University of Manchester
School of Computer Science
Project Report 2023

Generating Realistic and Efficient 3D Terrain Models

Author: Patrick Hume

Supervisor: Steve Pettifer

Abstract

Generating Realistic and Efficient 3D Terrain Models

Author: Patrick Hume

This report details the development of a terrain generation application implementing several algorithms to create realistic and optimised 3D terrain models. The application implements control point interpolation and fractal noise generation to shape terrain models and hydraulic erosion simulation to enhance the terrain with mountain ridges, valleys, rivers, and lakes. The application includes a texture generation which categorises terrain into rock, soil, grass, and water. Support is provided for exporting to OBJ and glTF files and a terrain optimisation algorithm is devised to simplify meshes based on their complexity. The evaluation of the application involves several aspects, including the realism of the generated terrain models, the efficiency and performance of the implemented algorithms, the effectiveness of the mesh simplification algorithm, and the usability of the user interface. The resultant application acts as an intuitive interface for users to model realistic, textured, optimised 3D terrain models which exhibit geographical features found in real-world terrain.

Supervisor: Steve Pettifer

Contents

1 Introduction **6**
- 1.1 Aims and Objectives . . . 7
- 1.2 Report Structure . . . 7

2 Background **8**
- 2.1 Terrain Generation . . . 8
  - 2.1.1 Heightmap-Based Terrain . . . 9
- 2.2 Interpolation . . . 9
  - 2.2.1 Interpolation Methods . . . 9
- 2.3 Noise Generation . . . 10
  - 2.3.1 Types of Noise . . . 10
- 2.4 Erosion Simulation . . . 12
- 2.5 Texturing . . . 12
- 2.6 Terrain Model Simplification . . . 13
  - 2.6.1 Simplification Methods . . . 13

3 Implementation **14**
- 3.1 Application Overview . . . 14
  - 3.1.1 Additional Requirements . . . 16
  - 3.1.2 Dependencies . . . 16
  - 3.1.3 Application Structure . . . 17
  - 3.1.4 Notable Classes . . . 17
- 3.2 User Interface . . . 18
  - 3.2.1 Main Toolbar . . . 18
  - 3.2.2 Application Command Line . . . 18
  - 3.2.3 Rendering Terrain Models in OpenGL . . . 21
- 3.3 Control Points Interpolation . . . 23
  - 3.3.1 Interpolating Control Points . . . 24
  - 3.3.2 Interacting with Control Points . . . 26
- 3.4 Detailing Terrains with Generated Noise . . . 27
  - 3.4.1 Fractal Noise Algorithm . . . 27
  - 3.4.2 Parameters . . . 30
- 3.5 Simulating Hydraulic Erosion . . . 30

3.5.1 Assumptions . . . 31
3.5.2 Algorithm Overview . . . 31
3.5.3 Experiments . . . 37
3.6 Generating Rivers . . . 39
3.6.1 Improving River Quality . . . 39
3.7 Generating Lakes . . . 40
3.7.1 The Lake Smoothing Algorithm . . . 41
3.7.2 Generating Soil . . . 42
3.8 Texturing . . . 43
3.8.1 Classifying Terrain . . . 43
3.9 Mesh Optimisation . . . 46
3.9.1 Patches and Tessellating in OpenGL . . . 46
3.9.2 Detail-Based Tessellation Algorithm . . . 46
3.9.3 Tessellation Results . . . 47
3.9.4 Error Threshold Experimentation . . . 47
3.10 Exporting Terrain Models . . . 49
3.10.1 Retrieving Tessellated Models . . . 49
3.10.2 Exporting to OBJ . . . 50
3.10.3 Exporting to glTF . . . 50

4 Evaluation **53**
4.0.1 Visual Quality . . . 53
4.0.2 Mesh Optimisation . . . 55
4.0.3 Efficiency and Performance . . . 55
4.0.4 Usability . . . 56

5 Reflections **57**

Bibliography **58**

List of Figures

2.1 5x5 heightmap data represented as an image. 9
2.2 White noise, represented as a greyscale image. 10
2.3 Value noise, represented as a greyscale image. 11
2.4 Fractal noise, represented as a greyscale image. 12

3.1 Application pipeline. 14
3.2 Modelling sub-stages. 14
3.3 Modelling sub-stages. 15
3.4 Interactions between classes and external libraries. 17
3.5 Application toolbar. 18
3.6 Example command inputs (black), outputs (blue), and errors (red). 20
3.7 5x5 heightmap data represented as a textured mesh. 21
3.8 5x5 heightmap data represented as an untextured mesh. 21
3.9 5x5 heightmap mesh visualised as triangle strips. 22
3.10 50x50 mesh sampling a 5x5 heightmap. 23
3.11 Four control points defining a terrain. 25
3.12 User placing a control point. 26
3.13 User adjusting a control point height. 27
3.14 4th octave of value noise. 29
3.15 8th octave of value noise. 29
3.16 Fractal noise, represented as a greyscale image. 29
3.17 Fractal-noise-generated heightmap represented as a mesh. 30
3.18 Mesh textured with the image generated by the normal function. 32
3.19 Raindrop paths traced by white lines. 34
3.20 Mesh textured based on levels of erosion (red) and deposition (blue). 36
3.21 Mesh textured based on levels of erosion (red) and deposition (blue). 36
3.22 Maximum drop velocity experiment results. 38
3.23 Terrain mesh textured using the river heightmap. 39
3.24 Terrain mesh textured using the re-configured river pass data. 40
3.25 Terrain mesh textured using the lake heightmap. 40
3.26 Water flow between neighbouring water levels. 41
3.27 Terrain mesh textured using the smoothed lake heightmap. 42
3.28 Before applying blurred terrain. 43
3.29 After applying blurred terrain. 43

3.30 Textured terrain representing areas soil (light). 44
3.31 Textured terrain representing areas of grass (light). 44
3.32 Textured terrain representing areas of water (light). 45
3.33 Textured terrain. 45
3.34 Wireframe view of tessellated patch edges. 46
3.35 Tessellated terrain model. 47
3.36 Tessellated terrain model (wireframe view). 47
3.37 Visual mesh simplification results. 48
3.38 Exported terrain object viewed in MacOS's Preview. 50
3.39 glTF model viewed in an online glTF viewer. 51
3.40 glTF model viewed with normal vectors in an online glTF viewer. 51
3.41 glTF model viewed, in a wireframe view, in an online glTF viewer. 52

4.1 Application-generated terrain labelled by geographical feature. 53
4.2 Application-generated terrain labelled by terrain type. 54
4.3 Application-generated terrain tessellated by complexity. 55
4.4 Dropdown buttons and sliders under the application toolbar. 56

List of Tables

3.1 Result set for the triangle reduction experiment. 49

Chapter 1

Introduction

Terrain models are digital representations of landscapes. They are an important aspect of several fields such as digital art, video games, virtual reality, and film. Terrain models are often used to simulate earth-like terrain to offer a sense of realism and immersion to the viewer. The visual similarity between 3D terrain models and the real world is therefore critical to their success. This report covers the entire terrain model generation process from model generation, to texturing, to simplifying and exporting.

This report begins by considering a number of approaches to generating terrain models. Many geographical patterns observed in nature can be mimicked using algorithms, such as noise generators. Likewise, many processes that give rise to these patterns can be mimicked too. For example, the smoothness of rolling hills could either be achieved via applying a smoothing algorithm to a terrain model, or by simulating the real-world processes that give rise to rolling hills. Many geographical features (including rolling hills) are caused by hydraulic erosion (the process by which terrain is shaped by water over time). Therefore, one method to increase the realism of terrain models is to simulate this erosion process, forming many of the same features as real-world erosion, such as mountain ridges and valleys. This report explores the effectiveness of these methods in generating realistic terrain.

This report also covers several terrain texturing methods, including one approach which uses data gathered from the hydraulic erosion simulation to identify and texture rivers and lakes, further increasing the realism of the terrain model.

Allowing users to export terrain models is a critical feature of terrain generation applications, enabling users to use their terrain outside of the application. The research in this report concludes by investigating the effectiveness of mesh simplification algorithms, demonstrating a detail-based tessellation approach implemented in the application.

I started this project by researching current terrain generation techniques including interpolation, noise generation, and erosion simulations. I then began developing a C++ application using OpenGL which implements a selection of terrain-generation algorithms for users to model, enhance, texture, simplify, and export 3D terrain models. The result is an application which acts as an intuitive interface for users to hand-model realistic, textured, simplified 3D terrain models which exhibit real-world geographical features of mountains, valleys, lakes and rivers.

1.1 Aims and Objectives

This project has the following aims and objectives:

- To research and evaluate existing techniques for terrain generation including interpolation, noise generation, and erosion simulations.
- To design and develop a C++ application for terrain generation using the techniques identified during research.
- To apply Gaussian RBF interpolation to control points to allow users to control the overall shape of the terrain.
- To implement fractal noise generation to add roughness to the terrain.
- To simulate hydraulic erosion to produce additional natural terrain features such as mountain ridges, valleys, rivers, and lakes.
- To implement texturing techniques for the 3D terrain models to categorise terrain as rock, soil, grass, or water and colour these features appropriately.
- To develop a mesh simplification algorithm to optimise the terrain model whilst maintaining a high level of detail.
- To provide users with the ability to export the 3D terrain models for use in other applications.
- To evaluate the effectiveness of the implemented techniques in producing realistic terrain models.

1.2 Report Structure

This report contains five chapters:

- Chapter 1 presents an introduction to the project.
- Chapter 2 presents an overview of terrain modelling terms and techniques.
- Chapter 3 presents the C++ implementation of the application.
- Chapter 4 presents an evaluation of the implementation.
- Chapter 5 presents the project conclusions and reflections.

Chapter 2

Background

2.1 Terrain Generation

Terrain generation is the process of creating a digital model that represents a 3D landscape or topography. Several methods of terrain generation exist, including manual modelling, procedural generation, and erosion simulation. Procedural generation involves the use of algorithms and mathematical functions, such as noise and interpolation, to generate terrain; it is a popular method particularly in "open-world" video games where vast amounts of terrain are required. Manual modelling involves the creation of terrain, by hand, within a 3D modelling application. Erosion simulation methods aim to shape terrain models by mimicking the real-world processes which give rise to many geographical features. This report aims to combine manual modelling, procedural generation, and erosion simulation methods in an intuitive application where users can design terrain models. The underlying data structure used to generate the terrain models is the heightmap.

2.1.1 Heightmap-Based Terrain

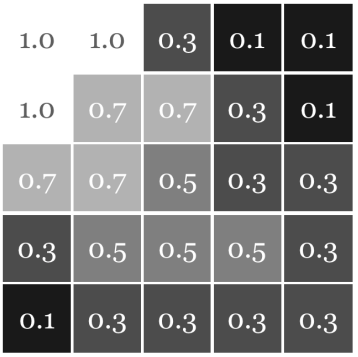


Figure 2.1: 5x5 heightmap data represented as an image.

Heightmaps are greyscale images which provide a discrete digital representation of a topographic surface. They can be populated with real-world terrain data using digital elevation models [HB04], or artificially generated using terrain generation methods. Figure 2.1 illustrates a heightmap of width and height 5.

2.2 Interpolation

Interpolation can be used to generate terrain models. Given a finite set of control points (through which terrain must pass), an interpolation algorithm can fill in the remaining space. This is an intuitive way to enable users to model terrain and therefore makes interpolation a useful tool in terrain generation. The resultant terrain is smooth and therefore lacks realism, this can be handled by treating the interpolated terrain as a "core" to be layered with detail, such as fractal noise, in order to provide an impression of realistic non-uniformity across its surface.

2.2.1 Interpolation Methods

Two interpolation methods are capable of producing smooth terrain: Natural neighbour interpolation and Gaussian radial basis function (RBF) interpolation.

Natural Neighbour Interpolation

Natural neighbour interpolation takes the weighted sum of several known neighbours to estimate unknown values. The result is a smooth terrain, capable of modelling steep cliffs and flat planes. Natural neighbour interpolation can, however, be computationally intensive when interpolating between many points.

Gaussian RBF interpolation

Gaussian RBF interpolation reconstructs a function given some scattered data [Buh03], and has the advantage of being more computationally efficient than Natural neighbour interpolation. However, it is not capable of creating sharp features in the data, such as cliffs. In the interest of computational efficiency, and operating under the assumption interpolation will only be used to form the general peaks and troughs of terrain models (and not sharp details), Gaussian RBF interpolation was implemented in the application.

2.3 Noise Generation

Layered noise is a useful method for generating and increasing the realism of terrain models [HMP17]. The random patterns exhibited by noise can be used to add roughness to terrain models, increasing their sense of realism by mimicking the non-uniformity of real-world terrain.

2.3.1 Types of Noise

This report implements a fractal noise approximation of terrain, generated using layered value noise. This section aims to provide an overview of these noise types but is not an exhaustive list. Advanced noise algorithms include Perlin noise, with further improvements to noise quality such as Simplex noise [Gus05].

White Noise

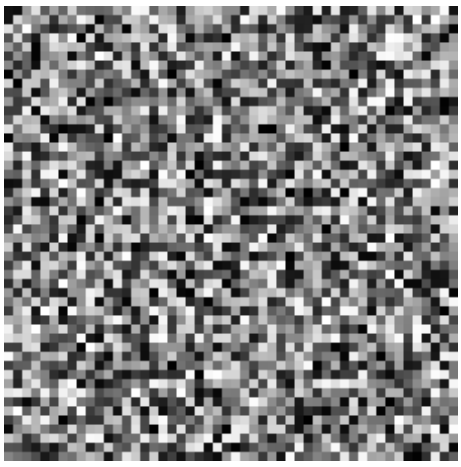


Figure 2.2: White noise, represented as a greyscale image.

Figure 2.2 illustrates a sample of white noise generated using a random number generator. White noise consists of independent random variables with a uniform probability

distribution, which is to say that each entry in white noise is truly random, and due to this, it appears to have no visual coherency. Without any coherency between values, it is incapable of simulating realistic, coherent terrain. It does, however, act as a basis for generating a form of noise more useful to terrain generation, value noise.

