from up.*

The front vector is a representation of the direction the camera is pointing, while the right vector points perpendicular to, and clockwise from, the front in the same plane. The up vector points at a 90 degree angle upwards from the front.

The function `keyControl(bool* keys, GLfloat dT)` handles updates to the camera position according to which elements of the `keys` array are set to true. A velocity is first calculated from the product of dT (the change in time) and the camera's movement speed property. The appropriate update is then applied to the position – for example, if the W key is held down, forward movement is made by adding the product of velocity and the front vector to the position. Conversely, if the S key is pressed, this product is subtracted from the position in order to move backwards.

`mouseControl(GLdouble dX, GLdouble dY)` handles mouse input, with `dX` and `dY` corresponding to the equivalent variables from Window. After being multiplied by the camera's turning speed, `dX` is added to the yaw, and `dY` is added to the pitch. A check must then be made to ensure that the angle made is below 90 degrees – failing to do so can lead to visual bugs and jagged camera movement.

Simply incrementing the pitch and yaw is insufficient – the updated values must be used to calculate new front, right and up vectors. The function `calculateDirection()` achieves this by first updating each value of the front vector. The x and z values depend on both pitch and yaw, while the y value only depends only on pitch.

The right vector is calculated from the cross product of world up and the new front vector. This is used in a further cross product with the front vector in order to determine the new up direction. All vectors are normalized to discard their lengths. With these calculations made, the 'tilt' of the camera, as well as the direction it points are correctly updated.

The boundaries of the camera's view (stored as a view matrix) help to determine what vertices are culled during the transformation from view space to clip space, as discussed in section 3.3.3. This can be retrieved from `getViewMatrix()`. A call is made to the GLM library `function glm::lookAt(vec3 position, vec3 eye, vec3 up)`. The `eye` vector represents the target of the camera (i.e. what it is looking at) and is calculated by simply adding the front vector to the current position.

5.1.3 – Shader

The Shader class facilitates the use of external GLSL shader files. It contains logic for reading a pair of shader files – representing a vertex and fragment shader - from a given file path, and then linking these to the current OpenGL instance and giving an ID to all required uniform values. A uniform value is any value marked as being present for every call to a given shader and will be applied in the same way to any pixel, vertex or fragment in the pipeline.

The function `createShaders(const char* vPath, const char* fPath)` serves as the public call to the backend logic, which is kept private. The two parameters are file paths provided by the user for the vertex and fragment shader respectively, which are then read from using the helper function `readSourceCode(…)`.

The retrieved data is passed to `compileShaders(…)`, which uses the following process to create a program and set up the shaders from their processed code (assuming that at each step verification is performed, and error information collected if necessary) –

- Call glCreateProgram(…) to set up a program with a unique ID for this instance.
- Call attachShader(…) for vertex and fragment code:
  - Call glCreateShader(…) to create an ID for this shader
  - Bind the code to this ID with glShaderSource(…)
  - Compile using glCompileShader(…)
  - Bind shader to program ID using glAttachShader(…)
- Link program to OpenGL context with glLinkProgram(…)
- Check that program can execute in current context with glValidateProgram(…)
- Locate each uniform variable within the shaders and retrieve its location with glGetUniformLocation(…)

Figure 5.1.3 *– The steps involved in taking raw shader code and turning it into an OpenGL program*

As each instance of Shader has its own unique program and ID, it is possible to create multiple sets of shaders for a given OpenGL context. Indeed, in most use cases, more than one set will be required. Hence, it is also important to be able to switch which shaders are currently in use. This is achieved with the function `useShader()`, which makes a call to `glUseProgram(…)` with the current instance's ID as its parameter. Switching between shaders is somewhat computationally expensive in OpenGL [12], but switches can still be performed multiple times in each rendering cycle without a hit to performance.

The GLSL Shader Files

For this particular system, three sets of vertex and fragment shaders have been implemented – a pair for coloured meshes, another for textured meshes, and a final pair for handling the skybox. They are stored in a subdirectory as plaintext code with the extension **".shader"** to mark them clearly as containing GLSL.

Although the specifics of each pair's implementation differ tailored to their requirements, each uses the following general logic –

- Input information is taken from the OpenGL pipeline into the vertex shader, according to the type of mesh it handles.
- The required output variables are calculated and set – this may include position, vertex colour, texture co-ordinates, normal values.

- The fragment shader takes these variables as input and calculates the final colour, which may consider lighting information.

For the colour and texture shader pairs, the fragment shaders contain the bulk of the logic as they calculate ambient, diffuse and specular lighting for each incoming fragment in order to determine the final colour of it. Conversely, the skybox fragment shader simply uses one line of code to apply the appropriate texture. GLSL is fairly flexible in this regard, and a user could easily implement their own custom shader with this engine as long as it follows the rules set out by the functions of the Shader class.

5.1.4 – Mesh

Two types of mesh are supported in this system – coloured and textured. The two types require slightly different logic, as rendering with texture as opposed to pure colour requires a different set of shaders. Both types can be created from the Mesh class; the user must specify a value of `COLOURED` or `TEXTURED` from the enum `MeshType` when creating a Mesh object in order to indicate which is required.

In order to create the actual mesh in OpenGL, a call to `createMesh(…)` must be made, to which we pass an array of vertex values and index values. While the format of indices is the same for both types of mesh – groups of 3, denoting the order vertices will be drawn in and joined to form a triangle primitive – the format of vertices differs with the type of mesh being created.

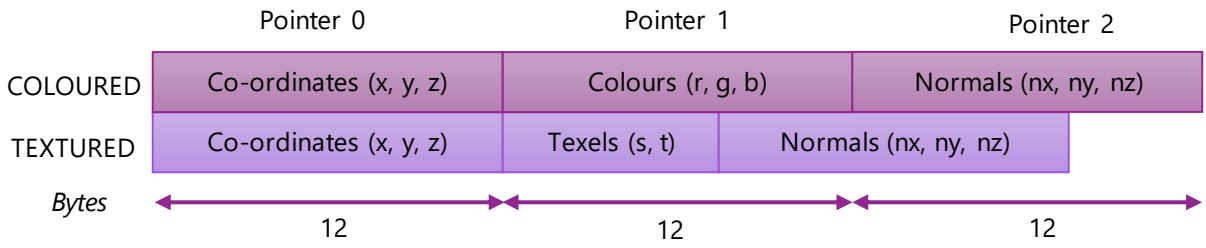
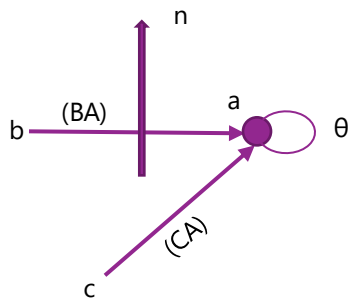


Figure 5.1.4 *– The attribute pointers of the two mesh types, i.e. the data provided in the vertex array and the order it must be in. A coloured mesh takes 36 bytes per vertex, while a textured mesh takes 32.*

First, the normal values must be calculated. In this engine, both for user convenience and to allow easier automation of mesh production, the three normal values of each vertex in the incoming vertex array are initially 0, and are updated by a call to `setNormals(…)`.

As each normal is a vector perpendicular to one of the triangles making up the mesh, vector mathematics is required in order to correctly set the nx, ny and nz values. The index array is looped through, considering three indices at a time. As these indices refer to three different vertices, we can multiply their values by the stride (the number of values per vertex) in order to retrieve the location of the first value of the corresponding vertices. These are passed to `calculateNormal(…)`, where the following steps are applied –



$$\vec{n} = \overrightarrow{BA} \times \overrightarrow{CA}$$

- From the three values, find and store the x, y and z co-ordinates of the vertices (here labelled a, b and c)
- Calculate the vectors BA and CA
- Calculate the cross product of the two vectors
- Normalize this vector and return it
- Set nx, ny and nz for the three vertices

Figure 5.1.5 – *Calculating the normal to a surface made by three vertices a, b and c. A diagram is provided for clarity.*

Once each normal has been calculated, a second pass is made through the vertex array with the objective of normalising once more. This must be done because some vertices will have had their normal calculated multiple times due to the nature of indices – an index that is repeated in order to join two drawn triangles will lead to a second normal calculation for the vertex pointed to. If the values are not normalised again after the fact, noticeable rendering bugs may be seen, and lighting will not apply correctly.

With all normals set and corrected, the next steps of setting up the mesh are completed. A VAO is created and bound, which is used to indicate to the pipeline what type of data a vertex contains, namely its format and size. An IBO and VBO can then be bound to this object, which store the vertex and index data respectively.

Properties referred to as attribute pointers are set in the final step. As shown in Figure 5.1.4, these indicate what information is stored in which location in a vertex. For example, the 0^{th} attribute pointer of both mesh types refers to the co-ordinate data, which takes up 12 bytes of space. This is vital for the shaders, which refer to the different components of the vertex data by their location.

