Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Håvard Borge

Framework for Rendering of Procedurally Generated Terrain

Master's thesis in Cybernetics and Robotics
Supervisor: Sverre Hendseth
June 2020

NTNU
Norwegian University of Science and Technology

Håvard Borge

Framework for Rendering of Procedurally Generated Terrain

Master’s thesis in Cybernetics and Robotics
Supervisor: Sverre Hendseth
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Problem Description

Procedural generation is the process of generating content with algorithms instead of manually. Procedural generation is being increasingly used by video game developers to decrease development time. This project will focus on the rendering of a procedurally generated world. A program that can be used to render a terrain with a set of models will be developed, and an end-user will be able to modify the properties of the terrain by changing the input to the program.

The student shall

- Do a background study to examine how different terrain models can be rendered.
- Develop a back end software for rendering procedurally generated terrain.
- Implement the program such that a text file can be used to list properties and features of the terrain.

Abstract

This paper studied how to implement a program for the rendering of a procedurally generated world. The essential parts of the project were to implement basic models and to have an input format to be used by an end-user.
Creating height variety was designed as a mountain implementation, and tests showed how the mountain model could be used to give the terrain plenty of variety. The project uses the midpoint displacement algorithm to implemented paths, and in the demo section, the paths make roads and rivers. The paper also explores how to implement textures and camera movement in a three-dimensional world.
Experiments conducted tested the limits of the system. The demo section shows how the implemented models can be used to generate more complex objects. And the possibilities section showcases how the models can be used to render different terrain representations. Lastly, the paper discusses how the project can be used as a foundation for future work and suggests options for developing the program further.

Sammendrag

Denne oppgaven undersøkte hvordan et program for a tegne en prosessuell generert verden kan implementeres. De viktigste delene av prosjektet var å implementere grunnlegende modeller og å ha et inndataformat som kan brukes av en sluttbruker.
Høydevariasjoner ble designed som en fjellimplementasjon, og tester ble gjennomført for å vise hvordan fjellmodellen kunne brukes for å gi terrenget rikelig med variasjon. Prosjektet bruker midtpunktforskyvningsalgoritmen for å implementere stier, og i demoseksjonen brukes stiene til å lage veier og elver. Oppgaven undersøker også hvordan teksturer og kamerabevegelse kan implementeres i en tredimensjonal verden.
Eksperimenter som ble utført testet systemets grenser. Demoseksjonen viser hvordan de implementerte modellene kan brukes til å konstruere mer komplekse objekter. Og mulighetsseksjonen viser hvordan modellene kan brukes for å skape forskjellige terrengrepresentasjoner. Til slutt diskuteres det hvordan prosjektet kan brukes som et grunnlag for fremtidig arbeid og det blir foreslått alternativer for å utvikle programmet videre.

Table of Contents

Glossary

API	=	Application Programming Interface
fBm	=	Fractional Brownian Motion
GLSL	=	OpenGL Shading Language
GLM	=	OpenGL Mathematics
GLEW	=	OpenGL Extension Wrangler Library
CPU	=	Central Processing Unit
GPU	=	Graphics Processing Unit
Heightmap	=	A two dimensional grid of height values
Noise	=	Refers to procedural gradient noise when not otherwise stated.
OpenGL	=	Open Graphics Library. A platform independent API for graphics rendering.
VAO	=	Vertex Array Object
VBO	=	Vertex Buffer Object
EBO	=	Element Buffer Object
PCG	=	Procedural Content Generation.
Tessellation	=	The process of dividing polygons into renderable primitives.
ASCII	=	American Standard Code for Information Interchange

Chapter 1

Introduction

1.1 Motivation

Procedural content generation is increasingly common in video game development. Procedural generation methods are usually implemented by using procedural noise to evaluate large areas of the terrain at once.

The problem with this approach is that objects in the world usually have to be rendered in a specific order to not interfere with each other. And since large patches of terrain are evaluated at once, it is hard to specify what should exist at a specific point within the large patch. Sometimes hierarchical software structures are insufficient to solve a given procedural generation problem.