Value Noise

Figure 2.3: Value noise, represented as a greyscale image.

Value noise, illustrated by Figure 2.3, can be thought of as interpolated white noise and, when considered as a heightmap, appears as a smooth series of peaks and troughs. The interpolation method used to generate the value noise determines its character. Bi-linear interpolation produces noise with sharp angles, whereas bi-cubic interpolation produces a smoother result. The resultant noise is coherent and appears closer to real terrain, but it is still uniformly random and visually uninteresting. To mimic the diversity of real terrain, several octaves of value noise can be layered to form fractal noise.

Fractal Noise

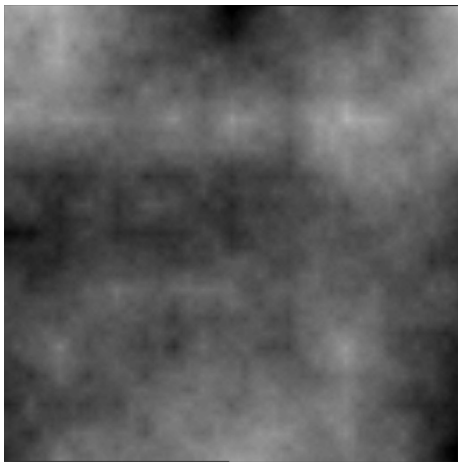


Figure 2.4: Fractal noise, represented as a greyscale image.

Fractal noise, illustrated by Figure 2.4, is generated by layering several octaves of value noise to create a coherent surface that exhibits both major and minor peaks and troughs. Fractal noise can be prone to visual artefacts, however, such as axis-aligned ridges and valleys.

2.4 Erosion Simulation

A greater level of realism can be achieved in terrain models by simulating hydraulic erosion, a method capable of improving terrains generated by fractal noise [MKM98, BF02].

Hydraulic erosion is the natural process by which terrain is shaped by water over time, this includes the crashing of waves against a shoreline giving rise to cliff faces, the meandering of rivers forming oxbow lakes, or rainfall eroding rocks to gradually shape mountain ridges. This single process is responsible for many of the features to be desired in a 3D terrain model, and so simulating the process is a popular method for heightening a model's sense of realism. It is for this reason that hydraulic erosion simulations are a popular tool in industry terrain generators such as *World Machine* and *Terragen*. Note that the computational effort required by hydraulic erosion simulations makes them less suitable as terrain size increases. One method of increasing performance is to perform the hydraulic erosion simulation on the GPU [MDH07].

2.5 Texturing

Texturing is an important aspect in increasing the realism of terrain models [DBH00]. By adding surface textures to a terrain model, the illusion of different materials can be created.

Chapter 3.8 investigates the generation of textures using an algorithm based on the hydraulic erosion simulation to colour terrain identified as rock, soil, grass, and water appropriately. An advanced texture generation method includes the use of machine learning. Generative adversarial networks (GANs) can be trained using digital elevation models and satellite imagery to automatically texture terrain models [SW19].

2.6 Terrain Model Simplification

The complexity of a terrain model determines the speed at which it can be rendered. A naive approach to mesh generation is to construct a rectangle at each set of four neighbouring heightmap entries. However, this results in large 3D object files which greatly over-represent simple sections of terrain, for which fewer triangles would suffice. It is therefore necessary to optimise terrain models prior to exporting, a process for which several methods exist.

2.6.1 Simplification Methods

Reducing Mesh Resolution

One approach is to reduce the resolution across the entire mesh, which can be done by skipping heightmap entries when generating the mesh. However, terrain models often vary in complexity across their surface and so by correctly representing one portion of the terrain, such as flat plains, other sections may become under-represented, such as mountain ranges. It is therefore necessary to devise a method which allows for a variable resolution across the mesh.

Vertex Addition and Subtraction

One method to achieve a variable resolution is vertex addition and subtraction. Vertex addition and subtraction involves repeatedly adding and subtracting vertices to and from the mesh in order to reach an optimal solution which minimises both the error of the mesh representation and the number of triangles used [GH97].

Tessellation

Another method to achieve a variable resolution involves mesh tessellation, the process of subdividing meshes. By separating the terrain into a grid of square patches and determining a maximum tessellation level for each patch, an appropriate resolution for each area of the terrain can be generated. OpenGL supports tessellation shaders [**?**] and so, due to its relative ease of implementation, a detail-based mesh tessellation algorithm was implemented in the application. Detailed in Chapter 3.9.

Chapter 3

Implementation

I decided to implement the application in C++, rendered using OpenGL. Choosing a compiled language was deemed an appropriate choice considering the computationally demanding nature of the hydraulic erosion simulation. The low-level nature of C++ would also allow for more control over application memory which was necessary for writing a performant application.

3.1 Application Overview

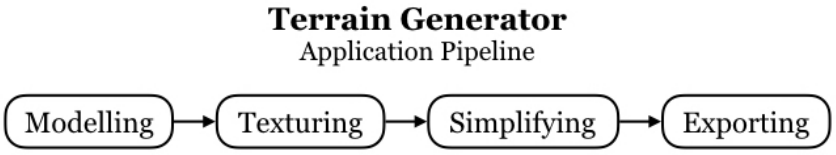


Figure 3.1: Application pipeline.

The implementation can be thought of as the following four-stage pipeline, illustrated in Figure 3.1: modelling, texturing, simplifying, and exporting.

The Modelling Stage

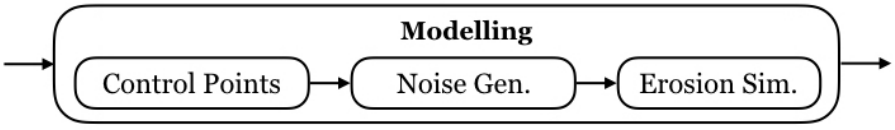


Figure 3.2: Modelling sub-stages.

The modelling stage concerns the topography of the terrain. Figure 3.2 illustrates the three sub-stages of the modelling stage, these are control point modelling, noise generation, and erosion simulation. Control-point modelling is how users will be able to form the overall shape of the terrain, by placing control points. Gaussian RBF interpolation is used to create a smooth and coherent terrain from these points. After this, fractal noise is generated based on user-defined parameters, this noise is layered on top of the existing model to add roughness to the terrain. This stage helps add a bias to the flow of raindrops in the coming erosion simulation, encouraging the formation of distinct features, such as ridges, in the terrain. At the end of the modelling stage, the terrain is passed on to the hydraulic erosion simulator. The erosion simulation involves simulating several hundred thousand virtual raindrops travelling down the terrain, collecting and depositing sediment to eventually form identifiable geographical features.

The Texturing Stage

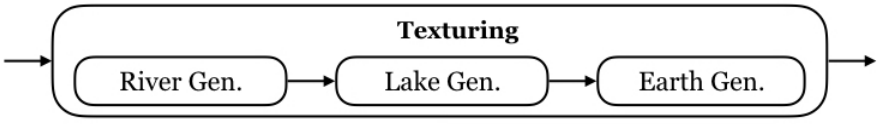


Figure 3.3: Modelling sub-stages.

Figure 3.3 shows the three sub-stages of the texturing stage. These are river generation, lake generation, and earth generation. River generation adapts the erosion simulation to identify river routes and generates a river texture map. This is combined with the lake generation algorithm, which identifies the destinations of rivers to approximate the locations of lakes on the terrain. The third sub-stage, the earth generation stage, uses an algorithm based on terrain incline, and sediment level to categorise the terrain into soil, grass or rock. The algorithm combines the river data, lake data, and earth data to generate an appropriately coloured texture image.

The Simplifying Stage

The simplifying stage involves the optimisation of the terrain mesh by reducing the number of vertices used to represent it. Here, the model is divided into a grid of patches which are simplified, using a tessellation algorithm, to fit the shape of the terrain.

The Exporting Stage

The exporting stage implements the exporting to two 3D object file formats, OBJ and glTF, allowing users to transfer their textured, optimised terrain models outside the application.

3.1.1 Additional Requirements

There are several requirements contributing to the utility and usability of the application. One such requirement is the inclusion of a 3D environment where users can view and edit their terrain from different angles. It is also necessary that a menu be provided to separate the application into distinct and intuitive sections. The application contains many internal parameters, some of which require experimentation to determine (particularly those used in the erosion simulation algorithm). As the application increases in size, the C++ compilation time increases; it is, therefore, necessary to implement a system which allows the modification of internal parameters, without recompilation of the program, to streamline the experimentation process.

3.1.2 Dependencies

The implementation relies on five external libraries which handle rendering, user input, text rendering, image file reading/writing, and matrix decomposition.

- *OpenGL* - An open-source graphics rendering library [Ope17]. This was used to render the application.
- *GLFW* - An open-source, multi-platform library for developing OpenGL applications, used for creating the window and detecting mouse and keyboard events.
- *FreeType* - An open-source C/C++ font rasterisation library, used for rendering UI text.
- *STB* - An open-source C/C++ audio and graphics library, used for reading and writing image files.
- *Eigen* - An open-source C++ library for linear algebra, matrix and vector operations, and more [GJ+20]. This was used to implement the interpolation algorithm.

3.1.3 Application Structure

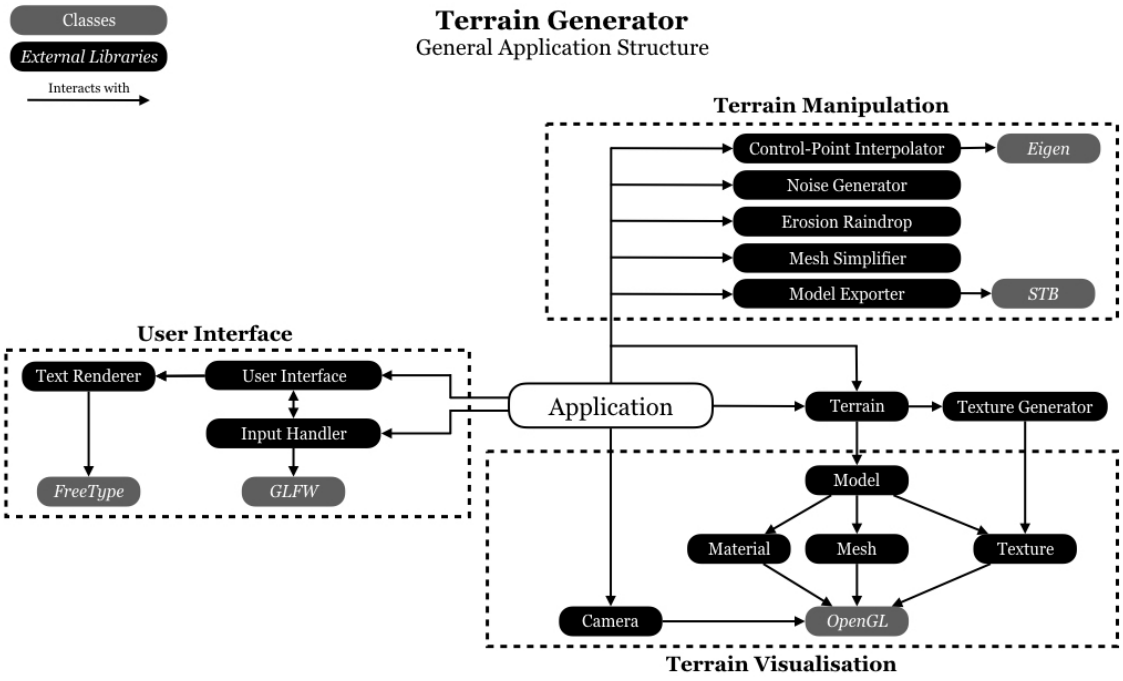


Figure 3.4: Interactions between classes and external libraries.

A diagram representing the interactions between classes and external libraries is shown in Figure 3.4.

3.1.4 Notable Classes

The implementation consists of a number of classes, most notably:

- *Terrain* - the 2D terrain heightmap.
- *Raindrop* - the raindrop used to simulate hydraulic erosion.
- *RBFInterpolator* - implements Gaussian RBF interpolation of control points.
- *NoiseGenerator* - implements the generation of fractal noise.
- *Simplifier* - implements the detail-based tessellation of terrain models.
- *ModelExporter* - implements the exporting of terrain models to OBJ and glTF files.
- *TextureGenerator* - implements the generation of terrain textures.

3.2 User Interface

The user interface was designed with simplicity in mind. The general application features are accessible via the main toolbar running across the top of the application window, whereas advanced features can be accessed via the application command line.

3.2.1 Main Toolbar

Figure 3.5: Application toolbar.

The overall design of the toolbar (shown in Figure 3.5) mirrors the underlying application pipeline. The application is separated into five views, navigable via the toolbar. The implementation consists of a number of classes, most notably:

- *File* - where the user can create, load, save, and export terrain to object files.
- *Model* - where the user can shape the terrain, add noise, and import heightmap images.
- *Erode* - where the user can perform the hydraulic erosion simulation.
- *Tessellate* - where the user can adjust mesh optimisation settings.
- *View* - where the user can cycle through custom render modes.

3.2.2 Application Command Line

The application command line allows internal parameters to be set from a text interface in the application window, its primary use is to streamline the conductance of experiments during the application's development, but the help command provides a listing of the available

commands for advanced users to experiment with. The application command line was implemented as a map of command strings and function pointers. For example, the input string *"get position"* could be mapped to the function *glm::vec3 getCamPos()*. The commands are listed and defined in the file commands.txt, using regular expressions.