The final mesh can be rendered using the `drawMesh()` function. This sets the VAO and IBO to be active in order to retrieve the relevant data, and then calls `glDrawElements(…)` with the mode `GL_TRIANGLES` active. The vertices are drawn and joined together according to the order of indices.

5.1.5 – Texture

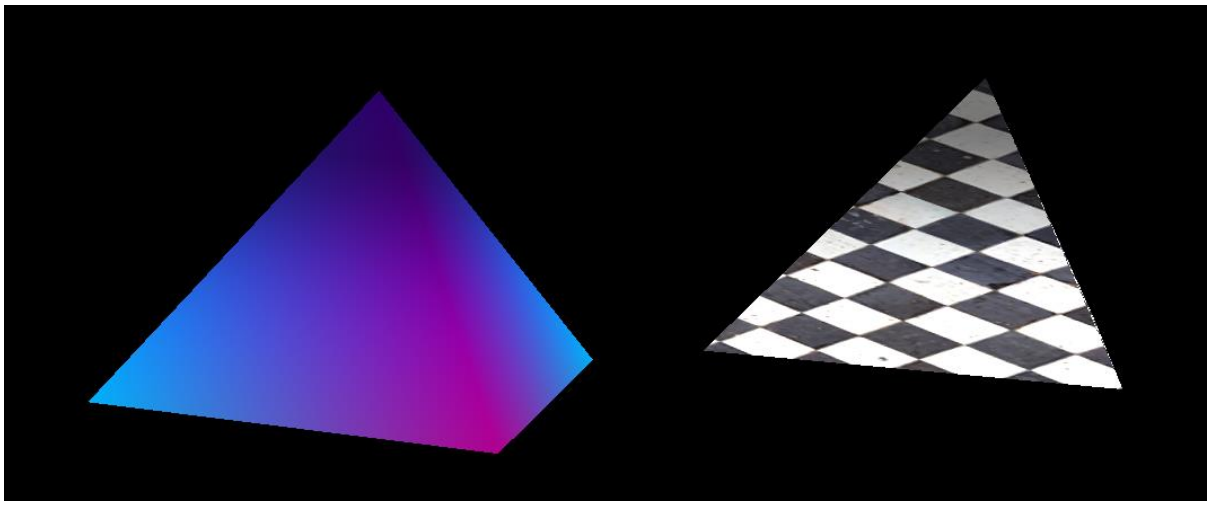


***Figure 5.1.6** – A comparison of two meshes, without and with texture. They are otherwise identical.*

Texture implements the logic of loading images and converting them into texture data, which can be loaded during the rendering cycle and applied to a mesh of the appropriate type. A header library, `stb_image.h` [17], is utilised to load the images, and the data retrieved from its functions is then handled appropriately.

Both JPG and PNG file extensions are supported, and an enum value specifying the type required is passed along with the file path to `generateTexture(…)`. Texture data is taken from the function `stbi_load(…)`, which also sets the height, width and bit depth to be used during texture creation. Upon successful load, the data is passed to the private backend function `bindTexture(…)`.

A call to `glGenTextures(…)` is made to set up an OpenGL texture with a unique ID. We can then call `glBindTexture(GL_TEXTURE_2D, id)` to denote that the texture with this ID will be a 2D image. Texture properties are set – the image is repeated in both x and y directions, and the texture type is set to `GL_LINEAR`, meaning smoothing is applied if image quality is low or pixelated.

The appropriate colour channels are then selected according to the file extension. PNG requires both channels to be RGBA, meaning it uses an additional alpha channel on top of its colour information (usually used to store transparency data). JPG requires one channel to RGBA, the other purely RGB.

With all settings in place, a call to `glTexImage2D(…)` is made with the width, height, channel types and texture data. This finalises texture creation. It can later be rebound with `useTexture()`, which will apply this texture to any mesh drawn after the call.

5.1.6 – Material

Material stores information about the shininess of an object and the intensity of specular lighting that reflects from it. These two settings could be stored with the Mesh and Light classes respectively but abstracting them into a separate class allows a greater degree of control – for example, two otherwise identical meshes could have different levels of shine, without wasting time or memory creating a second mesh.

Figure 5.1.7 *– A comparison of a shiny material (high specular intensity, high shininess) and a dull material (low specular intensity, low shininess).*

The logic of Material is simple, as it serves primarily as storage for its two variables. It has a function `useMaterial(…)`, which attaches this material as a uniform to the current shader. This applies it to subsequent meshes, until another material is used.

5.1.7 – Light

Light serves as a base class, storing information about the colour of light in a scene and its ambient intensity, and providing a virtual function `useLight(…)` that can be overridden as appropriate by other types of light. This function links the light properties to the current shader for use.

As ambient lighting is constant throughout the entire scene, it has no direction associated with it. However, diffuse and specular lighting both require a direction, which is handled by the derived class DirectionalLight. Naturally, this class also holds the intensity of diffuse light.

Lighting Calculations

The bulk of logic for lighting is implemented in the colour and texture fragment shaders; the Light class itself serves purely as a means of providing information to this shader. The Phong lighting model has been implemented as it strikes a balance between realism and complexity. The final fragment colour is calculated according to the equation $Fragment\ colour = vertex\ colour \times (ambient + diffuse + specular)$.

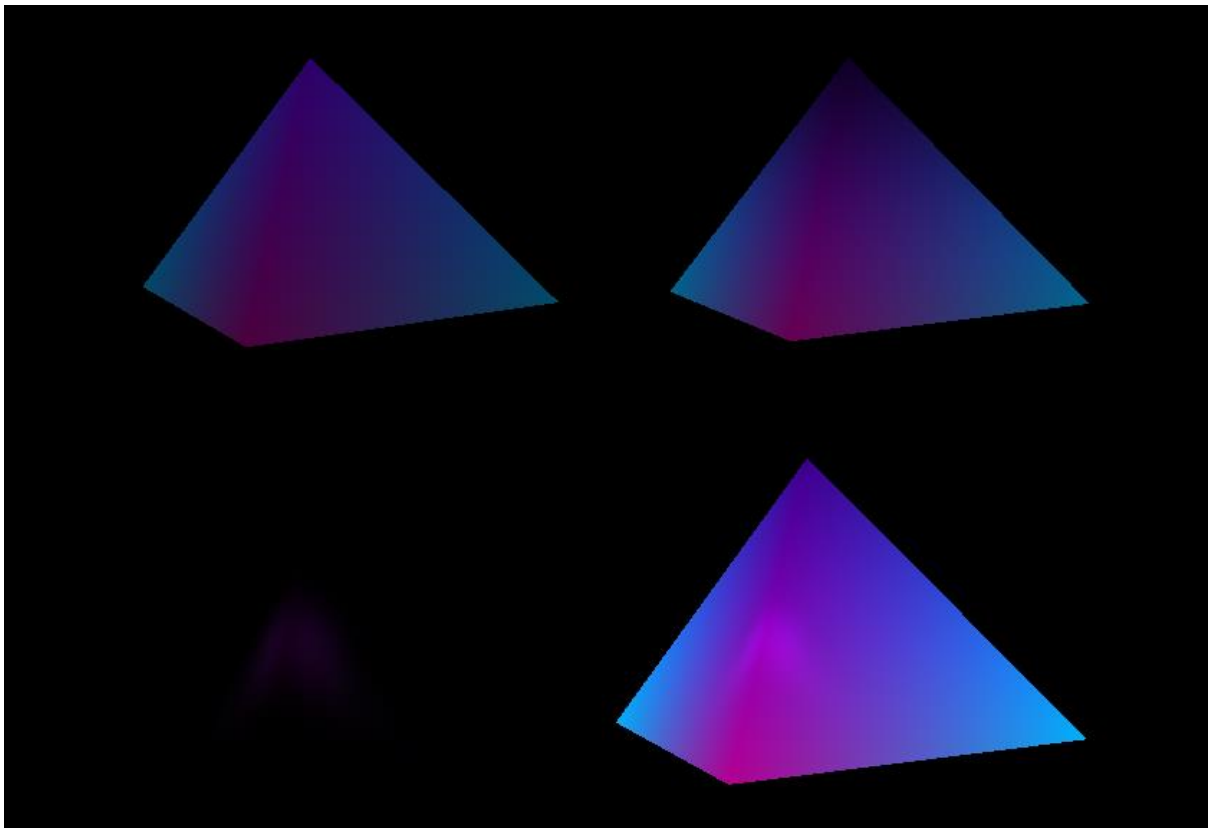


Figure 5.1.8 *– A simple example of the different lighting types: ambient, diffuse, specular, and then all three combined to produce the Phong lighting model.*

Ambient light can be calculated simply from the product of the light's colour and the ambient intensity, as no other factors need to be considered.