The claims motivating this project are:

- Is it possible to design a program for rendering a procedurally generated world that merges different parts of the terrain?
- Problems where recursive/ hierarchical structures are insufficient.

1.2 Thesis Goal

The goal of the thesis is to create a program for rendering procedurally generated terrains. The program will not procedurally generate worlds on its own but serves as a back-end rendering solution for worlds already generated. The program will use an input file that specifies the positions and dimensions of objects in the terrain.

Furthermore, the project will explore how different terrain objects can interact with each other and be merged seamlessly. Problems like; When a road crosses a river, a bridge should be generated. Mountains in close proximity should create mountain ranges. Rivers should always flow downstream and how to merge a river with a mountain.

1.3 Research Questions

The thesis aims to explore the following research question:

- What is the right approach for the development of a back-end rendering program for procedurally generated worlds?
- Is it possible for procedural generation methods to move away from noise and search algorithms?
- How can different objects in the terrain be merged?

Chapter 2

Background

2.1 Procedural Content

Procedural generation has been a feature of games for a long time. Procedural generation can be found in video games even before graphically oriented video games. Some of the earliest examples are dungeon-delving games like Rogue(1980) [1] and Beneath Apple Manor(1978). The dungeons in these games are procedurally generated in ASCII or regular tiles to define rooms, hallways, monsters, and treasures.

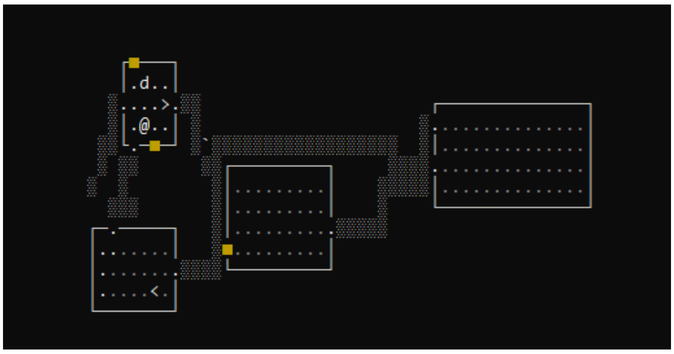


Figure 2.1: A procedurally generated ASCII dungeon in the videogame NetHack [2]

Even though procedural generation has a long history in game development, there has not been much academic interest in the subject until the past decade. The first textbook on the topic was published by Springer [3] in 2016.

2.1.1 Classification of Content

Procedural content generation means creating content for games automatically through the use of algorithms. It is a term closely related to the term procedural generation. Content in this context refers to different aspects of the game. Hendrikx et al. [4] classify content into six main classes that can be generated procedurally. The classes are presented as a pyramid with the most fundamental content at the bottom and more derived classes towards the top.

The first class is **Game Bits**. This class describes individual units of game content like textures, sounds, vegetation, and buildings. These can be thought of as the *bits* that make up the next level of the pyramid, the **Game Space**. The game space classifies indoor and outdoor maps, and bodies of water.

The next level is **Game Systems**. These are the systems that connect the game space. Examples of such systems are ecosystems, road networks, and urban environments.

The fourth level is **Game Scenarios**. The game scenarios are events in the game that describe how the game unfolds. Examples of game scenarios are puzzles, story elements, and levels.

The **Game Design** level describes the rules of the game, what can be done, and what can't. Broadly the game design level is divided into *System Design* and *World Design*.

The last level is the **Derived Content**. This level describes the content that is not created inside the game world but as a side-product. Examples include leaderboards to rank individual players, and news boards to announce changes to the game world.

2.1.2 Methods of Procedural Generation

Togelius et al. [5] provide a taxonomy for classifying different methods of procedural content generation. Their distinctions are meant to serve as a continuum rather than a binary. They specify that procedural generation methods usually lie on a spectrum between two extremes.

The first distinction described is **online versus offline** content generation. Offline content generation means generating content during development time, or before the end-user starts a game session. Online content generation, on the other hand, creates during runtime. When using offline generation during development time, the algorithm can suggest an overall layout of the terrain, but can then be edited by a human designer before releasing the game.