Listing 3.1: Example commands.txt input to define a set camera position command.

```
// ----TYPES----
    <position>  -> (float) (float) (float)
// ----COMMANDS----
//  Function name  :  Command
    setCamPos      :  set position <position>
// ----REGEX----
    (float)     -> ((?:\.\d+)|\d+(?:\.\d+)?)
```

Listing 3.1 illustrates an example of the contents of the file input, here defining a command of the form "`set position <position>`". A string substitution is performed to replace "`<position>`" with "`(float)`", and then a second substitution is performed to replace each "`(float)`" with the regular expression "`((?:\.\d+)|\d+(?:\.\d+)?)`". The resultant string is a regular expression to match the original command, with each matched group corresponding to each command argument.

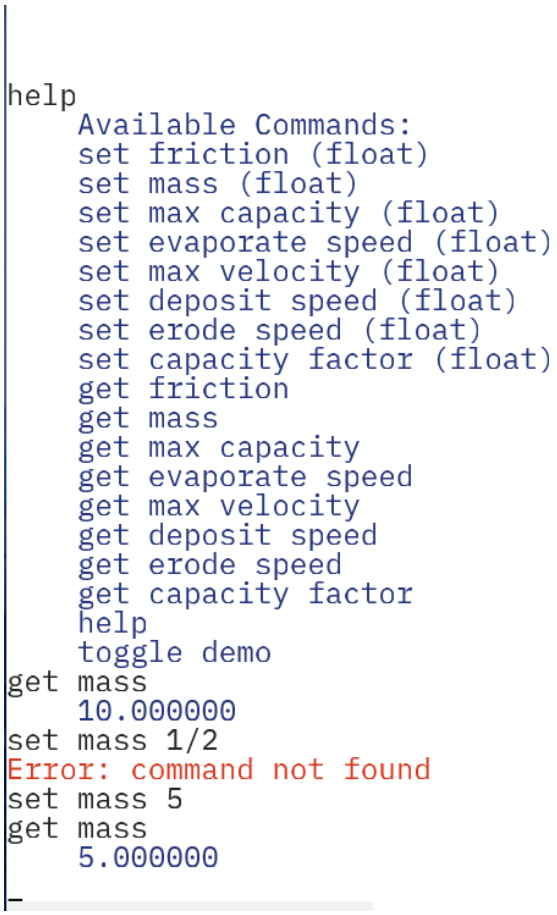


Figure 3.6: Example command inputs (black), outputs (blue), and errors (red).

Figure 3.6 shows a set of example commands being executed using the application command line. The first command is used to list all possible commands, the second command tests the input of an unidentified command, and the third and fourth commands demonstrate the value-getting and setting capabilities of the implementation.

3.2.3 Rendering Terrain Models in OpenGL

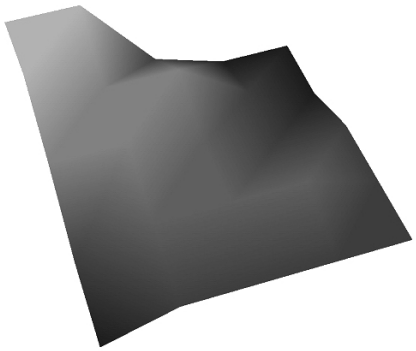


Figure 3.7: 5x5 heightmap data represented as a textured mesh.

Figure 3.7 shows a textured mesh constructed using the heightmap in Figure 2.1. This rendering and all other renderings in this report have been performed using OpenGL. OpenGL provides a wide range of functionality for rendering 3D geometry, and therefore there exist several approaches for rendering terrain models. Three approaches will be briefly compared here: Triangle-Based-Meshing, Triangle-Strip-Based-Meshing, and Shader-Based-Meshing.

Triangle-Based Meshing

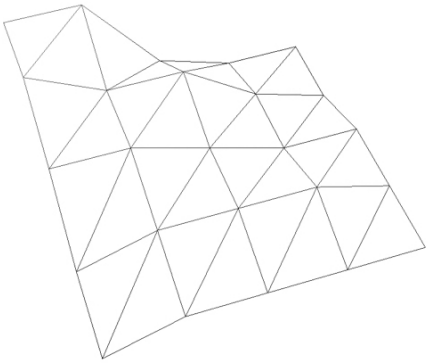


Figure 3.8: 5x5 heightmap data represented as an untextured mesh.

Triangle-based meshing involves iterating over a heightmap, generating pairs of triangles to connect each set of four neighbouring entries. The x and y positions of each vertex are determined by the relative position across the heightmap, whereas the y position is determined by the data entry at that heightmap index. The result is an exact 3D terrain representation

of the heightmap, illustrated by Figure 3.8, where each vertex represents a heightmap entry. This method is the by far the most simple but is also the slowest, this method can be improved by batching triangles together into strips.

Triangle-Strip-Based Meshing

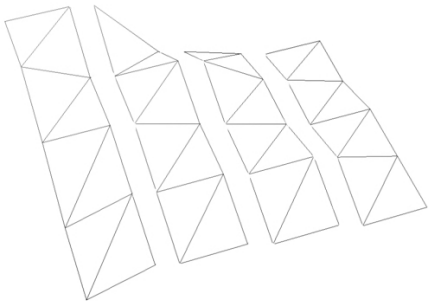


Figure 3.9: 5x5 heightmap mesh visualised as triangle strips.

A similar, yet faster approach involves generating triangle strips to represent the heightmap in rows, this concept is illustrated by Figure 3.9. By batching the vertices into triangle strips, the rendering process is accelerated, whilst still keeping the resolution of the model 1:1 with the heightmap. This approach was initially used in the terrain interpolation section of the program, the hydraulic erosion simulation, however, required a more performant approach involving mesh tessellation.

Shader-Based Meshing

Instead of constructing a mesh from heightmap data, a generic mesh can be used and shaped inside the vertex shader. This can be accomplished by rendering a flat plane and, inside the vertex shader, using the x and y position of each vertex to sample a heightmap texture, setting the z position to the sampled value.

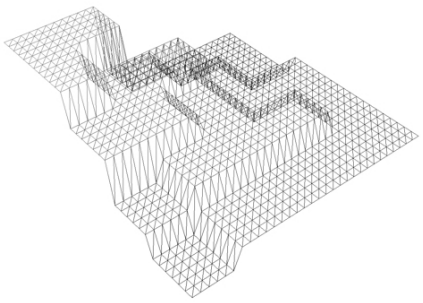


Figure 3.10: 50x50 mesh sampling a 5x5 heightmap.

Sampling a heightmap allows the shader to construct meshes from heightmaps regardless of their size, this can be demonstrated by sampling a 5x5 heightmap with a 50x50 mesh, shown in Figure 3.10. This method also removes the need to reconstruct a new mesh with each change to the heightmap, instead only the sampled texture is updated. All of the above methods were implemented during the development of the application, although, due to the performance advantages, shader-based-meshing replaced both prior methods. However, the shader-based-meshing approach does complicate the model exporting process. This is because outside of the shader program the mesh is still a flat plane, the mesh must therefore be retrieved from the shader program. A method for exporting shader-based meshes is detailed in section 3.10.

3.3 Control Points Interpolation

When creating terrain models, digital artists often have certain requirements, such as the placement of key features such as mountain peaks, valleys, and lakes. Whilst fractal noise generation algorithms are capable of creating varied terrain, and users are able to modify the general behaviour of the algorithm (terrain amplitude and roughness), the user is not able to place individual features. As a result, using noise algorithms alone leaves the artist unable to create the terrain they envision. The factor of user control can be implemented in several ways, one method for introducing user control over the terrain model is the use of a 3D paintbrush tool, allowing the user to interact with the terrain using the mouse cursor, providing the user with a set of tools to increase and decrease heights across the terrain. However, this approach treats the terrain as a map of unconnected heights, and the result can lack spatial coherency (i.e. the terrain is not smooth), leading to a landscape that appears unconnected and unnatural. The chosen approach was instead to allow the user to set a number of control points, through which the terrain must smoothly pass. Rather than taking time to hand-form the landscape using a 3D brush tool, the implementation allows the user to set the overall shape of the terrain with a minimal number of mouse cursor interactions.

3.3.1 Interpolating Control Points

To provide a responsive application, the terrain requires re-interpolation as the user modifies a control point. It is therefore important that the interpolation method be computationally efficient. For this reason, Gaussian RBF interpolation was the chosen interpolation method due to its speed. Gaussian RBF interpolation struggles to model sharp features such as cliffs but was implemented nonetheless with the assumption that users require only to model the general shape of the terrain.

Interpolation Equations

$$c_n = \begin{pmatrix} x_n \\ y_n \\ z_n \end{pmatrix} \tag{3.1}$$

Suppose that c_n in Equation 3.1 contains the 3D position vector of a control point placed by the user.

$$p_n = \begin{pmatrix} x_n \\ y_n \end{pmatrix} \tag{3.2}$$

For convenience, the 2D position vector which omits the control point's height will be referred to as p_n, as seen in Equation 3.2.

$$K(p, p^{'}) = \exp\left(-\frac{\left\| p - p^{'} \right\|^2}{2\sigma^2} \right) \tag{3.3}$$

In order to calculate the effect of each control point on its surroundings, it is necessary to calculate a measure of influence between two positions. To calculate this influence, the radial basis function kernel K (see Equation 3.3) is used, providing a measure of similarity between two positions. This similarity will determine the level of influence between each pair of control points.

$$\begin{bmatrix} K(p_0,p_0) & K(p_1,p_0) & \dots & K(p_n,p_0) \\ K(p_0,p_1) & K(p_1,p_1) & \dots & K(p_n,p_1) \\ \vdots & \vdots & \ddots & \vdots \\ K(p_0,p_n) & K(p_1,p_n) & \dots & K(p_n,p_n) \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} z_0 \\ z_1 \\ \vdots \\ z_n \end{bmatrix} \tag{3.4}$$

As illustrated by Equation 3.4, given a vector of control point heights z_0 through z_n, and a square matrix of pair-wise similarities between control point positions p_0 through p_n, it is possible to solve for control point weights w_0 through w_n. Each weight is a scalar measure of the control point's elevation and influence over other points. In order to solve for these

weights, it is necessary to perform a matrix decomposition.

The Eigen C++ library provides the following three functions for matrix decomposition:

- *FullPivHouseholderQR* - Householder rank-revealing QR decomposition.
- *ColPivHouseholderQR* - Householder rank-revealing QR decomposition.
- *HouseholderQR* - Householder QR decomposition.

According to Eigen documentation, these methods are listed in increasing order of speed and faster methods are less numerically stable [GJ+20]. However, numerical accuracy is not a critical factor for interpolating terrain, so the *HouseholderQR* method was implemented in the interest of performance. Suppose a set of 2D positions u_0 through u_n with unknown heights k_0 through k_n.

$$\begin{bmatrix} K(p_0,u_0) & K(p_1,u_0) & \dots & K(p_n,u_0) \\ K(p_0,u_1) & K(p_1,u_1) & \dots & K(p_n,u_1) \\ \vdots & \vdots & \ddots & \vdots \\ K(p_0,u_n) & K(p_1,u_n) & \dots & K(p_n,u_n) \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} k_0 \\ k_1 \\ \vdots \\ k_n \end{bmatrix} \quad (3.5)$$

Given the set of solved weights, Equation 3.5 allows for the solution of k_0 through k_n. The implementation generates a solution for every entry of the heightmap, generating a smooth terrain based on a set of user-defined control points.

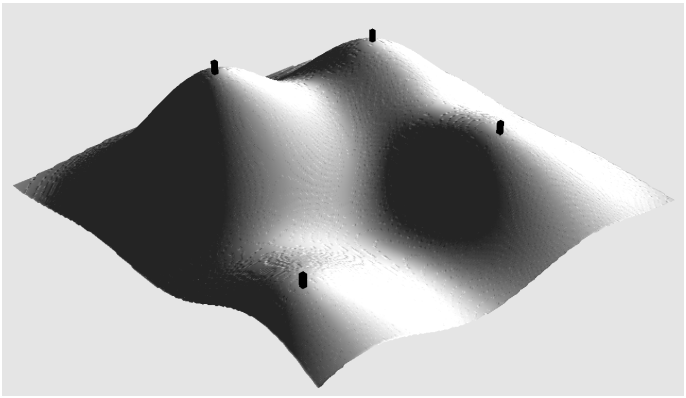


Figure 3.11: Four control points defining a terrain.

Figure 3.11 shows a rendering of four control points interpolated and rendered as a terrain model, verifying the functionality of the implementation. In order for users to place control points themselves, it is necessary to provide mouse cursor interaction with the 3D viewer.

3.3.2 Interacting with Control Points

The user is able to interact with the 3D viewer using the mouse cursor. This is done by casting a ray from the camera (at position c in direction d) and solving for the intersection $(x, y, 0)$ with the plane defining the base of the terrain, illustrated by Equation 3.6.

$$\begin{pmatrix} c_x \\ c_y \\ 0 \end{pmatrix} + \lambda \begin{pmatrix} d_x \\ d_y \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \\ 0 \end{pmatrix} \tag{3.6}$$

Rearranging for λ gives the equation

$$\lambda = \frac{-c_z}{d_z}.$$

This value can be substituted into Equation 3.6 to solve for the ray intersection.

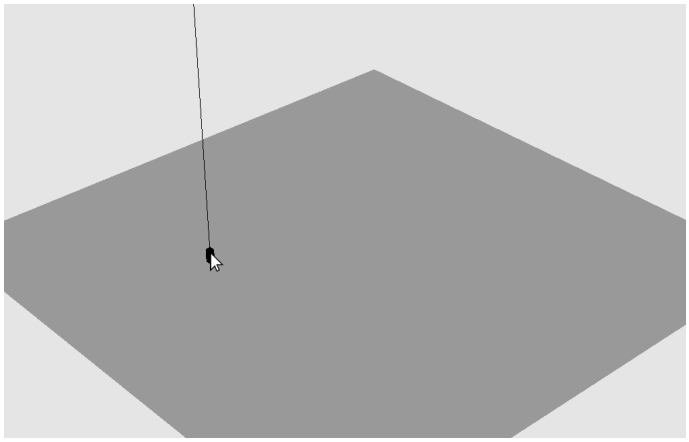


Figure 3.12: User placing a control point.