Diffuse lighting, however, requires an extra step to determine whether any light from the DirectionalLight is hitting the current fragment. According to Lambert's law, the strength of lighting that hits a diffuse surface is proportional to the cosine of the angle between the surface normal and light direction [13, p54]. Hence, a dot product can be calculated between the fragment's normal and the direction of light to find this strength.

Due to the nature of the cosine function, the closer the angle is to 90°, the closer the dot product will be to 1.0, meaning more diffuse light is hitting the surface. If a negative value is produced, 0 is substituted to avoid negative lighting.

The final diffuse amount is the product of the light colour, diffuse intensity, and the multiplier obtained from the dot product. Hence, if that value is 0, no diffuse lighting is applied – the fragment will only have ambient light.

Specular lighting is the most complex to calculate and depends on several factors – light must hit the surface, and the camera must be in a position to see the reflected light. If no diffuse lighting hits a fragment, no specular lighting will either, so the value is immediately set to 0. Otherwise, we calculate two vectors - the reflection of light from the surface, and the direction from the fragment to the camera. Light is reflected according to the direction of the surface normal.

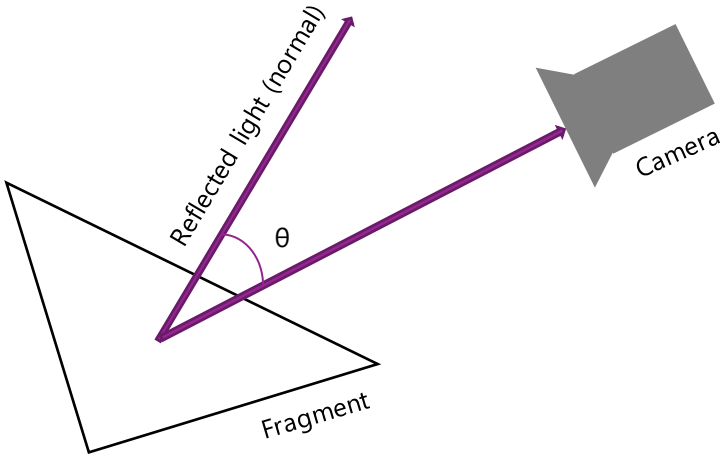


Figure 5.1.9 *– A visualisation of the angle between the normal and fragment-to-camera vectors. This assumes that the light ray is hitting the surface directly, so that the direction of the reflected light is identical to the normal.*

To find the angle between these two vectors, we calculate another dot product. This gives us the strength of visible specular lighting – if the camera is in a position where it couldn't possibly see the reflection, the value will be 0. Shininess is then applied – this value is raised to the power of the material's shininess. The final specular amount is the product of the light colour, specular intensity and the adjusted dot product result.

5.1.8 – Skybox

A skybox can be considered a backdrop to a 3D environment – it is a set of textures drawn at an infinite distance, meaning no matter the user's position or perspective, the skybox appears to be the same distance away.

In this engine, the skybox is implemented as a cubemap. This is a set of 6 textures loaded onto the faces of a cube. In the scene itself, the edges of the cube between the textures cannot be seen, as the textures are seamless, and no lighting is applied.

In a similar manner to Texture, logic is implemented for binding an OpenGL texture and generating a unique ID. The type here is `GL_TEXTURE_CUBE_MAP`, as opposed to the 2D texture type used previously. The texture parameters set ensure that the image clamps to each edge, so that no gaps can be seen.

The cube used to display the textures is a simple mesh. A VAO and VBO are generated, but an IBO is not required, as a single unit cube is used – this does not require indices to be drawn. Only one attribute pointer is used, storing the x, y, and z co-ordinates, as no texture co-ordinates or normals are needed.

The file paths of the desired textures are passed to the function `loadSkybox(…),` stored in a vector for convenience. For correct tiling, they must be in the order right, left, top, bottom, front, back. For each file path in the vector, `stbi_load(…)` from `stb_image.h` is used to load and store the texture data. This is used by `createFace(…)` to create the corresponding face of the cubemap. Each is its own 2D texture and is created as such.

Using `drawSkybox()` rebinds both the VAO and the cubemap, to ensure the textures are overlaid onto the mesh before it is drawn. When called from the main rendering loop, depth testing is switched to `GL_LEQUAL` mode, to ensure all fragments of it are drawn regardless of where they end up in screen space.

The implementation of Skybox was produced with help from LearnOpenGL [14].

The Skybox Shader

The illusion of infinite distance is achieved through the view matrix and a specialised shader. The camera's view matrix is retrieved and has its translation properties removed by converting it to a 3D matrix. This depthless view matrix is set as the uniform view for the vertex shader.

The positions of the skybox vertices are calculated from the product of the projection matrix, view, and co-ordinates. No model transformation is applied, because the skybox doesn't need to be positioned relative to the other meshes in the scene. The depth component can then be removed entirely from the position using 'swizzling' – the z column of the position matrix is replaced with a copy of the final w column.

The fragment shader can then simply apply the appropriate texture from the cubemap to the corresponding cube face in order to determine the final fragment colour.

5.1.9 – FPSCounter

FPSCounter is a small utility class implemented for performance monitoring. It links to a Window object and updates its title with an FPS value periodically, tracking how many rendering loops are made in a second. The higher the value, the better performance generally is.

The `update()` function is called once per rendering loop. It checks the current time by using `glfwGetTime()` and from this determines how much time has passed since the last FPS calculation. If

this is less than the minimum interval set by the user, the frame counter is incremented. Otherwise, if at least the minimum interval has passed, a new FPS value is calculated from $\frac{change\ in\ time}{frame\ count}$. The frame counter can then be reset, and the reference time updated.

This function was adapted from [15] and improved upon, being made into its own class for the benefits of encapsulation and reusability.

5.1.10 – ColourMap

`ColourMap` is a struct used to store light colour, direction, and a set of RGB colour values separated by height boundaries. The colours are separated into `peak`, `high`, `mid` and `low` categories. The `bounds` vector provides three values representing the heights at which a transition is made to the next lowest colour. For example, the first boundary value marks the transition from `peak` to `high`.

The struct is used in the PerlinGrid class (see section 5.2.2) in order to striate colours in the terrain by their calculated height. It creates a colour scheme that may be used to represent a certain biome or environment; the inclusion of lighting information helps to reinforce this.

It is kept standalone to give the user the option to create their own custom mappings, which may then either be set directly for PerlinGrid, or used in tandem with the Scene class.

5.1.11 – Scene

Scene combines all previously described engine classes, with the aim being to provide a single point of access a user may interact with to produce their own 3D scene. It removes the requirement entirely of having to know how to interact with the OpenGL pipeline, with the added benefit of providing a reusable render function that no longer relies on being hardcoded in `main`.

The logic of Scene can be considered in several distinct parts – adding meshes and instances of PerlinGrid, setting variable values and render options, and handling lower level OpenGL logic with the aim of producing the final result. It also makes use of three helper enums – MaterialType, BiomeType, and TimeType – to further simplify use for the end user and group together related settings.

Adding Meshes

Adding coloured and textured meshes is achieved through using the overloaded functions `addColourMesh(…)` and `addTextureMesh(…)`. All overloads take a pointer to a Mesh object as a parameter, as well as a `MaterialType` value indicating whether specular lighting for the mesh will be `SHINY` or DULL.

Additional possible parameters for both types are the desired position, scale and rotation of the mesh when it is rendered. For textured meshes, there exist options to either directly provide a pointer to a Texture object, or to provide a file path to the desired image.

Wherever appropriate, validation is put in place for these functions. Null pointers are checked for first; passing a null Mesh, for example, results in immediate exit from the function with an error message. The type of the Mesh passed is also examined to ensure no type mismatches exist when the mesh data is passed to the shaders. Naturally, for a textured mesh added with a file path, the validity of the path is also examined before a Texture object is created and stored.

Should all validation succeed, an instance of the struct `MeshInfo` is created. This groups together relevant data pertaining to the added mesh, namely its transformations, `MaterialType`, and corresponding Texture if it exists. This is added to a storage vector, allowing all required information to be accessed at once during rendering.

Adding a PerlinGrid

Each instance of Scene may hold a pointer to exactly one instance of PerlinGrid. It may either be passed and set directly with `setGrid(…)`, or be created according to the parameters passed to `addGrid(…)`.

A flag `gridExists` keeps track of whether a grid is currently in use. At render time, if a grid does exist, the speed of the camera is adjusted according to its scale in order to ensure the full scene can be navigated at a reasonable pace. If a new grid is added while one already exists, the old instance is removed using `deleteGrid()`.

Setting Variables

Control is given over key variables such as camera positioning and angle, light properties, and window title through simple set methods. Particularly of note are `setBiome(…)` and `setTime(…)`, which affect the look and feel of the resulting scene.