The Elder Scroll IV: Oblivion developed by Bethesda Game Studios is an example of a game where an offline procedural terrain generator was used during development time to reduce the time taken to create the game world [6].

The second distinction is **necessary versus optional content**. The player of the game is required to complete all necessary content to progress in the game, whereas optional content is what the player can ignore. The requirements for quality in necessary content are far greater than the requirements for optional content. Games like Diablo, Borderlands, and Destiny have randomly generated weapons and equipment. Several of these weapons will not be useful to the player, but exploring and comparing them is a central part of the game. Necessary content always needs to be correct, unbeatable monsters or unsolvable puzzles would never be accepted.

The third distinction is **random seeds versus parameter vectors**. At one extreme, the only input to the procedural generator is a seed for the random number generator. While the other extreme the algorithm takes in a multidimensional vector that specifies the properties of the generated content. For a terrain, these properties could be how many moun-

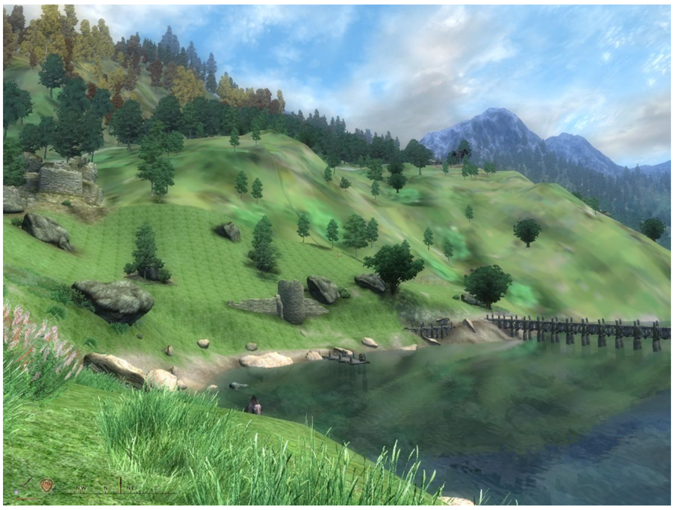


Figure 2.2: The terrain in The Elder Scrolls IV: Oblivion. Image from [7]

tains are created, how clustered the mountains are, the maximum height of the terrain, the amount of water in the terrain, etc.

The fourth distinction is **stochastic versus deterministic generation**. This distinction can also be seen as the level of randomness in the generation. How far towards the stochastic extreme the procedural generation falls is an expression of how different two worlds with the same input parameters are. A completely deterministic procedural generation algorithm can be viewed as a form of data compression.

The final distinction is **constructive versus generate-and-test**. A constructive algorithm generates content once and is done with it. All testing to make sure the content is correct is done during generation. Limits are put on the generation to guarantee the algorithm never produces broken content. In a generate-and-test algorithm, parts of the generated content can be broken when tested according to some criteria, and may then be discarded or regenerated. This distinction is similar to the distinction between implicit and explicit generation presented in section 2.4.

2.2 Basics of Computer Graphics

Three-dimensional computer graphics are made of a series of connected vertices that form polygons. These polygon surfaces are often called primitives. The primitives don't have to be triangles. They can be quadrilaterals or higher as well. However, most computers are optimized to rendering triangles because every polygon can be split into triangles. A set of polygons can be combined into a polygon mesh, usually just called a mesh.

2.2.1 Textures

In the real world, the word *texture* refers to the surface characteristics and appearance of an object. When doing computer graphics, the concept of texture is simplified. In its purest form, the texture is just a single color applied to an object, but the texture is more often a two-dimensional image. Much work goes into making the two-dimensional textures look more realistic.