The implementation of this equation can be verified by rendering a point and vertical line at the intersection position, shown in Figure 3.12. This marker acts visual aid for the user and was included in the final implementation.

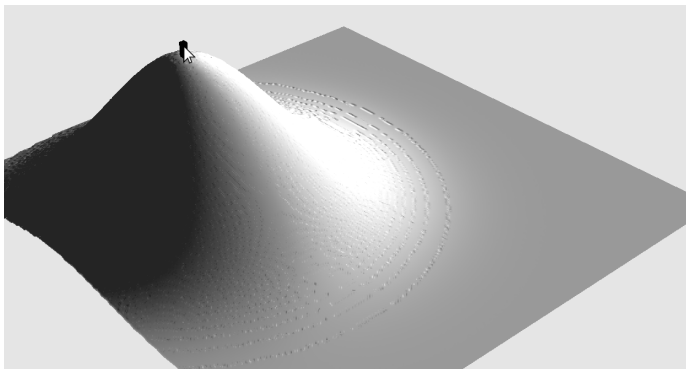


Figure 3.13: User adjusting a control point height.

Pressing the left mouse button creates a new control point, and clicking an existing control point allows the user to drag it to a new height (see Figure 3.13). Hovering the cursor over a control point and pressing the backspace key removes a control point. This simple editing environment acts as an intuitive and responsive terrain modelling interface.

3.4 Detailing Terrains with Generated Noise

In order to add detail to the terrain, produced by control point interpolation, the class **NoiseGenerator** was implemented to generate fractal noise as described in chapter 2.3. The value noise, layered to generate the fractal noise, was generated using linear interpolation.

Rather than implementing bi-cubic interpolation, which produces smooth value noise, bi-linear interpolation was implemented. This is due to the fact that bi-linear interpolation produces noise which, when rendered as a texture, appears coarser and therefore more reminiscent of the rocky terrain. It is important to provide the hydraulic erosion simulation with terrain that does not appear already eroded, allowing the simulation to selectively smooth areas of the terrain through erosion, whilst still leaving coarse mountain ranges.

3.4.1 Fractal Noise Algorithm

An array of white noise, named seed, is populated with random values in the interval [0:1]. The seed is used to generate several octaves of value noise, generated via linear interpolation. Each increasing octave is split into a greater amount of "chunks" to interpolate between and therefore represents the seed noise map at an increasingly high resolution. The seed noise map, value noise maps, and fractal noise map can all be considered square matrices S, V_o (where $1 \leq o \leq o_{max}$), and F. Each matrix is made up of L rows and columns. First, the seed

matrix S is populated with random numbers such that:

$$S = \begin{bmatrix} s_{1,1} & s_{2,1} & \dots & s_{L,1} \\ s_{1,2} & s_{2,2} & \dots & s_{L,2} \\ \vdots & \vdots & \ddots & \vdots \\ s_{1,L} & s_{2,L} & \dots & s_{L,L} \end{bmatrix} \text{ where } s_{x,y} \in \{x | x \in R, 0 \leq x \leq 1\}.$$

At each octave V_o, where o is the octave number, the chunk width per octave is w_o. Each octave is twice as wide in chunks as the previous octave, making

$$w_o = \lfloor 2^o \rfloor.$$

At each octave, the width in chunks w_o multiplied by the chunk size s_o should equal the map width L, therefore

$$s_o = \frac{L}{w_o}.$$

It should not be possible to have a chunk size of s_o less than 0, which means the maximum number of octaves o_{max} is

$$o_{max} = log_2(L).$$

Functions p_s and p_e both take an octave number o and matrix index x. p_s returns the start index whereas p_e returns the end index.

$$p_s(o,x) = \left\lfloor \frac{xw_o}{L} \right\rfloor c_{o,s},$$

$$p_e(o,x) = \left\lceil \frac{xw_o}{L} \right\rceil c_{o,s}$$

The function f takes an octave number o and matrix indices x and y. It is used to populate each octave with bi-linearly interpolated values. The function bilerp (see Equation **??**) returns an estimate for $S_{x,y}$ using bi-linear interpolation.

$$f(o,x,y) = \text{bilerp}(x, y, p_{start}(o,x), p_{end}(o,x), p_{start}(o,y), p_{end}(o,y))$$

Each value noise matrix V_o, at octave o where $1 \leq o \leq o_{max}$, is generated by linearly interpolating over the seed.

$$V_o = \begin{bmatrix} f(o,1,1) & f(o,2,1) & \dots & f(o,L,1) \\ f(o,1,2) & f(o,2,2) & \dots & f(o,L,2) \\ \vdots & \vdots & \ddots & \vdots \\ f(o,1,L) & f(o,2,L) & \dots & f(o,L,L) \end{bmatrix}$$

The function l layers the octave matrices at the index x,y with diminishing weights starting at $\frac{1}{2}$. Each layer is given half the weight of the previous layer, making the weight at each octave o equal to $\frac{1}{w_o}$.

$$l(x,y) = \sum_{o=1}^{o_{max}} \frac{1}{w_o} V_{o,x,y}$$

The fractal noise matrix F is constructed by layering each octave at each value and is therefore

$$F = \begin{bmatrix} l(1,1) & l(2,1) & \dots & l(L,1) \\ l(1,2) & l(2,2) & \dots & l(L,2) \\ \vdots & \vdots & \ddots & \vdots \\ l(1,L) & l(2,L) & \dots & l(L,L) \end{bmatrix}.$$

Figure 3.14: 4th octave of value noise.

Figure 3.15: 8th octave of value noise.

Lower octaves of value noise (Figure 3.14) have the greatest effect on each pixel's value and determine the larger peaks and troughs of the fractal noise. Higher octaves observed in Figure 3.15 appear coarse and have little effect on a pixel's value, contributing to the slight roughness observed in Figure 3.16.

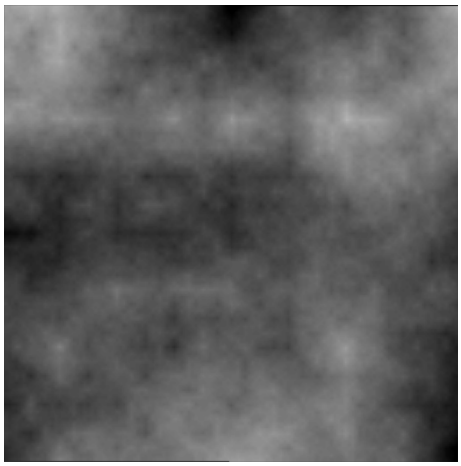


Figure 3.16: Fractal noise, represented as a greyscale image.

Figure 3.16 shows the resultant fractal noise generated by layering several octaves of value noise.

Figure 3.17: Fractal-noise-generated heightmap represented as a mesh.

Figure 3.17 shows a terrain representation verifying the C++ fractal noise generation implementation. The terrain exhibits distinct mountains and valleys (due to the lower octaves of noise) whilst still appearing rough and non-uniform at its surface (due to the higher octaves of noise).

3.4.2 Parameters

The implementation defines several parameters to determine the overall character of the fractal noise used to populate the array:

- *extrude* - a height multiplier, determining the relative height of peaks.
- *resolution* - the resolution level, relative to the number of octaves layered together.
- *skip* - an integer to suppress a number of initial octaves (those which have the greatest effect on terrain height).

When *skip = 0*, no octaves are skipped and the resultant terrain can be unpredictable in its shape, a *skip* value of 3 was found to be optimal for respecting the shape of the user's interpolated terrain.

3.5 Simulating Hydraulic Erosion

The implementation defines a Raindrop class to simulate hydraulic erosion. The virtual raindrop is repeatedly placed at a random location on the heightmap and, by determining the incline at its position, travels down the heightmap as if affected by gravity. By enabling the raindrop to collect and deposit small amounts of the terrain, the real-world process of hydraulic erosion can be simulated.

Aside from a number of physics attributes, the Raindrop class contains several attributes which vary throughout the course of the raindrop's lifetime. Most notably:

- *water* - the amount of water held by the raindrop.
- *sediment* - the amount of sediment held by the raindrop.
- *capacity* - the amount of sediment a raindrop can hold.

3.5.1 Assumptions

The algorithm is based on a set of assumptions about how rainfall interacts with sediment, these are as follows:

- Raindrops of a greater speed can carry more sediment.
- Raindrops of a greater amount of water can carry more sediment.
- Raindrops are of a constant mass.
- Raindrops evaporate at a constant rate.
- Raindrops cannot add or remove sediment to or from the overall system.
- The acceleration vector due to gravity is (0, 0, -1).

3.5.2 Algorithm Overview

Each iteration of the algorithm can be thought of in three stages, movement, erosion and evaporation. In the first stage, movement, the terrain's normal vector and the gravity vector are used to determine the acceleration vector. This acceleration vector influences the velocity, which then influences the position. In the second stage, erosion, the sediment capacity is determined relative to the water quantity and the magnitude of the velocity. If a raindrop is carrying more sediment than its capacity, it deposits some fraction of its sediment. Otherwise, it erodes some sediment from the terrain. In the final stage, evaporation, the water held by the raindrop is removed relative to time. The following three sections formalise these stages as vector mathematics.

Movement Equations

Two forces act on the raindrop at all times. These forces are gravity and friction. In order to calculate both, it is necessary to determine the normal vector at the terrain where the raindrop lies. Suppose the existence of a function, N, which takes some position and returns the normal vector at that position on the terrain. The terrain map is represented by a matrix T, for which each entry is a different height.

$$p = \begin{pmatrix} x \\ y \end{pmatrix} \tag{3.7}$$

The position p (see Equation 3.7) of the raindrop lies at x,y on the terrain T. The elevation of the raindrop can therefore be written as $T_{x,y}$.

$$\mathrm{N}(p) = \frac{1}{4} \begin{pmatrix} 2(T_{x+1,y} + T_{x-1,y}) \\ 2(T_{x,y+1} + T_{x,y-1}) \\ 4 \end{pmatrix} \tag{3.8}$$

The C++ implementation of the normal function (see Equation 3.8) can be verified, visually, by creating a texture. Using the calculated normal vectors to set the RGB values of each pixel, each component of each normal can be visualised in colour.

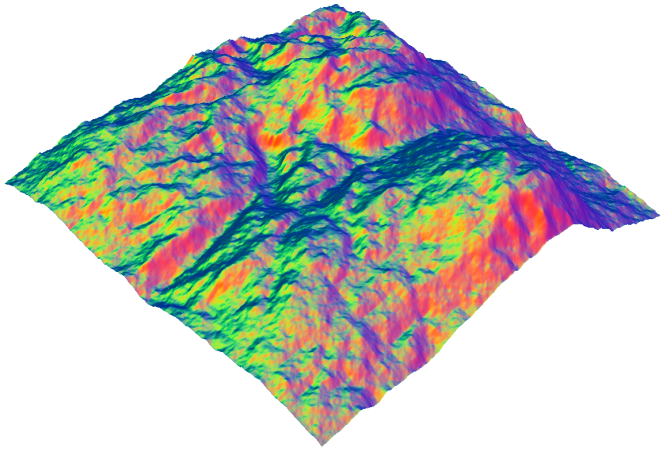


Figure 3.18: Mesh textured with the image generated by the normal function.

Figure 3.18 illustrates that the colours red, green, and blue correlate to the x, y, and z values of the appropriate normal vectors.

$$u = (n \times \hat{g} \times n) \tag{3.9}$$

The normal vector allows the vector down the slope, perpendicular to the normal, to be determined. This has been labelled u and is a unit vector.

$$\begin{aligned} a_g &= ug \sin\theta \\ a_g &= ug \frac{|-n \times \hat{g}|}{|n||\hat{g}|} \\ a_g &= ug(|-n \times \hat{g}|) \end{aligned} \tag{3.10}$$

The normal vector also allows the acceleration due to gravity, labelled a_g, to be determined. θ is the angle between the slope and the horizontal plane, which is also the angle between the normal vector and the gravity vector. Cross-product substitution for sine can be performed and both denominators $|n|$ and $|\hat{g}|$ equal are magnitudes of unit vectors (and therefore equal 1) so they can be simplified.

$$\begin{aligned} F_n &= mg\cos\theta \\ \cos\theta &= \frac{-n\cdot g}{|n||g|} \\ F_n &= m|g|\frac{-n\cdot g}{|n||g|} \\ F_n &= m\frac{-n\cdot g}{|n|} \\ F_n &= m(-n\cdot g) \end{aligned} \tag{3.11}$$

In order to calculate the friction force, it is necessary to calculate the magnitude of the normal force acting on the raindrop. As shown by Equation 3.11, this is calculated by taking the normal force equation and substituting cosine for the dot product of the normal vector and gravity vector. Both denominators can be simplified to give the magnitude of the normal force.

$$\begin{aligned} f_f &= -\mu F_n \hat{v} \\ a_f &= \frac{f_f}{m} \end{aligned} \tag{3.12}$$

The friction force, labelled f_f in Equation 3.12, acts in the opposite direction to the velocity, v, with a magnitude relative to μ, the friction coefficient. Given $f = \frac{m}{a}$, it is possible to determine a_f, the acceleration due to friction.

$$a = a_g + a_f \tag{3.13}$$

Summing the acceleration due to gravity and acceleration due to friction gives a (see Equation 3.13), the total acceleration of the raindrop. This is used to update the velocity, v, and, in turn, the position, p, of the raindrop.

$$\begin{aligned} v &= \frac{\Delta d}{\Delta t}, \Delta d = 1 \\ \Delta t &= \frac{1}{|v|} \end{aligned} \tag{3.14}$$

The raindrop travels a fixed distance of 1 at each iteration, regardless of velocity, to prevent any traversed sections of terrain from being skipped. This makes the theoretical timestep between iterations, Δt, equal to the inverse of the velocity (see Equation 3.14).

$$v_{t+1} = v_t + a\Delta t \tag{3.15}$$
$$p_{t+1} = p_t + \hat{v}_{t+1}$$

The timestep Δt is used to determine the new velocity and to update the position (by a step of length 1) accordingly (see Equation 3.15). The implementation of these physics equations can be verified by tracing the paths taken by the raindrop in the 3D viewing environment, as shown in Figure 3.19.