Biome choice affects the `ColourMap` used for the PerlinGrid. Five types exist, each with a corresponding colour scheme appropriate for the name, with boundaries customised to produce a suitable colour distribution. Time affects the result as a whole, as it determines the skybox and light settings.

Certain combinations of biome and time settings result in overly bright reflections, while others cause underlit environments. As a result, combinations are tested for compatibility before rendering using the `checkCompatibility()` function, which accounts for such pairs by adjusting light intensity.

Rendering

A public call to `render()` begins the rendering process. The process can be summarised as follows –

- Check that rendering window exists
- Calculate projection matrix
- While the window is not set to close:
    - Calculate the current time, and change in time
    - Update the FPSCounter
    - Poll for user input and assign key/mouse controls
    - Update the view matrix
    - Set up viewport and clear buffers
    - Draw the PerlinGrid object, if it exists
    - Draw any stored meshes
    - Draw the skybox
    - Swap buffers to display all drawn objects
- Clear the shader in use
- Hide the window and reset its should close flag

Figure 5.1.10 *– The procedure of the main render function.*

Each drawing step makes a call to a corresponding function. All specialised render functions set up and use the appropriate shader, link the required uniform values to this shader, and update transformation matrices if necessary.

`renderGrid()` adjusts the scale element of the model according to the grid's horizontal and vertical scale factor, and calls its own `render()` function. `renderColouredMeshes()` and `renderTexturedMeshes()` examine each element of the meshes vector, determining which meshes to render from the presence (or lack thereof) of a corresponding Texture. `renderSkybox()` follows the process described in section 5.1.8, calculating its depthless view matrix before calling `drawSkybox()`.

5.2 – Noise and Procedural Generation

As previously stated, Perlin noise is used as the coherent noise algorithm in this system. A class has been implemented to calculate this noise with various parameters, which is then combined with elements of the OpenGL engine in the class PerlinGrid in order to produce the procedurally generated terrain.

5.2.1 – Perlin

This class adapts Ken Perlin's improved noise algorithm [5] from its original Java syntax to C++, adding additional features and control over output not present in Perlin's work – namely the ability to 'seed' the noise and layer several noise calculations.

The permutations array is implemented here as a vector. This provides more flexibility with regards to resizing and changing the order of values in C++ with minimal additional overhead. The default permutations vector is as per Perlin's original order of values, and this is duplicated to reach the required 512 values.

If a seed value is provided through the constructor or `setSeed(…)`, a call is made to `shufflePermutations()`. This function makes use of the built-in `std::default_random_engine` class, using the seed to set up its RNG. The shuffle operation can then be applied, which will arrange the values 0 to 255 in a pseudorandom order. In this way, the same seed will always produce the same permutations.

Noise generation is implemented for both 2D and 3D input in the functions `noise2D(x, y)` and `noise3D(x, y, z)`. These follow the algorithm as laid out in section 3.2.1. Functions for fading, calculating the pseudo-random gradient, and linearly interpolating are all written as specified by the algorithm.

First, the co-ordinates of the unit square (or cube in 3D) are calculated by taking the floor of the original co-ordinates and 255, where 255 is introduced as a factor to avoid leaving the bounds of the permutations vector. The co-ordinates also have the `fade(…)` function applied.

Hashing the vertices of the unit square is achieved through accessing the corresponding permutation, for example $permutations[X]$. These hashes are used to calculate the pseudorandom gradients required.

The `gradient(…)` function has been made clearer in comparison to Perlin's version, separating each step of the hash comparisons in order to make the source of the final value more apparent. The full process is as follows –

Input: hash, x, y, z

- Calculate (hash & 15) to isolate the first 4 binary digits of the hash
- Consider most significant bit – if it is 0, then a = x. Else a = y.
- Consider 2nd most significant bit:
    - If 1st and 2nd bits are *both 0*, then b = y
    - If *both are 1*, then b = x
    - If the bits are *different*, then b = z
- Consider 2nd to last bit: if it is 1, then make b negative.
- Consider final bit: if it is 1, then make a negative.

Output: (a + b)

Figure 5.2.1 *– The comparisons made to calculate the dot product described in Perlin's algorithm (here represented by a and b).*

The final step linearly interpolates been the faded values and the results of the gradient calculations using `lint(…)`. This is again split into smaller steps for clarity, which can especially be seen in the 3D function where many calls to `lint(…)` and `gradient(…)` must be made to produce the final result.

Octave Noise

In addition to the base noise algorithm, functions for producing octave noise in 2D and 3D are implemented. These make use of two further parameters representing the octave count and persistence. Noise is calculated from the original functions a number of times equal to the octave count, and then summed.

The proportion of the value from each octave added to the total scales with persistence – the higher the persistence value, the higher the proportion of the value added. Persistence is always between 0 and 1, meaning a given octave may never have a greater influence over the result than previous ones. This effect is represented in the function by a variable 'amplitude', which is multiplied by persistence after each octave is accounted for.

Each octave's noise result will differ, as the co-ordinate values are multiplied by a 'frequency' value before being passed to the noise function. This is equal to 2^i, where i is the current octave. Hence, we can see that if only one octave is used, the final result will be identical to a call to the original noise function.

The exact steps of calculating octave noise are demonstrated in the following code fragment –

```
for (uint i = 0; i < octaves; ++i) {
	total += noise2D(frequency*x, frequency*y) * amplitude;
	max += amplitude;

	frequency *= 2;
	amplitude *= persistence;
}

return total / max; // Normalise value so it remains in range [-1,1]
```

Figure 5.2.2 *– A code fragment demonstrating how 2D octave noise is produced from the given octave count and persistence value. This function was produced with help from [7].*

Octave noise was implemented to greatly increase the range of the procedural generator – combining the ability to change permutations through a seed value with these additional functions gives potentially infinite final results. The disadvantage, however, is that increasing the octave count greatly increases time spent calculating noise values; for this reason, a hard limit of 10 octaves has been set for the class, in order to avoid unacceptable effects on performance.

5.2.2 – PerlinGrid

This class uses the results of noise calculations from Perlin with the features of the implemented engine in order to create and render a set of meshes in the appearance of terrain. The final appearance of a PerlinGrid depends on the parameters provided for both its intrinsic sizes and its internal Perlin instance. In this way, it serves as the backbone of procedural

The grid is formed of `size` rows and columns. A maximum `size` of 4000 (`MAX_SIZE`) is enforced to limit the final vertex count and prevent memory overflow. Each row is a separate Mesh object, and its individual cells have width and height `cellSize`. Together, the meshes for each row combine to give a seamless effect when rendered.

Use of any constructor will result in a call to `regenerate()`. This function prepares the data required to complete the grid and its constituent meshes.

Producing the Vertex Arrays

As each row is its own mesh, a vertex array must be produced for each one. Production of each array is achieved with `generateVertexArrays()`. The total length of a given array is calculated from $vCount = ((size \times 2) + 2) \times 9$. This is twice the number of columns (represented by `size`), plus an additional two for the first two vertices of the row, multiplied by the number of values per vertex (taken from the number of values per colour vertex).

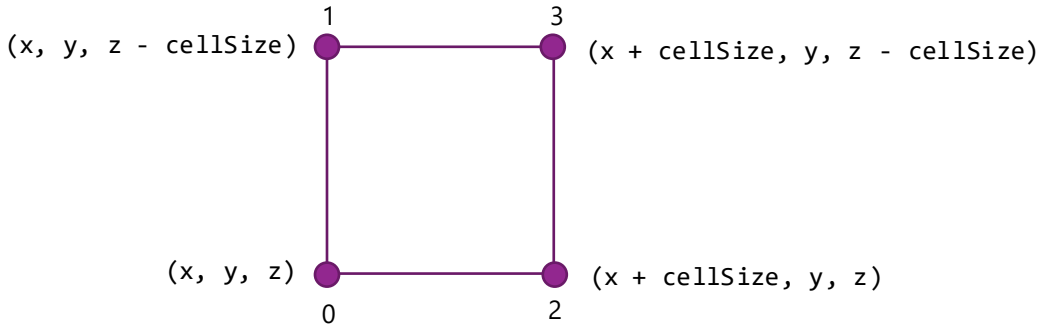


Figure 5.2.3 – *A representation of a single cell, where vertices are labelled with their corresponding index and their raw co-ordinates. Vertices 0 and 2 are low, while 1 and 3 are high.*

Three values representing `x`, `y` and `z` co-ordinates are initialised to 0. A boolean `low`, representing whether the current vertex is low or high (see figure 5.2.3), is initialised to true. With these values prepared, a nested for-loop handles setting the values for each array – the outer loop ranges from 0 to `size`, while the inner loop ranges from 0 to `vCount`, handling individual arrays with a stride of 9.