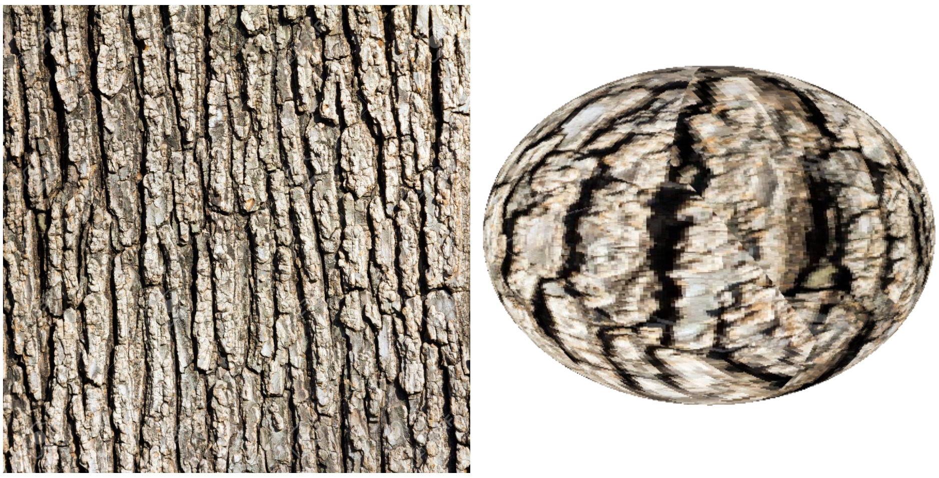


Figure 2.3: Two-dimensional texture mapped onto a three-dimensional sphere. Texture from [8]

2.3 Terrain Representation

When rendering large worlds with plenty of terrain variety, and an efficient way of representing terrain is needed. Terrain representation that allows for higher levels of detail requires more powerful computer hardware, while more straightforward terrain representations are limited in scope. The following sections present a few terrain models, and describe their limits on performance and supported levels of detail.

2.3.1 Heightmap

A heightmap is a two-dimensional grid of height values. The value stored at each grid point is the offset from the lowest point in the terrain. Heightmaps are a highly efficient way of storing terrain data since only one float value is required for each position in the terrain. One of the drawbacks of heightmaps is that overhangs and caves cannot be created since there is only one height value at every point in the terrain. Another problem is the even spacing of the heightmap grid makes it hard to implement a varying level of detail function.

Even though heightmaps have restrictions on the available terrain geometry, they are still used in modern game engines like the Frostbite engine for Battlefield 3 [9] and Unreal

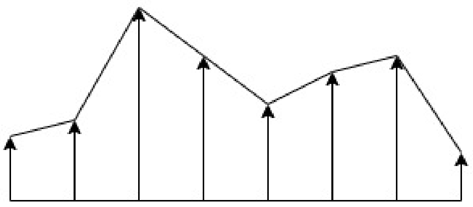


Figure 2.4: Heightmap

Engine [10].

2.3.2 Vector Displacement Map

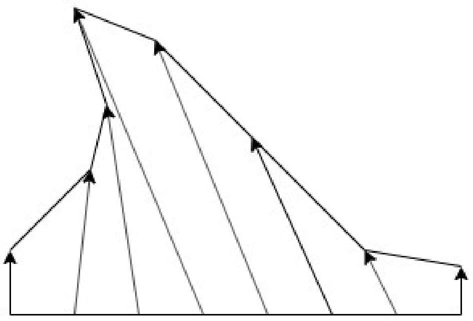


Figure 2.5: Vector Displacement Map

Vector displacement maps are similar to heightmaps, but with three floating values at each point rather than one. Thus the same position can have several height values. With vector displacement maps, it is possible to make overhangs and caves with one entrance. It is also possible to move vertices from areas where not much detail is needed to areas where more detail is required, effectively using the terrain model to implement a varying level of detail function.

McAnlis [11] describes how vector displacement maps were used to create terrains in the game Halo Wars. McAnlis also explains how their artificial intelligence systems had a harder time navigating a vector displacement map than a heightmap. Hence they developed a method of converting displacement maps to heightmaps for artificial intelligence navigation. Since displacement maps and heightmaps are so similar, many of the same algorithms can be used on them.