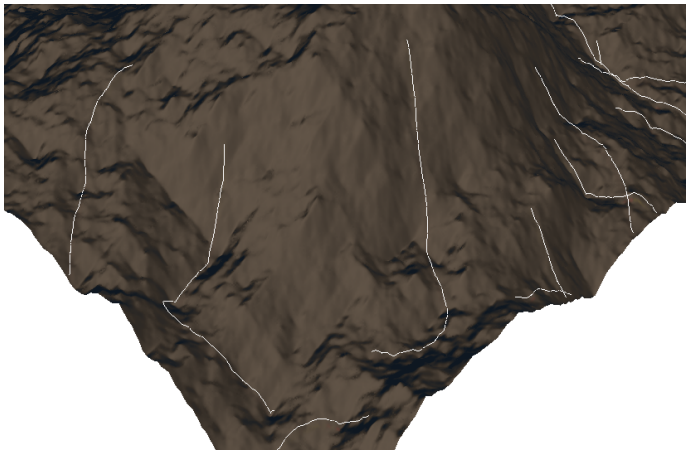


Figure 3.19: Raindrop paths traced by white lines.

As expected, the paths follow the contours of the terrain, travelling down slopes and into channels.

Erosion and Deposition Equations

$$c = (|v|)wc_f \tag{3.16}$$

Equation 3.16 illustrates that the amount of sediment the raindrop can hold at any time, that is the capacity (c), is the product of the speed ($|v|$), water (w), and capacity factor (c_f).

$$\mathrm{d}(c,s) = \begin{cases} (s-c)d_f, & \text{if } s > c \\ (s-c)e_f, & \text{otherwise} \end{cases} \tag{3.17}$$

The deposit amount is provided by the deposit function d (see Equation 3.17), which takes two inputs, capacity (c) and sediment (s). If the raindrop holds more sediment than

it can carry, a portion of its excess sediment (relative to the deposit factor d_f) is deposited. Otherwise, a portion of the remaining capacity (relative to the deposit factor e_f) is eroded - erosion is treated as negative deposition.

$$s_{t+1} = s_t - d(c_t, s_t) \tag{3.18}$$

From this, the update equation (see Equation 3.18) can be formed to update the raindrop's sediment level at each iteration.

$$T_{t+1,x,y} = T_{t,x,y} + d(c_t, s_t) \text{ where } p_{t+1} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{3.19}$$

Similarly, terrain matrix T can be updated, as shown by Equation 3.19. The implementation of these equations can be verified by texturing the terrain based on the amount of erosion and deposition performed.

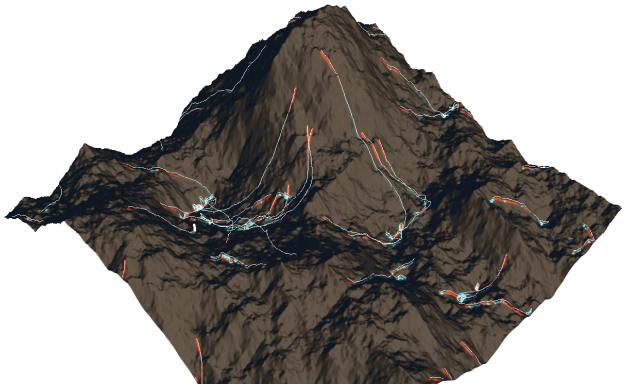


Figure 3.20: Mesh textured based on levels of erosion (red) and deposition (blue).

Figure 3.20 shows, as desired, that the raindrops erode terrain as they travel down the peaks, and deposit sediment once they slow down and evaporate in the troughs.

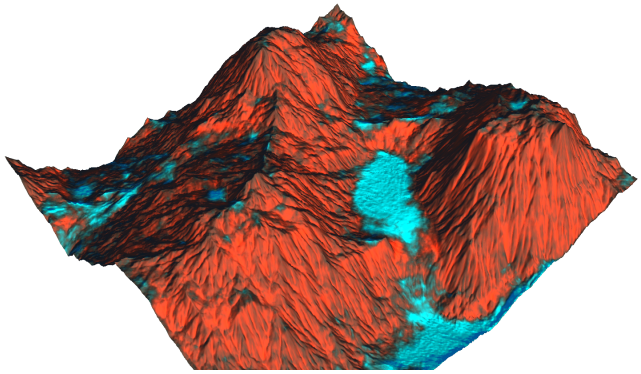


Figure 3.21: Mesh textured based on levels of erosion (red) and deposition (blue).

With enough iterations, the raindrop performs two tasks. First, by following minor contours down the terrain, it erodes and therefore deepens them into distinct channels, giving rise to ridge-like formations common to mountainous terrain. Second, by gravitating towards troughs, the raindrop tends to fill in any uneven terrain in the troughs, giving rise to flat, valley-like formations (also common to mountainous terrain). This can be observed in Figure 3.21. Erosion areas (red) exhibit rough, ridge-like formations, whereas deposition areas (blue) appear smooth.

Evaporation Equation

The raindrop evaporation is assumed to be constant relative to time, for simplicity. The update equation for the water (see Equation 3.20), where v_f is the evaporation factor, is as follows.

$$w_{t+1} = w_t - v - f\Delta t \tag{3.20}$$

This, in turn, reduces the capacity of the raindrop over time, encouraging it to deposit sediment over some distance, rather than depositing it all in one location. Without this gradual evaporation, the raindrop can lose all of its capacity at once, creating unnatural heaps of sediment.

3.5.3 Experiments

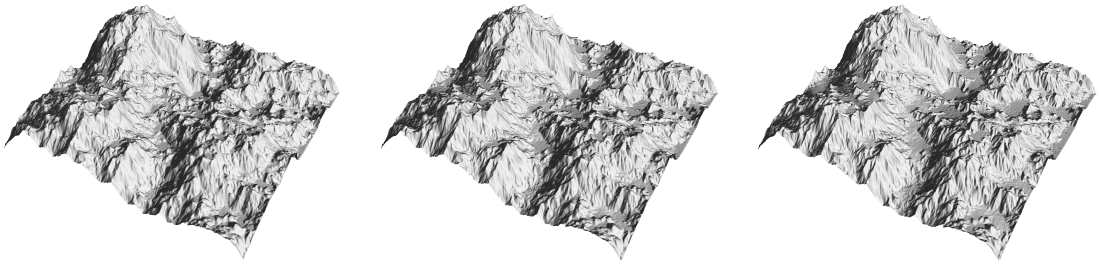
The Raindrop class contains a number of attributes determining the behaviour of the simulation and character of eroded terrain, including:

- *maxVelocity* - the maximum velocity of the raindrop.
- *friction* - the friction coefficient between the raindrop and the terrain.
- *water* - the amount of water held by the raindrop.
- *evaporateSpeed* - the rate at which a drop evaporates.
- *sediment* - the amount of sediment held by the raindrop.
- *depositSpeed* - the rate at which the raindrop deposits sediment.
- *erodeSpeed* - the rate at which the raindrop collects sediment.

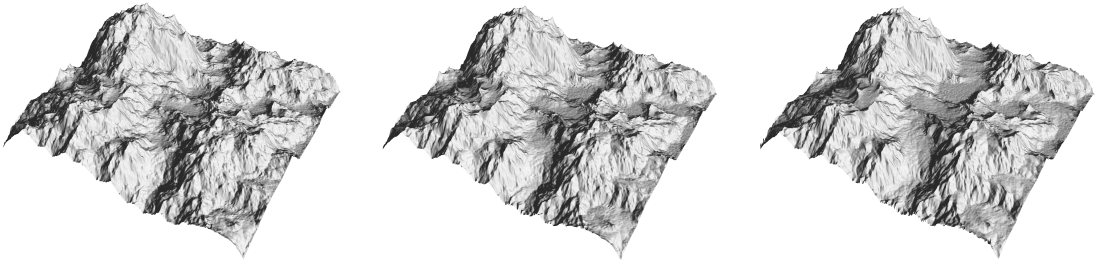
Experiments showed that modifying the maximum drop velocity had the greatest effect on the character of the terrain produced. A series of experiments were conducted to analyse the effects of modifying the maximum drop velocity.

Modifying Maximum Drop Velocity

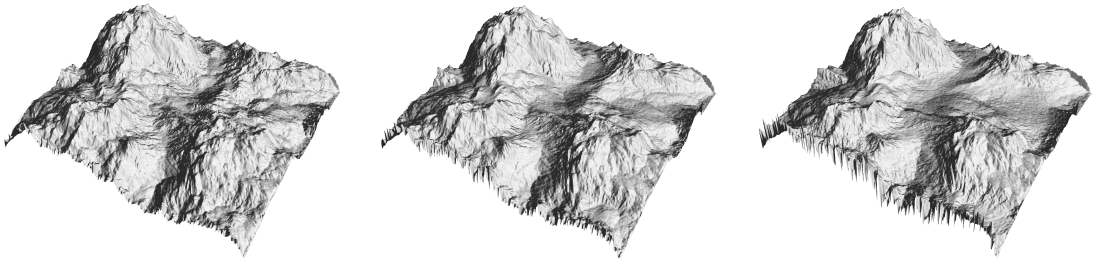
Experiments have been conducted for three maximum drop velocities; 1, 5, and 10. Results for each were recorded, from the beginning of each erosion simulation, at time t = 5s, 10s, and 20s.



(a) max. vel. = 1, t = 5 (b) max. vel. = 1, t = 10 (c) max. vel. = 1, t = 20



(d) max. vel. = 5, t = 5 (e) max. vel. = 5, t = 10 (f) max. vel. = 5, t = 20



(g) max. vel. = 10, t = 5 (h) max. vel. = 10, t = 10 (i) max. vel. = 10, t = 20

Figure 3.22: Maximum drop velocity experiment results.

Figure 3.22 shows that a lower maximum velocity produces flatter valleys and smoother mountains, whereas a higher maximum velocity instead produces rounded valleys and coarser mountains. User feedback found a maximum velocity of 5 was found to produce the most realistic results in this experiment, for this reason, it was set as the default maximum drop velocity.

3.6 Generating Rivers

During the hydraulic erosion simulation, the popularity of each heightmap entry is recorded. At each iteration, the raindrop contributes some small value to a heightmap, referred to here as the river heightmap, with the intention of recording overall raindrop flow. Rendering this river heightmap as a texture provides a visualisation of the data (see Figure 3.23).

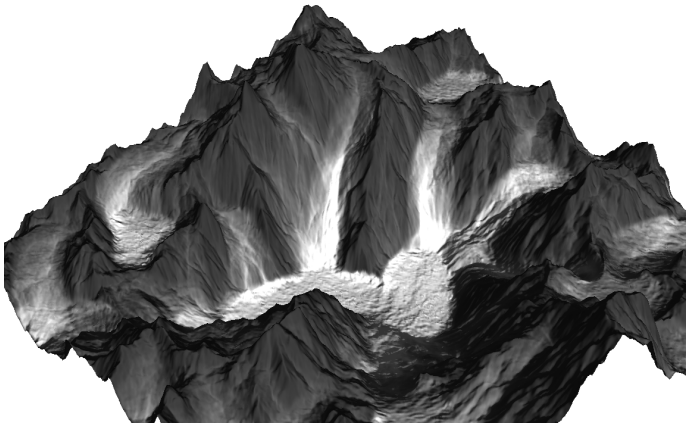


Figure 3.23: Terrain mesh textured using the river heightmap.

The result gives a general picture of raindrop flow, however, it does not appear similar to the routes rivers, brooks, and tributaries would be expected to take. Instead, the drop exhibits a flow similar to that of fast-moving waterfalls. This is due to the fact that the raindrop, configured for erosion/deposition, has a high velocity, and high friction; this produces desirable results during the erosion simulation, however increasing the realism of the river map requires re-configuring the raindrop.

3.6.1 Improving River Quality

After each erosion pass, the raindrop is re-configured for a brief river pass. By lowering the max velocity, the raindrop travels slower and thus follows the terrain shape more closely, and by lowering the friction, the raindrop is able to travel long distances on relatively flat terrain.

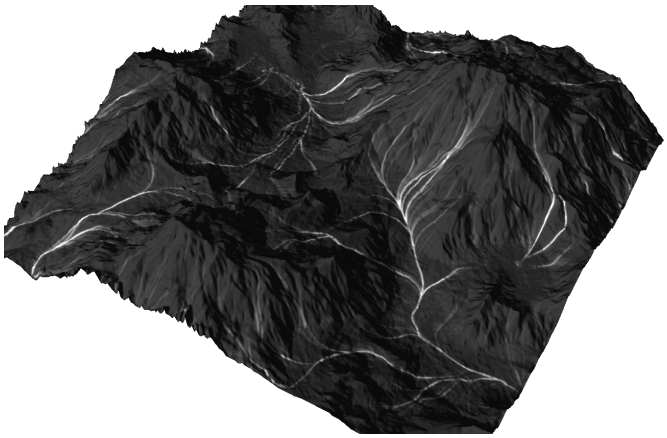


Figure 3.24: Terrain mesh textured using the re-configured river pass data.

Figure 3.24 demonstrates the result of reducing the raindrop's maximum velocity and friction coefficient. By reducing the number of iterations performed during the river pass, a more selective image of rivers can be generated.

3.7 Generating Lakes

A similar approach to that of river identification has been taken to lake identification. At the end of the raindrop's lifetime, some small value is contributed to a heightmap known as the lake heightmap. The assumption is that by recording raindrop destinations, river destinations, and therefore lake locations, can be identified.