The assigned co-ordinate values for each vertex depends on whether it is low or high. Both have their `y` value calculated by the Perlin noise function `octaveNoise2D(...)`. Low vertices take the current `x` and `z` values as is, whereas high vertices take `x` and (`z` – `cellSize`), as per Figure 5.2.3. The subtraction of `cellSize` moves the vertex further 'into' the screen, thus separating it from the lower vertices while still being aligned on the x-axis.

Whenever a high vertex has its values calculated, we then increment `x` by `cellSize`. This moves the vertices ahead one column, and so the process repeats to produce the entire row.

`r`, `g`, and `b` values are set using `calculateRGB(...)`. This function examines the current `y` value and compares it to the boundaries contained within the current `ColourMap`. The appropriate colour is returned according to the height represented by `y`. As would be done for any other Mesh, the three normal values for the vertex are initialised to 0, to be calculated while the Mesh is being created.

Once a vertex array is filled, the value of `x` is reset to 0, bringing it back to the first column. `z` is decremented permanently by `cellSize` to move along to the next row.

Producing the Index Array

As every row is of identical length, with the same number of vertices, only one index array needs to be produced. The values can be reused for each row mesh. Each cell is formed of two triangle primitives, designated by three indices, which gives a total index count of $size \times 6$.

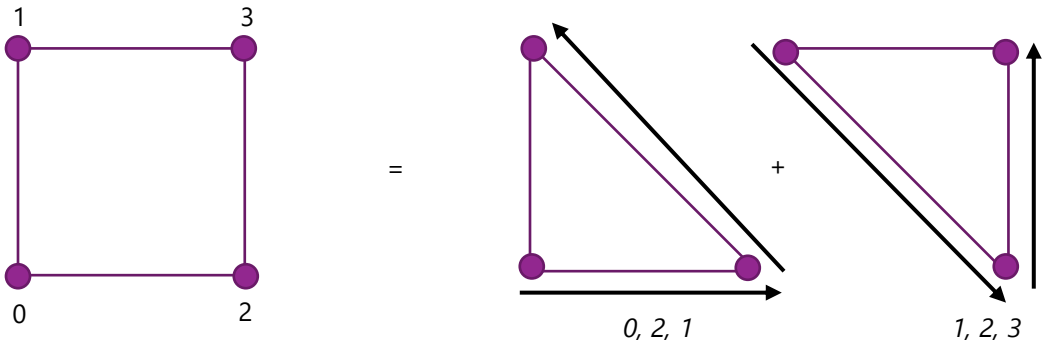


Figure 5.2.4 *– A cell split into its two constituent primitives, with the order of indices labelled in both number and arrow form. Here the 'pivot' vertex is 0.*

Within a for-loop, a full set of indices for a cell is produced each iteration. The process is generalised by selecting a 'pivot' vertex, from which the other values are relative. The bottom left vertex was selected for simplicity, as this is always the lowest vertex number in a given cell. The following code fragment demonstrates how the pivot value is applied:

```
for (uint i = 0; i < iCount; i += 6) {
	// Produces the index corresponding to the bottom left vertex
	int pivot = (2 * i) / 6;

	indexArray[i] = pivot; // First triangle
	indexArray[i+1] = pivot + 2;
	indexArray[i+2] = pivot + 1;

	indexArray[i+3] = pivot + 1; // Second triangle
	indexArray[i+4] = pivot + 2;
	indexArray[i+5] = pivot + 3;
}
```

Figure 5.2.5 *– Code fragment demonstrating how index values are determined with the pivot.*

Generating and Drawing Meshes

Mesh creation is performed at the end of each vertex array production loop. A new Mesh representing a row is created and stored in a vector. It is supplied with the current vertex array and universal index array through `createRow(…)`. The vertex array can be discarded safely after this, which prevents excess memory being dedicated during grid creation; a full set of vertex arrays for a grid of `size` 3000 can take upwards of 1.5gb of memory.

Rendering the grid is similarly simple – the function `render()` calls `drawMesh()` for each row Mesh.

Variable Control and Regeneration

In addition to set functions for sizes, scales and the chosen `ColourMap`, the class provides options for updating the noise result. This is achieved through either directly setting the Perlin related variables (seed, octaves, persistence), or providing an entirely new Perlin object.

Any variable update that would change the final grid's appearance results in a new call to `regenerate()`, as data must be recalculated in order to reflect the new values.

5.3 – The User Interface

The GUI for this system is built with the Qt API. It consists of a main form used to take user input, an additional help dialog to provide instructions and advice, and a small utility dialog used as a loading screen. Qt logic relies upon two key concepts – signals and slots. Signals are 'emitted' when an event occurs, and slots are event handler functions that respond to a corresponding signal.

5.3.1 – FormDialog

The FormDialog class serves as the base for this Qt application, taking no other Qt object as its parent. It inherits from `QDialog` and consists of a set of widgets representing parameters that can be altered. It behaves as a form, to be submitted or cancelled.

An instance of FormDialog is created within the main method and displayed using the `exec()` function. The nature of `exec()` ensures that the form must be filled or closed before continuing, as it sets the dialog to be modal. A signal of `Accepted` or `Rejected` must be emitted before the rest of the main method can execute.

Within the main class files, logic is implemented to collect and store the values each parameter upon form submission. A helper struct, `UserInput`, contains fields that allow these values to be transferred into the main execution of the system without reliance upon Qt types and classes. This is filled when the OK button is pressed, emitting the `Accepted` signal and making a call to `getUserInput()` –

```
FormDialog gui;
UserInput results;

if (gui.exec() == QDialog::Accepted) { // User presses OK.
	results = gui.getUserInput();
} else { // User presses cancel, escape, or 'X'.
	return 0; // Exit.
}
```

Figure 5.3.1 *– A code extract demonstrating the use of the signals to detect that a user has submitted the form, and act appropriately.*

The result of `getUserInput()` also updates the private `UserInput` variable belonging to this instance, and sets an internal flag `resubmit` to true, in order to indicate the dialog has been submitted at least once. This keeps the values stored in memory for the next execution of the dialog, after the rendering window closes. Upon reopening, a call is made to `setFormValues()`. The widgets within the form have their values set to those previously submitted, providing a more seamless experience.

A specific slot, `on_helpButton_clicked()`, has also been implemented for the class. As the function name suggests, this slot listens for the signal that the 'Help' button has been clicked. Upon receiving this signal, an instance of HelpDialog is created and executed.

Information about the contents, formatting and style of FormDialog is stored within `userinterface.ui` and its corresponding C++ header file `ui_userinterface.h`. The application Qt Designer was utilised to construct these files and create a GUI that considers user experience without investing a large period of time into learning the syntax of the API.

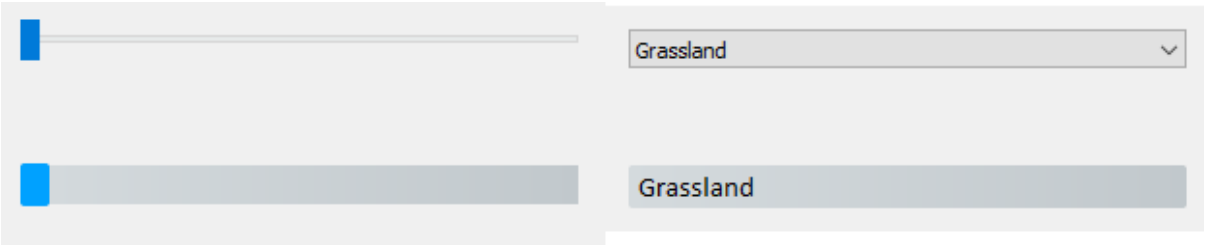


Figure 5.3.2 *– A comparison of the default slider and combo box styles to those used for FormDialog.*

Additional stylesheets have been applied to the base widgets in order to create a more visually appealing application. Sliders, spin boxes, combo boxes and buttons have all been modified to have a cohesive colour scheme and overall appearance.

5.3.2 – HelpDialog

The HelpDialog class inherits from both `QDialog` and its own user interface class, `Ui_helpdialog.h`. When an instance is created, its parent is set to FormDialog, as it spawns from a button click on the main form.

The purpose of HelpDialog is to display useful information to the user; it consists entirely of formatted label objects containing rich text, arranged into an easy to read layout. A header image has been added to the uppermost label to add visual interest and to ensure the whole dialog isn't just a block of text.

Initially, the dialog only contained information regarding use of the main form and what its parameters do. After some informal user feedback, a small section about controls was added, describing how to navigate while using the rendering window.

As this dialog is also modal, the user must either press the 'OK' button or exit manually in order to return to the main form. Nothing is submitted upon leaving the dialog, so the exact signal emitted does not matter.

5.3.3 – LoadDialog

LoadDialog is a simple dialog window inheriting from `QDialog`. It is independent of the other two Qt classes, with no parent object.