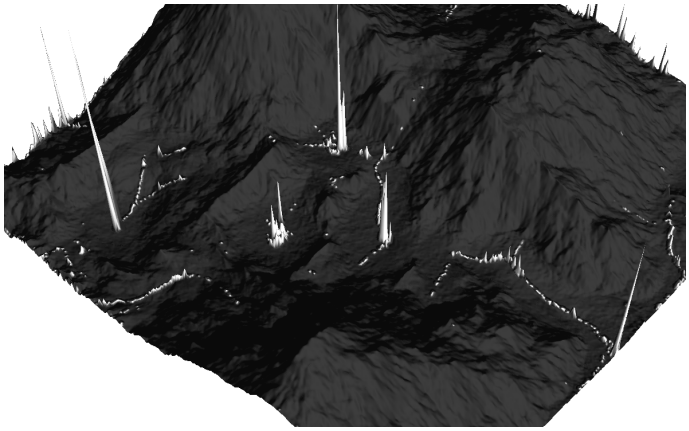


Figure 3.25: Terrain mesh textured using the lake heightmap.

Figure 3.25 illustrates the location and height of lake heightmap entries on the terrain model. Without any physics, the lakes form as tall piles on the terrain. As such, a smoothing algorithm has been devised to collapse the lake data into flat pools.

3.7.1 The Lake Smoothing Algorithm

The smoothing algorithm considers the lake heightmap as a grid of interconnected pools. The flux between each pool is calculated as the difference in water level, thus causing higher water levels to flow into lower neighbouring pools. The algorithm is iterative and only finishes once the difference between all neighbouring water levels is below a certain threshold.

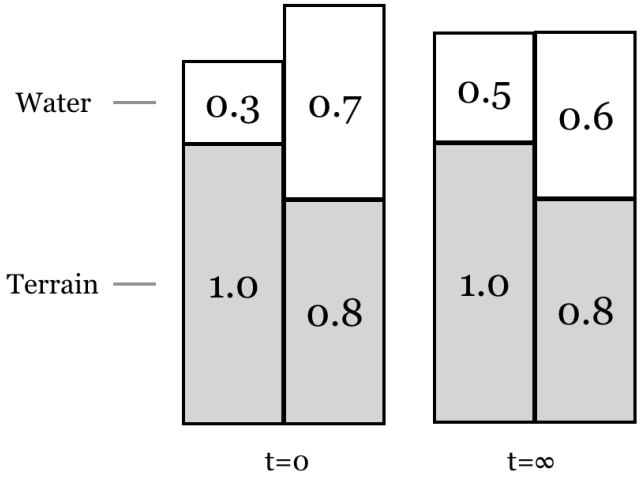


Figure 3.26: Water flow between neighbouring water levels.

Figure 3.26 illustrates two neighbouring entries of the lake heightmap and terrain heightmap equalising in water level.

$$\textbf{flux}(a,b) = \begin{cases} \frac{a-b}{4}, & \text{if } a \geq b \\ 0, & \text{otherwise} \end{cases} \tag{3.21}$$

Equation 3.21 shows the flux equation allowing a water level a to flow into b if b is lower. The difference is quartered as the flux is taken four times for entries north, south, east, and west of entry (x,y).

$$W_{x,y} = T_{x,y} + L_{x,y} \tag{3.22}$$

Equation 3.22 defines the water level W at entry (x,y) as the sum of the terrain heightmap T entry and lake heightmap L entries at (x,y).

$$\begin{aligned} f_n &= \frac{1}{2}\textbf{flux}(W_{x,y}, W_{x,y+1}) \\ f_s &= \frac{1}{2}\textbf{flux}(W_{x,y}, W_{x,y-1}) \\ f_w &= \frac{1}{2}\textbf{flux}(W_{x,y}, W_{x+1,y}) \\ f_e &= \frac{1}{2}\textbf{flux}(W_{x,y}, W_{x-1,y}) \end{aligned} \tag{3.23}$$

Equation 3.23 defines values for lake heightmap entries north, south, east, and west of the entry at (x, y). At each iteration, only half of the total flux is taken, allowing the algorithm to tend towards a stable solution.

$$\begin{aligned} L_{x,y} &\leftarrow L_{x,y} - f_w - f_e - f_n - f_s \\ L_{x,y+1} &\leftarrow L_{x,y+1} + f_n \\ L_{x,y-1} &\leftarrow L_{x,y-1} + f_s \\ L_{x+1,y} &\leftarrow L_{x+1,y} + f_e \\ L_{x-1,y} &\leftarrow L_{x-1,y} + f_w \end{aligned} \tag{3.24}$$

Equation 3.24 updates the lake heightmap entry at (x, y) and all four of its neighbouring entries. These update equations are applied to each entry of the lake heightmap until the difference between each neighbouring water level is below a small threshold $t = 0.0005$. The implementation of the algorithm can be verified by rendering visualising the heightmap as a texture.

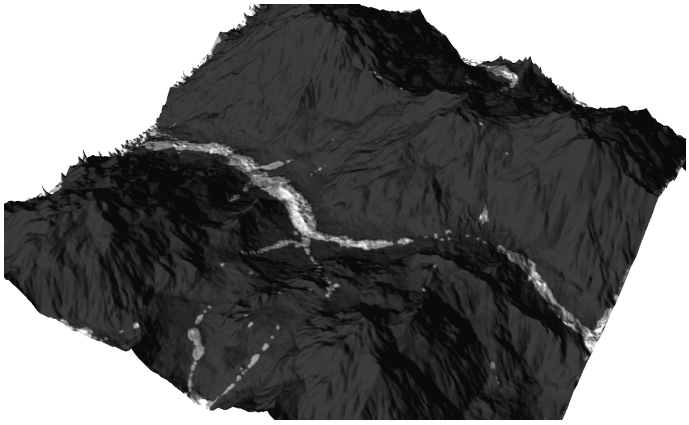


Figure 3.27: Terrain mesh textured using the smoothed lake heightmap.

By iteratively applying the algorithm to the lake heightmap, the water flows back and forth between neighbouring cells, eventually stabilising into flat pools. Figure 3.27 demonstrates the result of applying the smoothing algorithm to the lakes shown in Figure 3.25.

3.7.2 Generating Soil

The existence of soil can be simulated using Gaussian smoothing kernels. The implementation uses a Gaussian smoothing kernel to smooth a copy of the terrain heightmap. At each entry, the maximum value between the terrain heightmap and the smoothed terrain heightmap is taken and used when rendering. The effect is purely visual, as the erosion simulation continues on the underlying terrain heightmap, but greatly enhances the terrain's perceived realism.

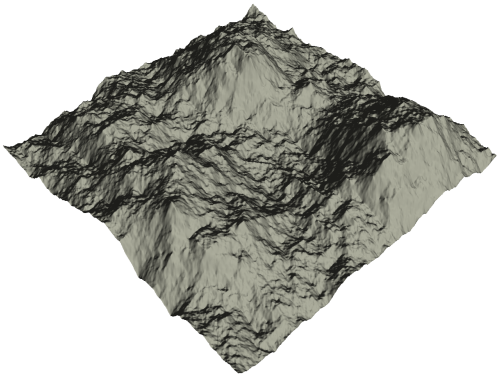


Figure 3.28: Before applying blurred terrain.

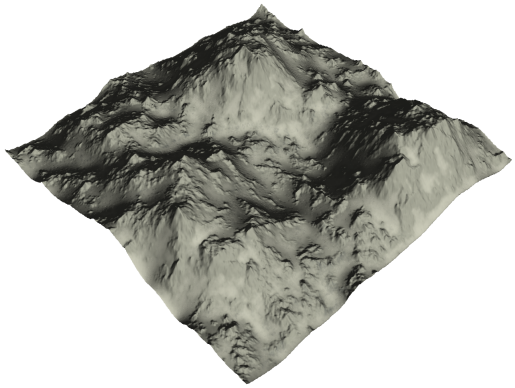


Figure 3.29: After applying blurred terrain.

Figure 3.35 and Figure 3.36 show the effect of applying this simple algorithm to a terrain, prior to any erosion simulation. Taking the maximum causes minor troughs to be filled in by the noisy terrain, and minor peaks to emerge from the approximated soil. This is a computationally efficient method of increasing the perceived realism, and it provides a basis for categorising rock and soil in the texturing stage.

3.8 Texturing

The texturing algorithm combines data from the terrain heightmap, the smoothed terrain heightmap, the lake heightmap, and the river heightmap to categorise the terrain into rock, soil, grass, and water. The incline at each area of the terrain is also a determining factor when categorising grass, this is based on the assumption that grass should be less likely to grow on steep inclines.

3.8.1 Classifying Terrain

The terrain heightmap and smoothed terrain heightmap are used to distinguish between rock and soil. Rock is identified by taking areas of the terrain which have a higher altitude than the smoothed terrain.

Figure 3.30: Textured terrain representing areas soil (light).

Figure 3.30 represents areas of rock as dark in colour and areas of soil as light. All areas of terrain which are not rock can be considered soil at this point.

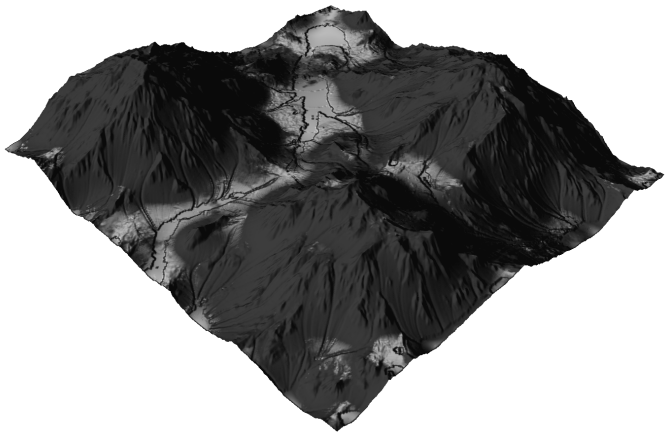


Figure 3.31: Textured terrain representing areas of grass (light).

Areas of soil with a low incline, that is a normal vector at a low angle to the vertical axis, are categorised as areas of grass. Figure 3.31 highlights areas identified as grass.

Figure 3.32: Textured terrain representing areas of water (light).

Areas of water are determined by taking the maximum value of the lake heightmap and river heightmap. Figure 3.32 shows a map of all the identified water of a terrain. Note that by slightly indenting the terrain at rivers and lakes before rendering, the impression of river banks and shorelines can be created, suggesting that the identified water features have some depth.

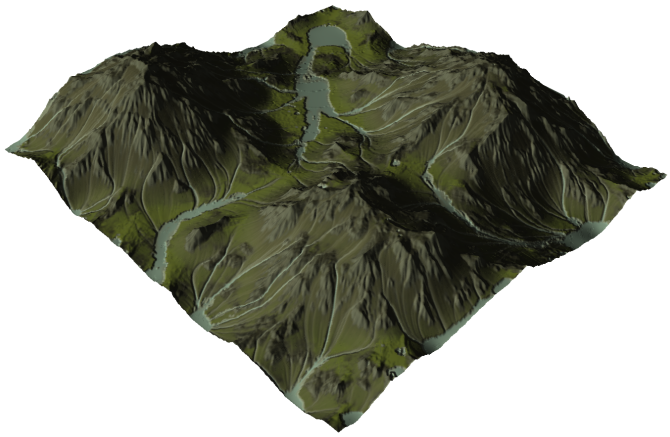


Figure 3.33: Textured terrain.

Finally, a simple algorithm blends colours of rock, water, soil, and grass as grey, blue, brown and green respectively. From this, a texture image is written at the same resolution as the heightmap. The texture image is continually updated during the erosion simulation, to match the changing terrain, and is saved as a PNG file when exporting the terrain model.

3.9 Mesh Optimisation

The aim of the mesh optimisation stage is to reduce the number of triangles used to construct the mesh, whilst still maintaining an accurate representation of the underlying heightmap. An algorithm has been devised to minimise the number of triangles used, given a resolution value. The algorithm tessellates different areas of the mesh by varying amounts based on their topography.

3.9.1 Patches and Tessellating in OpenGL

OpenGL's tessellation feature allows quadrilaterals to be generated with varying resolutions. By dividing the terrain model into a grid of patches, each patch can be rendered with a different resolution. The proposed algorithm renders each patch based on its complexity. Complex areas of the terrain with a higher resolution, and simple areas of the terrain with a lower resolution.

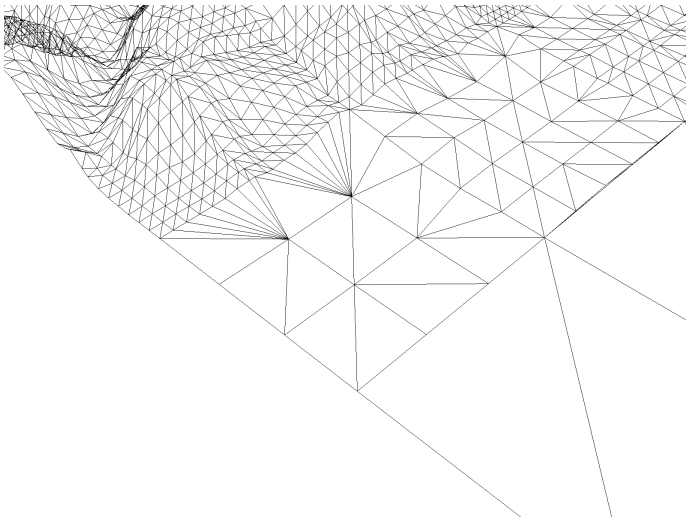


Figure 3.34: Wireframe view of tessellated patch edges.

Figure 3.34 shows a tessellated patch connected to neighbouring patches.

3.9.2 Detail-Based Tessellation Algorithm

The devised algorithm is performed over several iterations. At each iteration, all patch resolutions are increased. Once the average absolute error of a patch (relative to the original heightmap) is below a certain threshold, the patch resolution is locked.

Rendering Tessellated Heightmaps

In order to generate heightmaps from tessellated meshes, it is necessary to colour the mesh using the height of each fragment. Rendering this with an orthographic projection, a viewport scaled to the original heightmap dimensions, and a camera matrix facing vertically downwards results in a heightmap representation of the tessellated mesh. Rather than rendering to the screen, the image can instead be rendered to one of OpenGL's auxiliary buffers and read out to the main memory. This enables the comparison between the original heightmap and the tessellated heightmap.

Determining Patch Error

The patch error is determined by subtracting the tessellated heightmap from the original heightmap and taking the average absolute difference at each patch. This is used as a metric of the accuracy of each patch. Through experimentation, an appropriate per-patch average absolute error threshold was determined to be 0.001 for heightmap data in the range [0:1].

3.9.3 Tessellation Results

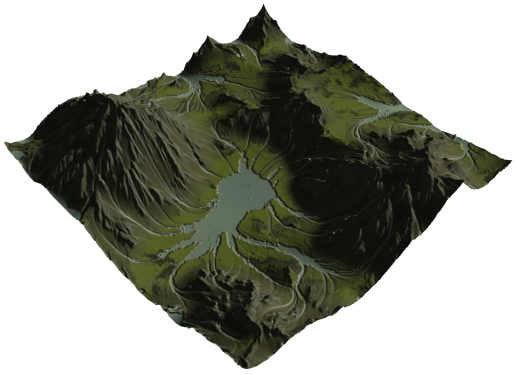


Figure 3.35: Tessellated terrain model.

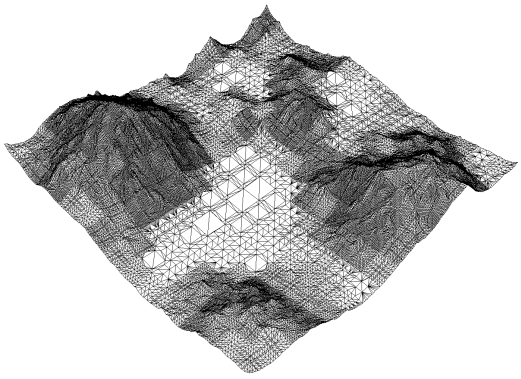
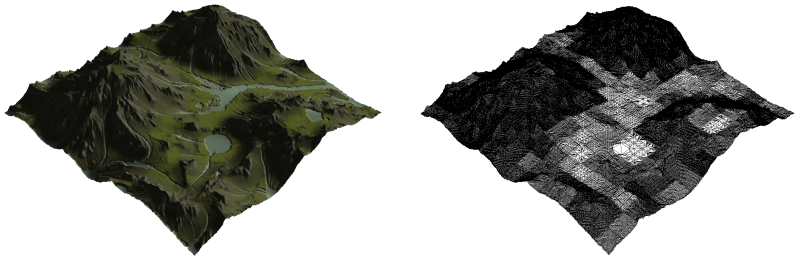


Figure 3.36: Tessellated terrain model (wireframe view).

Figure **??** and Figure **??** show a terrain model produced using the detail-based tessellation algorithm. The result is a terrain mesh with varying resolutions across its surface to match the terrain shape. The algorithm correctly simplifies the flat lake in the centre of the mesh, whilst maintaining a higher level of detail in the mountainous sections. The average patch error threshold can be modified, and the results analysed, to fine-tune the algorithm.

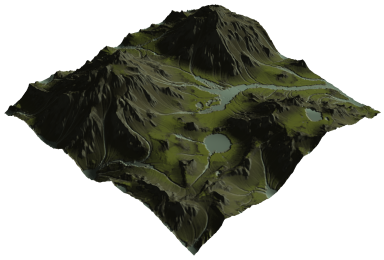
3.9.4 Error Threshold Experimentation

Four different average patch error thresholds were used to tessellate a sample terrain model. The number of triangles used to construct the mesh at each tessellation level was recorded and compared against the triangle count when rendering the model at a 1:1 resolution to the heightmap.

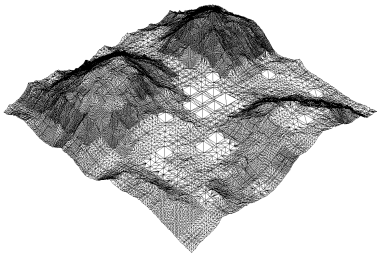


(a) Threshold = 0.0005 (regular)

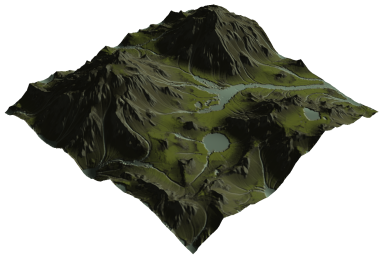
(b) Threshold = 0.0005 (wireframe)



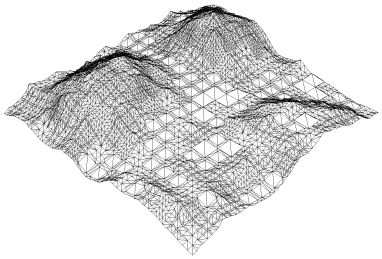
(c) Threshold = 0.001 (regular)



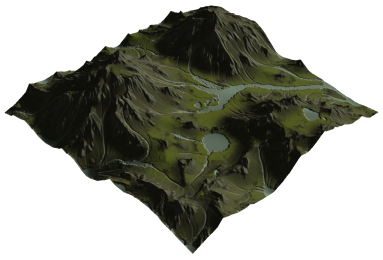
(d) Threshold = 0.001 (wireframe)



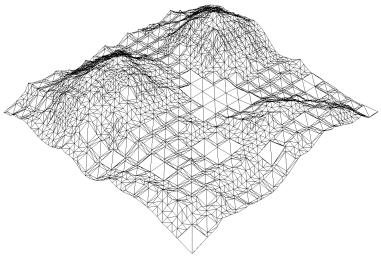
(e) Threshold = 0.0015 (regular)



(f) Threshold = 0.0015 (wireframe)



(g) Threshold = 0.002 (regular)



(h) Threshold = 0.002 (wireframe)

Figure 3.37: Visual mesh simplification results.

Figure 3.37 illustrates the effect of rendering at different average patch error thresholds. The number of triangles used to render each mesh is detailed in Table 3.1.

Analytical Mesh Simplification Results		
Avg. Patch Err. Threshold	Triangle Count	Relative Triangle Reduction
Control Mesh	522242	0.000%
0.0005	317108	39.279%
0.001	64560	87.638%
0.0015	12936	97.523%
0.002	6072	98.837%

Table 3.1: Result set for the triangle reduction experiment.

Table 3.1 shows that the mesh simplification algorithm is capable of reducing the number of triangles used by over 87% when using an average patch error threshold value of 0.001 or greater.

3.10 Exporting Terrain Models

To allow users to export their terrain models for use in other applications, the application allows the export of terrain models to 3D object files. Two object file formats have been chosen; OBJ and glTF.

The OBJ file format specification, created by Wavefront Technologies [Tec92], is a widely used format for storing 3D model data. It uses a simple ASCII text format to describe the geometry, materials, and textures of a 3D model and has become a popular format for interchange between 3D modelling software applications. Due to its support and readability, exporting to OBJ files was implemented. However, as the OBJ file format does not support textures without an MTL companion file, the OBJs exported are texture-less (in the interest of implementation simplicity). Textured model exporting is, however, supported in the implemented glTF model exporter, the results of which can be converted to almost any other 3d model file format using existing 3d model file conversion software.

According to the specification document for glTF 2.0 published by The Khronos Group Inc. [The17], the file format is designed for efficient 3D asset delivery and provides support for a wide range of features, including PBR materials, animations, and morph targets. Many 3d model file types exist, each with different features and application support, the glTF file format was created in the interest of standardising the transfer of 3d models and has been released as the ISO/IEC 12113:2022 International Standard. It is for this reason that full support for glTF models was provided.

3.10.1 Retrieving Tessellated Models

Due to the fact that model tessellation is performed inside the tessellation shader, it is necessary to pass the results from the rendering engine back into the program memory space. The implementation uses a transform feedback buffer to capture triangles generated by the

tessellation shaders. The array of triangle faces is iterated over to construct an array of vertices and indices. The normal vector at each triangle face is calculated, and the normal at each vertex is calculated as the average of the face normals connected to the vertex. The vertex coordinates are each between -0.5 and 0.5, and so can therefore be increased by 0.5 to generate appropriate texture coordinates to sample the heightmap between 0.0 and 1.0. Given the arrays of vertices, triangle face indices, vertex normals and texture coordinates, OBJ and glTF files can be generated.

3.10.2 Exporting to OBJ

OBJ files are written in plain text, with each vertex in the form "v x y z" and each face in the form "f v1 v2 v3" where v1, v2, and v3 are indices to the vertices listed above. Normal vectors and texture coordinates can also be stored with a similar syntax.

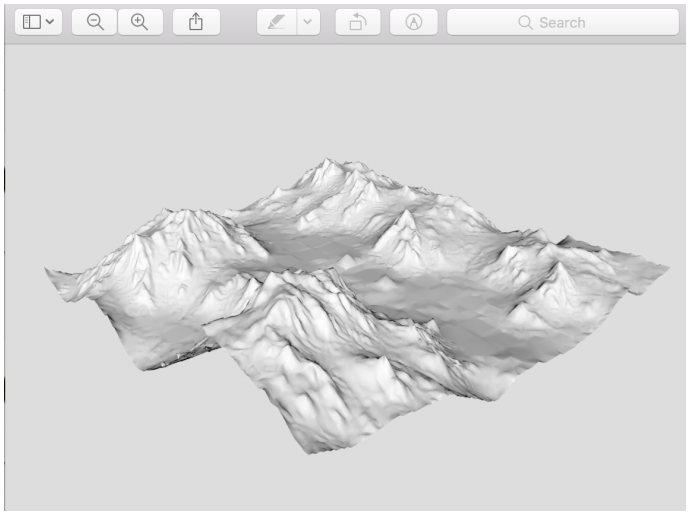


Figure 3.38: Exported terrain object viewed in MacOS's Preview.

Running the model exporter and opening the result in the MacOS file viewer Preview, as shown in Figure 3.38, confirms the functionality of the OBJ exporter.

3.10.3 Exporting to glTF

The glTF file contains a JSON object which points to the relevant data files in its directory. Vertices, indices, normals, and texture coordinates are written to a binary file which is pointed to by the JSON object. Similarly, the texture image is written to the directory, allowing the exported terrain model to be viewed in colour.

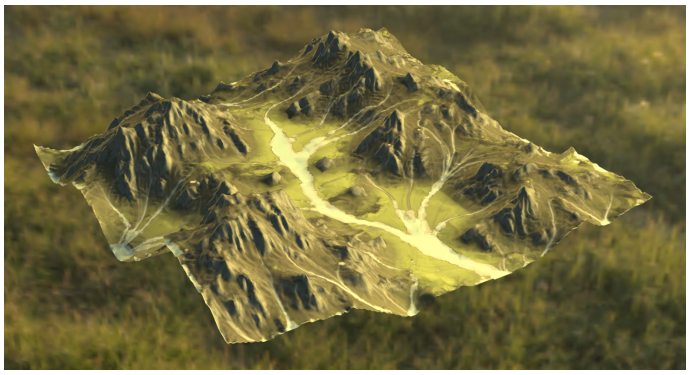


Figure 3.39: glTF model viewed in an online glTF viewer.

The implementation of the exporter can be verified by viewing the glTF model in an online glTF viewer, as shown in Figure 3.39.

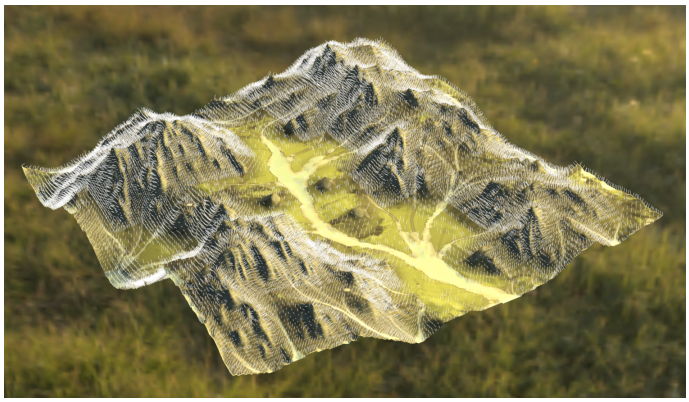


Figure 3.40: glTF model viewed with normal vectors in an online glTF viewer.

Similarly, online viewing tools allow the generated normal vectors to be verified too, as shown in Figure 3.40.

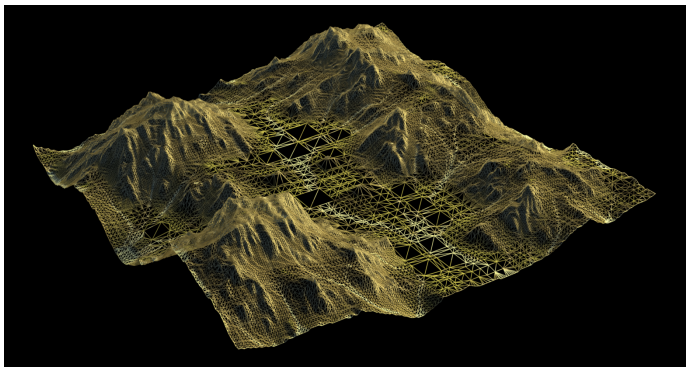


Figure 3.41: glTF model viewed, in a wireframe view, in an online glTF viewer.

Finally, the mesh quality of exported tessellated models can also be verified, as shown in Figure 3.41.

The model exporting implementation successfully retrieves tessellated meshes from the rendering engine and allows users to interact with their textured terrain models outside of the application.

Chapter 4

Evaluation

The evaluation of the implementation is based on the visual quality of generated terrain, the level of optimisation achieved during mesh simplification, and the performance and usability of the application.

4.0.1 Visual Quality

The visual quality of the terrain models is a naturally subjective measure, however, there exists a range of geographical features common to real-world terrain. The presence of such features in the generated terrain model can be used to evaluate the success of the application. The analysis will be performed on a sample terrain generated using the application.