The sole purpose of this dialog is to give the user indication that the system is working on their request – the time taken to generate an instance of PerlinGrid can vary greatly with the submitted parameters and the user's hardware. Without visual feedback that generation is proceeding, the user may believe that the system is stuck or has crashed.

Within the main method, an instance of LoadDialog is displayed using `show()` after the FormDialog form is submitted. Using this function as opposed to using `exec()` makes the dialog non-modal, meaning code execution may continue in the background. It displays throughout the process of creating the grid and setting the appropriate parameters for the Scene instance, and is finally closed with `hide()` before the rendering call is made.

6. Testing and Evaluation

This section covers testing and the difficulties faced in doing so during this project, including unit tests and manual render tests. Performance evaluation is discussed, and checks are made against PCG metrics.

6.1 – Tests

6.1.1 – Unit Tests

Visual Studio's *C++ Unit Test Framework* was used to implement unit tests for this project. Severe difficulties and setbacks were experienced during the testing process, namely that the libraries being used did not seem to be compatible with the framework, and furthermore that an OpenGL context suitable for running tests was not producible. In short, any part of a class requiring functions from GLEW and GLFW could not be tested. Accessing their functions or features would result in all tests failing with a generic error regarding context being unable to be initialised.

Over the course of several months, methods of creating a context suitable to run the tests were attempted, but no progress was made. Further research was done into the matter, but little in the form of resources exist either online or within literature accessible through the university, with the only suggestion being to invest time in making a custom test framework and mocking the OpenGL pipeline. Similarly, requests for advice from departmental staff did not shed further light on the matter. Thus, the decision was made to abandon unit testing in order to focus more on the software system and manual testing.

Several unit tests were still created successfully, fully covering Perlin, and partially covering Camera and Window. Some other classes from the engine had their constructors tested without issue. Of the 16 test cases written, all pass. A full list of these can be seen in Appendix A.

6.1.2 – Manual Tests

The nature of computer graphics necessitates manual testing, as rendering is imprecise, and it is unlikely two runs of a rendering setup will produce identical output. The lack of consistency on a pixel by pixel scale would mean unit tests of the final result would most likely be meaningless, or at the very least fail frequently.

To properly ensure that a given input or setup of the system produces the desired output, without visual bugs, it must be examined by eye. This poses an issue for a procedural generator – the core

idea of the system is that there is theoretically no bound on the number of possible outputs. It would be infeasible to test every single combination of parameters.

Thus, a number of test cases have been designed that cover the broad functionality of the generator, and further test cases explore more specific combinations of parameters. The goal was to cover as many potential types of user input as is sensible and ensure that these give no unexpected bugs or undesirable visual output. All 48 manual test cases passed, and tables documenting the elements tested can be found in Appendix A.

6.2 – Performance

6.2.1 – System Load and FPS

As a graphics-based application, the load put on the system is a useful metric for measuring the efficiency of the engine, finding the limits of hardware compatibility, and from these two results adapting the software to work with a wider range of user systems.

Analysis of system strain identified a large memory leak (around a gigabyte) part way through implementation, which thankfully was quickly fixed. The program now uses an average of around 300mb once the scene is loaded, which is perfectly acceptable for modern 3D rendering on current hardware.

Results vary widely, however, due to the differing amounts of VRAM available on different GPU models. A GPU with a lower amount of VRAM results in the system putting heavier load on the RAM, with the maximum possible amount tested being around 900mb.

The observations were also utilised to make a 'low-end' compatibility mode, which reduces the size of the produced PerlinGrid object, thereby massively reducing the vertex count and easing the load on the GPU, VRAM and RAM.

FPS was also monitored as a performance metric. A target of 60fps was set, as most displays have a refresh rate of 60Hz or greater. The minimum acceptable framerate was set to 30fps, which is still present in some modern games and applications. This too contributed to the production of the compatibility mode.

The tables in Appendix B demonstrate the statistics identified across different graphics cards during normal use. CPU load is not examined, as the CPU used was not constant.

6.2.2 – Loading Times

Loading times were measured between the point of submitting the main form and the render loop beginning, therefore covering the length of time required to generate the grid. This was achieved by making two calls to `glfwGetTime()` and taking the difference of the values.

As octave count is the only factor that should affect load time – each new octave requires an extra noise calculation per vertex – this was repeated for each one. It helped in making the final decision to limit the maximum octave count to 10.

The measurements were performed on a GTX-1050Ti M model GPU. A different GPU may produce different timings, as all shader operations take place in the GPU, but the pattern for time increase per extra octave should remain roughly the same, as noise calculations are performed in the CPU.

The times range from 2.56 to 7.87 seconds for the high-end setting, while the range for the low-end settings is 0.62 to 2.45 seconds. A full table of results can be seen in Appendix B.

6.3 – PCG Metrics

As part of literature analysis for this project, a set of criteria for developing a good PCG system was identified in the article '*Procedural approach to volumetric terrain generation*' [16]. In section 2.1 of the article, a collection of statements regarding aspects such as structure, scalability and realism with corresponding scores are established.

As a measure of what was achieved, the final system was evaluated against these statements and given a score for each category. The final score was 11 of a possible 17. Considering the generator was built as an independent project and was not specifically tailored to these metrics, this is a respectable score. A full table of the criteria and the score the system matched can be seen in Appendix C.

7. Critical Appraisal

Here, the overall success of the project is evaluated against the original aims, including analysis of what could have been improved. The impact and context of the result is considered.

7.1 – Analysis of Objectives

7.1.1 – Implementation of the OpenGL Engine

I believe the final engine has a solid and reusable code base, as intended. All key functional aspects are there, namely a stable rendering window navigable by camera, facility to make lit meshes of different types, and a variety of shaders that can be further added to by anyone who learns GLSL. Relevant requirements from the original plan have mostly been fulfilled, with the exception of model loading and shadow mapping.

These missing features were not core to the system, but I believe perhaps with better foresight and planning, shadow mapping at least could have been implemented as well. Another feature that could have been expanded upon is the lighting – there is ambient and directional lighting, but the ability to use floating point lights would have also been a good feature to have.

These extras fell to the wayside as I began to focus more on the procedural generator – with the core in place at the end of the first semester, I made the decision to focus fully on combining the engine with the noise algorithm. Difficulties faced implementing this ended up leaving little time to return to the engine, outside of optimisations to the core classes.

Another factor for the missing features was the decision to add the Scene class. This was made fairly late into the project, but I think it was worth the development time – it removes repetition of code, for the internal benefit of the project, and makes the engine as a whole more accessible for external use. It helps demonstrate that all aspects of the engine can work smoothly together, and opens up the possibility for an end user to further develop their own 3D graphics and procedurally generated content.

Were I to develop another engine, or indeed even expand upon this one, I would place more of an emphasis on basic physics – most importantly, collision detection. This was not considered in the original project plan, and thus it didn't occur to me how greatly realism could benefit from it until a point where it was too late to dedicate time to it. I believe that alongside shadow mapping, this would be the next natural extension of this work.

7.1.2 – Implementation of Perlin Noise

The adaptation of Perlin's algorithm from Java to C++ went about as smoothly as expected, although one bug with the permutations vector persisted through much of the project and was not identified until the PerlinGrid class was being implemented. This delayed progress with that class somewhat, as I did not realise the fault lied in Perlin. Fortunately, it was an easy fix once identified – it was simply a matter of the latter values in the vector not being set correctly.

I placed an emphasis on making the original work more understandable and easier to follow, rather than directly lifting the original code, while also making small optimisations that make sense in the new language. This worked in the project's favour, as the better overall understanding gained helped with making the further developments and using the algorithm in the procedural generator.

The addition of octave noise greatly improved the usability of the algorithm and proved key to making interesting results from the generator. The style in which the function was implemented changed several times, including a switch to split 2D from 3D noise. The final version has functions with clear use cases and relies on class variables for seed, persistence and octaves.

7.1.3 – Implementation of the Procedural Generator

Development of the generator was initially slow. I struggled to come up with a way to efficiently generate meshes with noise values, mostly due to overcomplicating the concept in my own mind. Several ideas were drawn up, and initially the grid was going to be one large mesh – however, difficulty was experienced connecting each cell, particularly when transitioning between rows.

In the end, a compromise was made to have each row as its own mesh. This is still much more efficient that having each cell as an individual mesh vertex-wise but loses a lot of efficiency over the single mesh, as the seams between each row take double the vertices. By the end of the project, optimisations in other areas of PerlinGrid meant the system overall is still efficient with memory. However, given a second chance I would have worked on the generator earlier, to create a class that uses a more optimised mesh in terms of vertices. This would also lessen the load on the GPU.

7.1.4 – Augmentation of the Procedural Generator

The base generator was improved upon with several different approaches – for example, the addition of seeds and octave noise to the Perlin class naturally improves the range of results through parameter combinations. Similarly, the implementation of a skybox improves the sense of scale and realism in the final generated scene.

Within PerlinGrid itself, colour striation was made possible by means of the implemented struct `ColourMap`. This added another layer of customisability, as both colours and height boundaries can be set. A future user could create their own mappings, which means potential isn't limited to the ones added by default.

At the beginning of the project, plans were made to use a second algorithm, Worley noise, to add procedurally generated textures to the terrain. In addition to time constraints, I decided that this wouldn't work with the design direction taken for PerlinGrid. Opting for pure colour striation rather than developing a technique to not only load textures by height, but also add algorithmic textures, was more feasible and left more time for other features to be developed or further optimised.

7.1.5 – Designing the User Experience

Initially, implementing the user interface posed difficulty due to having to get a grasp of an entirely new API over halfway into the project. In hindsight, I should have researched the available APIs before beginning the project, as I had done with other aspects, and at least learned the basics. I stand by the decision to leave GUI implementation until last, however, as this allowed me to prioritise the core of the system first.

Once the learning curve was addressed, development proceeded at a much more reasonable pace. Studying Qt stylesheet syntax helped to make the final appearance of the interface much more professional, and while it is still quite simple I believe this works in its favour. There are quite a few parameters to control that a user may not have prior understanding of, so having a dedicated help menu and leaving the interface uncluttered should help to reduce frustration.

7.1.6 – The Overall Project

As a whole, I am proud of the progress made in this project. While not every initial requirement has been met, new, more sensible ones have been added and implemented throughout the process. Most issues faced were either a result of insufficient preparation or overestimating what could be achieved in the timespan, and a majority of these were addressed successfully or compromised upon sensibly.

As stated, given another opportunity at developing this system, I would make more reasonable estimates of the length of time needed for some features, and more comprehensive planning would be done in order to give more complex aspects the time they need.

7.2 – Commercial and Academic Context

With some further polish as discussed in the critical analysis, I believe the system could have potential commercial use. One of the aims of the project was to appeal to and aid game designers, and this is certainly still possible. In its current state, the system could very easily be used by concept artists as inspiration for map design, and with a few further tweaks – for example, exporting vertex data – could be used by designers with an active role in programming.

Perlin noise is certainly widely used in graphics, but the way in which it has been applied in this project (the method of creating the grid row by row with noise) is to the best of my knowledge unique. In this way, in an academic context, it could be used in comparisons to other methods, such as using a bitmap height map as a means of producing 3D terrain from noise. It could similarly be extended to show the results of other coherent noise algorithms, and in that way compare and contrast results.

The engine itself could be used in a teaching context, as an introduction to the graphics pipeline and OpenGL. A student could move from using the Scene class as a guide to making the low-level calls themselves; this would offer a gentler learning curve than learning OpenGL from the ground up.

7.3 – Personal Development

This project as a whole has been an eye-opening learning experience for me. I was aware that it would be tough, as I designed the project title and outcomes myself, with the knowledge there would have to be a large amount of research and learning done before the process began. In this pre-project time, I developed the basic OpenGL skills I needed to begin the engine.

These basics alone were not enough to carry me through the project – often, difficulties were faced with hard to identify bugs that simply stopped the rendering process. The process of manually tracking down these bugs not only solidified my knowledge of OpenGL and how different parts of its pipeline interact, it also allowed me to identify inefficient portions of code and therefore improve them. At the end of the project, I have a strong grasp of OpenGL, and can now tackle more advanced topics with the tools to more quickly identify issues and make progress.

In a similar vein, I have had to rapidly improve my skills with C++ in order to utilise OpenGL properly. This also took a lot of independent research, learning and experimentation. While I did end up taking the C++ module to support this learning, the majority of work was done in my own time, and through this I became a better independent learner and problem solver.

Though the decision to learn Qt so late into the project was not the best in hindsight, it did allow me to improve the speed at which I find and absorb new information. I learned to apply skills practically

earlier into the learning process, and in doing so improved my confidence with handling unfamiliar concepts.

Thinking more generally, I have learned to better plan and manage my time, and thus also have a better grasp of how long certain aspects of development take. From this, I can now more reasonably estimate how much can be completed in the scope of a project undertaken alone. The initial plan I had for the system and its requirements was far too ambitious, and even with perfect time management from the start, I would most likely not have completed all features listed.

Overall, I feel I have learned to better handle the issues faced throughout a programming project, gaining a better understanding of how to independently troubleshoot, and how to seek help if this is insufficient to solve the problem. I am now a more confident programmer and feel ready to tackle new projects, especially those where there are new skills to learn.

8. Conclusion

Procedural generation may not be a major topic in the field of computer science, but it is regardless fascinating and constantly evolving. Through this project, the idea of using coherent noise algorithms to power procedural generation was explored, and the final result gives a good indication of how these algorithms became key to the area.

One of the aims was to inspire game designers and show them the potential of noise to inspire their own work, and I believe this has been achieved, with the possibility to branch out and expand further in several ways.

A user may produce their own procedurally generated terrain through the main program, or may even opt to interact with the engine directly to take more control over the final result. Part of the reason I consider this project successful is the potential to use it in more than one way according to the needs and skillset of the end user.

In conclusion, the project has in my opinion met its major objectives, with all core features and some extras implemented. I am proud of the progress made, and consider the whole process a learning experience.

Bibliography

[1] *Procedural Content Generation Wiki*. Available at: http://pcg.wikidot.com (Accessed: 25th February 2019)

[2] *Engadget: Here's how 'Minecraft' creates its gigantic worlds*. Available at: https://www.engadget.com/2015/03/04/how-minecraft-worlds-are-made/ (Accessed: 25th February 2019)

[3] Giacomello, E., Lanzi, P.L., Loiacono, D. (2018). *DOOM Level Generation using Generative Adversarial Networks*. CoRR abs/1804.09154.

[4] *libnoise: What is coherent noise?* Available at: http://libnoise.sourceforge.net/coherentnoise/index.html (Accessed: 27th February 2019)

[5] Perlin, K. (2002). *Improved Noise reference implementation*. Available at: https://mrl.nyu.edu/~perlin/noise/ (Accessed: 28th February 2019)

[6] *Procedural Content Generation Wiki: Teleological vs. Ontogenetic*. Available at: http://pcg.wikidot.com/pcg-algorithm:teleological-vs-ontogenetic (Accessed: 28th February 2019)

[7] Zucker, M. (2001). *The Perlin noise math FAQ*. Available at: https://mzucker.github.io/html/perlin-noise-math-faq.html (Accessed: 28th February 2019)

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

[9] *OpenGL Wiki*. Available at: https://www.khronos.org/opengl/wiki/Main_Page (Accessed: 18th January 2019).

[10] Shreiner, D., et. al. (2013). *OpenGL programming guide: the official guide to learning OpenGL, version 4.3*. 8th ed. Ann Arbor, Michigan: Pearson Education.

[11] de Vries, J. *Learn OpenGL: Co-ordinate Systems*. Available at: https://learnopengl.com/Getting-started/Coordinate-Systems (Accessed: 2nd February 2019)

[12] de Vries, J. *Learn OpenGL: Final Thoughts*. Available at: https://learnopengl.com/In-Practice/2D-Game/Final-thoughts (Accessed: 25th March 2019)

[13] Vince, J. (2014). *Mathematics for computer graphics*, 4th edition. ed, Undergraduate topics in computer science. Springer, New York.

[14] de Vries, J. *Learn OpenGL: Cubemaps* Available at: https://learnopengl.com/Advanced-OpenGL/Cubemaps (Accessed: 24th February 2019)

[15] *A Simple GLFW FPS Counter* – r3dux.org (2016). Available at: https://web.archive.org/web/20161218030137/https://r3dux.org/2012/07/a-simple-glfw-fps-counter/ (Accessed: 20th February 2019). *This website has been removed from the internet since it was first accessed and must now be retrieved through an archive link.*

[16] Santamaría-Ibirika, A., Cantero, X., Salazar, M., Devesa, J., Santos, I., Huerta, S., Bringas, P.G. (2014). *Procedural approach to volumetric terrain generation*. The Visual Computer 30, 997–1007. https://doi.org/10.1007/s00371-013-0909-y

[17] Barrett, S. T. *stb single-file public domain libraries for C/C++*. Available at: https://github.com/nothings/stb (Accessed: 20th October 2018).