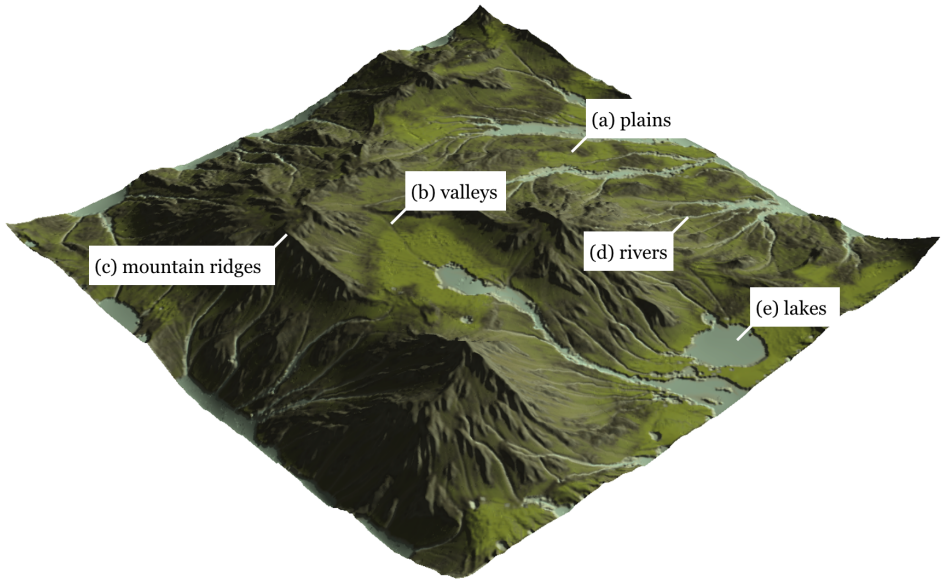


Figure 4.1: Application-generated terrain labelled by geographical feature.

Figure 4.1 illustrates the range of geographical features capable of being produced by the application including plains, valleys, ridges, rivers, and lakes. However, the nature of the implemented hydraulic erosion simulation (applied drop-by-drop) gives rise to features only created by rainfall. This prevents certain features of hydraulic erosion, such as the meandering of rivers, from appearing. The realism of the model could be increased by applying advanced erosion algorithms to generate additional features such as cliffs, rockfalls, and landslides would require an additional set of algorithms. The use of a heightmap representation limits the generated terrain from exhibiting any overhanging cliffs or arches, which could be generated by simulating coastal erosion [PGGM09].

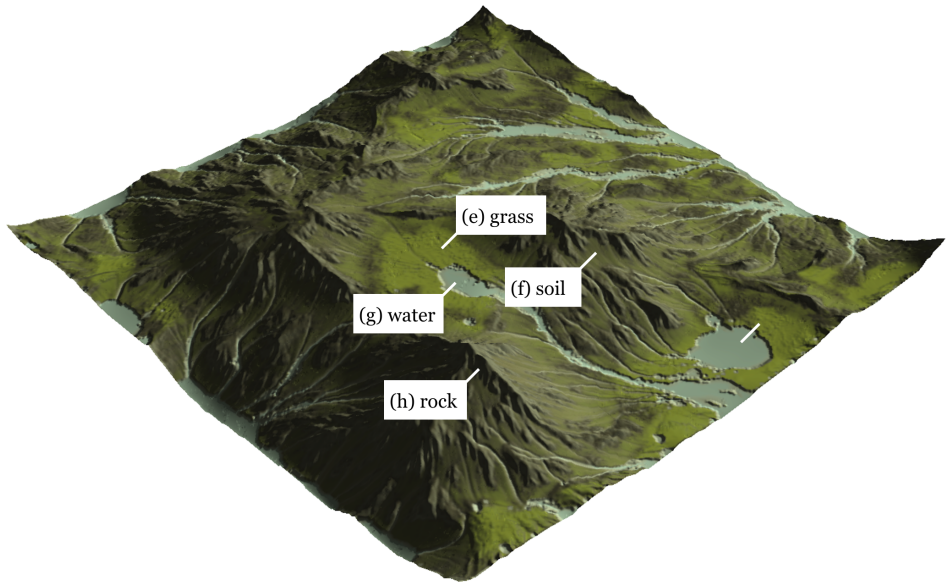


Figure 4.2: Application-generated terrain labelled by terrain type.

Figure 4.2 illustrates the range of terrain types generated by the texturing algorithm including grass, soil, water, and rock. One subtle visual artefact of the generated terrain is the presence of axis-aligned ridges and valleys, due to the use of fractal value noise. The generation of higher-quality noise, such as Perlin noise or Simplex noise [Gus05], could resolve this issue. Additionally, the effectiveness of the texturing algorithm is limited by the use of blended colours rather than blended images. Additionally, the realism of the terrain model is limited by the absence of vegetation. Although the impression of vegetation is provided by the texture representation of grass, the simulation of individual trees across the terrain could greatly increase the sense of realism.

4.0.2 Mesh Optimisation

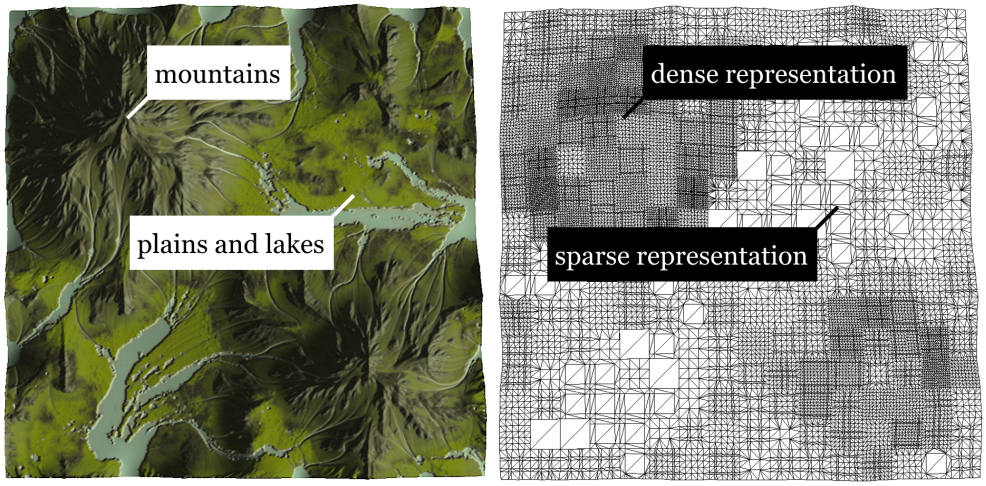


Figure 4.3: Application-generated terrain tessellated by complexity.

As shown by Figure 4.3, the implemented mesh optimisation stage is able to adjust the resolution across terrain meshes based on their complexity. It is capable of reducing the triangle count of terrain meshes by an average of 86% at an average patch error threshold of 0.001, which is to say that the average error of each patch in the 20x20 patch grid is less than 0.001 for heightmaps of width, length, and height 1.0.

However, the method of separating the terrain into a grid of tessellated patches places a limit on the level of optimisation that can be achieved. This is due to the fact that the contours between simple and complex terrain don't necessarily fall into a grid. Greater mesh optimisation and accuracy could be achieved by editing mesh vertices without following a grid pattern.

4.0.3 Efficiency and Performance

The application runs at 30 frames per second when modelling and viewing the terrain. However, during the hydraulic erosion simulation, which is the most computationally demanding aspect of the application, the frame rate is reduced to 10 frames per second in order to increase the speed of the simulation at the cost of user experience. This has been done under the assumption that the user will spend the duration of the erosion simulation waiting. Under these settings, the erosion simulation typically takes between 5 and 10 seconds on a 2.3 GHz Intel Core i5 processor. The performance of the application could be greatly increased by implementing a multi-threaded hydraulic erosion simulation, or by redesigning the simulation to run on the GPU [MDH07].

4.0.4 Usability

The application provides a simple user interface of buttons and sliders to navigate the menus and alter the application parameters, as seen in Figure 4.4.

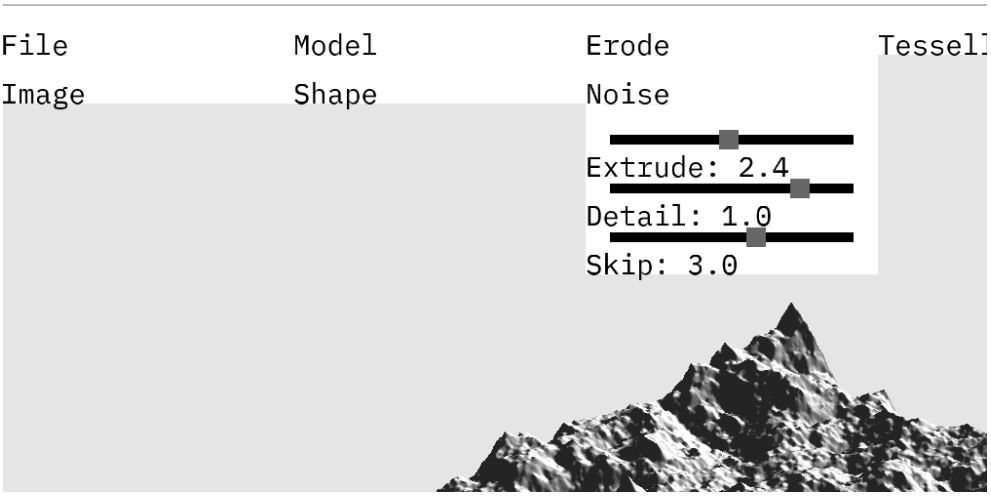


Figure 4.4: Dropdown buttons and sliders under the application toolbar.

Click and drag interaction with the 3D viewing environment has been implemented to allow users to drag peaks and troughs in the terrain. For advanced users, the application provides additional tools, such as a wireframe viewing mode, information on the mesh's triangle count, and the application command line. The application command line provides advanced functionality, allowing users to type commands to modify the erosion simulation and mesh simplification algorithms. However, the application lacks important accessibility features, such as navigation using the tab key. Similarly, some features of the application are only accessible via the keyboard, such as those accessed via the application command line. The usability of the application relies currently on the simplicity of the interface; the inclusion of additional features would warrant an in-application tutorial or application manual, which are missing from the implementation.

Chapter 5

Reflections

The techniques implemented in the application were successful in producing realistic and optimised 3D terrain models, allowing users to easily shape terrain models and enhance them with a range of geographical features. Additional aspects of model generation such as texturing, mesh simplification, and model exporting were researched and successfully implemented. However, due to the scope of the implementation, additional experiments regarding the usability, intractability, and customisation of the application remain.

Recent advancements in machine learning provide an interesting avenue for future research into terrain generation, where digital elevation models and satellite imagery can be used to train neural networks with the potential of generating complex geographies and textures without the need for computationally expensive erosion simulations or bespoke feature generation algorithms.

To conclude, this project has provided valuable insights into the development of terrain generation applications and has been an excellent opportunity to explore and understand the advantages and disadvantages of a range of terrain generation techniques. Covering interpolation, noise generation, and hydraulic erosion to texture generation, mesh simplification, and 3D model exporting, my hope is that this report is able to provide a thorough account of the entire terrain model generation process.

Bibliography

[BF02] Bedrich Benes and Rafael Forsbach. Forsbach r.: Visual simulation of hydraulic erosion. volume 10, pages 79–94, 01 2002.

[Buh03] Martin Buhmann. Radial basis functions: Theory and implementations. *Radial Basis Functions*, 12, 07 2003.

[DBH00] Jürgen Döllner, K. Baumann, and K. Hinrichs. Texturing techniques for terrain visualization. pages 227–234, 11 2000.

[GH97] Michael Garland and Paul Heckbert. Surface simplification using quadric error metrics. *Proceedings of the ACM SIGGRAPH Conference on Computer Graphics*, 1997, 07 1997.

[GJ+20] Gaël Guennebaud, Benoît Jacob, et al. Eigen. `http://eigen.tuxfamily.org`, 2010–2020. Version 3.4.0.

[Gus05] Stefan Gustavson. Simplex noise demystified. 01 2005.

[HB04] Michael Hodgson and Patrick Bresnahan. Accuracy of airborne lidar-derived elevation: Empirical assessment and error budget. *Photogrammetric Engineering & Remote Sensing*, 70:331–339, 03 2004.

[HMP17] Tuomo Hyttinen, Erkki Mäkinen, and Timo Poranen. Terrain synthesis using noise by examples. In *Proceedings of the 21st International Academic Mindtrek Conference*, AcademicMindtrek '17, page 17–25, New York, NY, USA, 2017. Association for Computing Machinery.

[MDH07] Xing Mei, Philippe Decaudin, and Bao-Gang Hu. Fast Hydraulic Erosion Simulation and Visualization on GPU. In Marc Alexa, Steven J. Gortler, and Tao Ju, editors, *PG '07 - 15th Pacific Conference on Computer Graphics and Applications*, Pacific Graphics 2007, pages 47–56, Maui, United States, October 2007. IEEE.

[MKM98] F. Musgrave, Craig Kolb, and R. Mace. The synthesis and rendering of eroded fractal terrains. *ACM SIGGRAPH Computer Graphics*, 23:41–50, 02 1998.

[Ope17] Opengl. https://www.opengl.org/registry/, July 2017. Version 4.6. Technical specification.

[PGGM09] Adrien Peytavie, Eric Galin, Jérôme Grosjean, and Stéphane Mérillou. Arches: a framework for modeling complex terrains. *Computer Graphics Forum*, 28:457 – 467, 04 2009.

[SW19] Ryan Spick and James Walker. Realistic and textured terrain generation using gans. pages 1–10, 12 2019.

[Tec92] Wavefront Technologies. Obj file format specification. https://www.loc.gov/preservation/digital/formats/fdd/fdd000507.shtml, 1992.

[The17] The Khronos Group Inc. glTF 2.0 Specification. https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html, 2017.