Appendices

A. Test Results

A1 – Unit Tests

Perlin:

Test Name	*Test Description*	*Passed?*
Perlin_DefaultConstructor	Test default constructor	Y
Perlin_SeededConstructor	Test seeded constructor	Y
Perlin_Noise	Check default noise values match Perlin's own	Y
Perlin_OctaveNoise	Check octave noise values remain consistent	Y

Camera:

Test Name	*Test Description*	*Passed?*
Camera_Constructor	Test constructor	Y
Camera_Direction	Check direction is calculated and normalised properly	Y
Camera_ViewMatrix	Check that correct view matrix is generated	Y
Camera_UpdatePosition	Test that key input can change position of camera	Y
Camera_UpdateDirection	Test that mouse input can change direction of camera	Y

Mesh:

Test Name	*Test Description*	*Passed?*
Mesh_DefaultConstructor	Test default constructor	Y

Window:

Test Name	*Test Description*	*Passed?*
Window_DefaultConstructor	Test default constructor	Y

Shader:

Test Name	*Test Description*	*Passed?*
Shader_DefaultConstructor	Test default constructor	Y

Mesh:

Test Name	*Test Description*	*Passed?*
Mesh_DefaultConstructor	Test default constructor	Y

Texture:

Test Name	*Test Description*	*Passed?*
Texture_DefaultConstructor	Test default constructor	Y

Material:

Test Name	*Test Description*	*Passed?*
Material_DefaultConstructor	Test default constructor	Y
Material_Constructor	Test constructor	Y

A2 – Manual Tests

1 (a – f). Testing for visual regularity with arbitrary noise configurations

Desired Result: No visual tearing or bugs, rows attach to each other seamlessly.

Seed	**Octaves**	**Persistence**	**Scaling**	*Visual Description*	*Passed?*
0	1	0.00	0%	Default noise view. All rows blended.	Y
3000	2	0.20	20%	All rows blended. No visual bugs.	Y
6000	4	0.40	40%	All rows blended. No visual bugs.	Y
9000	6	0.60	60%	Unnatural terrain type, but no visual bugs.	Y
12000	8	0.80	80%	Unnatural terrain type, but no visual bugs.	Y
15000	10	1.00	100%	Almost white noise, but no visual bugs.	Y

2 (a – e). Testing for appropriate lighting in biome and time combinations:

Desired Result: Lighting for biome is neither too bright nor too dark for each possible time option.

Biome	*Description of Result*	*Passed?*
GRASSLAND	All lighting appropriate. Brightening for EVENING/MIDNIGHT ok.	Y
MOUNTAINS	All lighting appropriate. Brightening for EVENING/MIDNIGHT ok.	Y
DESERT	All lighting appropriate. Darkening for MORNING/MIDDAY/NONE ok.	Y
GLACIER	All lighting appropriate. Darkening for MORNING/MIDDAY/NONE ok.	Y
HEATMAP	All lighting appropriate. Darkening for MORNING/MIDDAY/NONE ok.	Y

3 (a – n). Testing GUI behaviour

Item being tested	*Observed Result*	*As expected?*
	Main Form (FormDialog)	
Scalability	Main form scales appropriately and has a sensible minimum size	Y
Bar and spin box link	The two element types are linked for each variable. Changing the value of one correctly updates the paired item.	Y
Bar and spin box bounds	Values cannot be set outside of the set boundaries on either type.	Y
Biome and time combo boxes	Combo boxes can be interacted with correctly and all menu items are accessible.	Y
Checkbox	Checkbox can be interacted with and set/unset as many times as required.	Y
Signals	Closing or cancelling the form correctly emits a rejected signal. Submitting the form with OK correctly emits an accepted signal.	Y
Submission of values	Values submitted from main form are retrieved correctly according to user input.	Y
Reusability of values	Previously submitted values are successfully set in the form when it reopens.	Y
	Help Screen (HelpDialog)	
Scalability	Minimum size ensures no text is obscured. Dialog scales in a way that keeps all text in proportion.	Y
Modality	The main form cannot be interacted with for as long as the help dialog is open.	Y
Link to main form	Both closing and pressing OK returns you to the main form successfully.	Y
	Loading Screen (LoadDialog)	
Modality	Dialog is not modal.	Y
Interaction with other logic	Does not interfere with other instructions. Loading times are practically identical without the dialog present.	Y
Scalability	Scale of the dialog cannot be changed.	Y

4 (a – q). Testing Scene functionality (w.r.t. rendering)

Functionality being tested	*Observed Result*	*As expected?*
Adding coloured mesh	Coloured mesh renders with correct RGB values.	Y
Adding textured mesh with Texture object	Textured mesh renders with the texture supplied by the associated object.	Y
Adding textured mesh with texture from correct file path	Textured mesh renders with texture image generated from provided file path.	Y
Adding textured mesh with texture from incorrect file path	Mesh is rejected as the texture cannot be loaded. It is not rendered, and an error message is printed.	Y
Translating mesh	The mesh is moved the correct distance and direction specified by the vec3.	Y
Scaling mesh	The mesh is scaled according to the provided vec3. It is first translated and then scaled.	Y
Rotating mesh		Y
Adding PerlinGrid	PerlinGrid object renders as desired.	Y
Adding PerlinGrid while one already exists	The new PerlinGrid is rendered, the old one is deleted and cannot be seen in final scene.	Y
Setting PerlinGrid	PerlinGrid object renders as desired.	Y
Setting PerlinGrid while one already exists	The new PerlinGrid is rendered, the old one is deleted and cannot be seen in final scene.	Y
Setting a null pointer as a PerlinGrid while one already exists	The null pointer is rejected, and an error message is printed. The original grid is rendered.	Y
Changing time	Light properties are adjusted to match the enum value given.	Y
Changing biome	Grid colour properties are adjusted to match the enum value given.	Y
Adding custom ColourMap	Grid colour properties are adjusted to match the new ColourMap object.	Y
Adjusting light colour and intensity	Light colour and strength are visibly changed in the scene.	Y
Adjusting camera angle and position	Camera direction is correctly calculated from the given properties.	Y

5 (a – f). Testing other OpenGL functionality

Functionality being tested	*Observed Result*	*As expected?*
Cursor lock	While the OpenGL window is active, the cursor is locked to the window and can't leave its bounds.	Y
Keyboard controls: W, A, S, D	The four keys translate the camera position in the appropriate direction.	Y
Keyboard controls: Space, Up Arrow, Down Arrow	The keys translate the camera position in a purely vertical direction.	Y
Keyboard controls: Escape	Pressing escape causes the window to close.	Y
Mouse controls	The mouse can be used to rotate the camera, updating its direction.	Y
Skybox position	The skybox is infinitely distant regardless of camera movement. It does not approach or recede no matter how the position of the camera changes.	Y

B. Performance Results

B1 – System Load and FPS

On High-End Mode:

GPU	*FPS*	*Memory Usage (mb)*	*Average GPU Load (%)*
Intel HD Graphics 630	21.98	922.5	99.2
GTX-1050Ti M	66.76	74.7	97.8
GTX-970	88.23	50.7	42.3
GTX-1080	143.50	102.8	22.8

On Low-End Mode:

GPU	*FPS*	*Memory Usage (mb)*	*Average GPU Load (%)*
Intel HD Graphics 630	77.89	292.4	98.4
GTX-1050Ti M	230.60	118.9	88.5
GTX-970	327.78	97.7	11.1
GTX-1080	439.12	101.5	6.3

Figure B1 *– Tables of performance results for low and high-end run modes. 4 GPUs were tested across 3 computers. These tests were run with the framerate on the systems uncapped. Tests on capped systems will show a framerate of no higher than 60fps, as it is locked by the GPU.*

B2 – Loading Times

Octave Count	*High-End Load Time (s)*	*Low-End Load Time (s)*
1	2.0431	0.6168
2	2.5389	0.6949
3	2.6562	0.7394
4	2.8651	0.9393
5	3.2418	0.9883
6	3.6556	1.1989
7	4.6932	1.5188
8	5.2963	1.9760
9	6.6386	2.0955
10	7.8700	2.4504

Figure B2 *– Tables of loading times per mode and octave count.*

C. Metrics

Criteria	**Score Given**
Structure *Ability to obtain recognisable elements.*	1 – The results have independent coupled elements.
Interest *Ability to obtain interesting results to interact with.*	1 – The terrains are explorable.
Speed *Speed of obtaining the result.*	1 – Hybrid approach. Fast enough to be executable online but is run offline.
Usability *Ability to hide technical details from the user.*	1 – The parameters are related to the technique.
Control *Ability to allow the designer to alter the result.*	3 – The designer can specify all aspects of the result and correct them iteratively.
Extendibility *Ability to obtain different types of results.*	1 – The generator needs internal changes to obtain other types of results.
Scalability *Ability to obtain results at different scales/detail.*	1 – The generator can obtain results at different levels of scale and detail.
Realism *Ability to obtain plausible results.*	2 – Based on our reality, but not on a simulation.
Total:	**11 out of 17**

Figure C *– A table of the metrics set out in [16] and the score that is most appropriate for this system in each category. Criteria and score descriptions are either directly quoted or paraphrased from the article.*