2.3.3 Voxel Terrain

Voxel terrain is a three-dimensional grid of terrain data. At every coordinate, there is terrain data like water, dirt, air, wood, etc. How smooth the terrain looks is determined

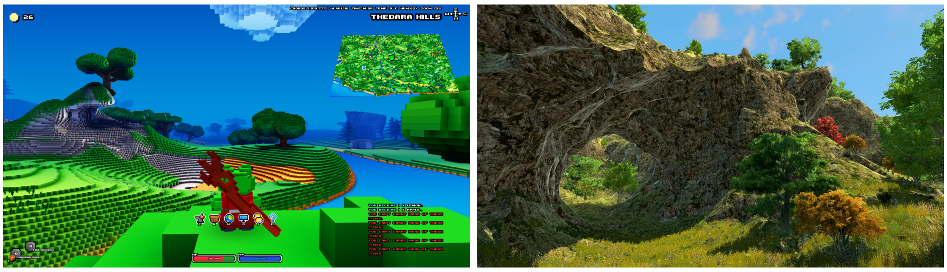


Figure 2.6: Cube voxel terrain and smooth voxel terrain. Smooth image from [12]

by how small each voxel coordinate is. In voxel terrains there can be caves and floating islands, there is no requirement for the terrain to be connected like in displacement maps.

Game developers have ignored voxel terrains because of the massive disk space required to store the grid. However, procedural generation can generate the grid at run time, effectively becoming a form of data compression, see section 2.1.2. The game Minecraft uses a noise function, see section 2.4.2, along with voxel terrain to generate its worlds.

2.4 Fractals

A fractal is a never-ending pattern. Fractal patterns repeat themselves at increasingly smaller scales as they are zoomed into. In his 1982 book *The Fractal Geometry of Nature* [13], Mandelbrot claims terrains have a self-symmetry similar to fractals. Mandelbrot suggested using fractional Brownian motion(fBm) as a good approximation of terrain.

Using fractional Brownian motion to generate terrain can be done either explicitly or implicitly. Explicit terrain generation means evaluating a large batch of noise values at once. When using such a technique, individual points in the terrain cannot be evaluated without generating the rest of the model. Implicit generation, on the other hand, can generate arbitrary points independent of other points. Musgrave [14] calls the implicit approach *point evaluation*, and in Gamito and Musgrave [15] they call it *stochastic implicit surface modelling*.

2.4.1 Explicit Methods

The diamond-square algorithm is a method for approximating fractal noise first introduced by Fournier et al. [16]. The algorithm generates a grid of height values that can be used as a heightmap. The algorithm is also known as the *cloud fractal* or the plasma fractal because the generated heightmap looks ”plasma-like”.

The diamond-square algorithm works like this:

1. Create an array of height and width $2^n + 1$ with initial values at the four corners.
2. **The diamond step:** For each square in the array, set the midpoint to be the average of the four corners plus a random value.

3. **The square step:** For each diamond in the array, set the midpoint to be the average of the four corners plus a random value.

4. Reduce the magnitude of the random value, and repeat the diamond step and square step until all values in the array are set.

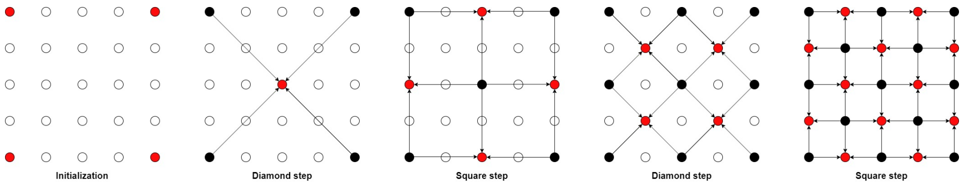


Figure 2.7: Visualization of the diamond-square algorithm

2.4.2 Noise

Noise on its own does not resemble fractional Brownian motion. For procedural purposes, the term procedural noise has been used to describe a form of interpolated noise. Interpolation makes transitions between high and low values in the noise smoother. An approximation of fractional Brownian motion can be constructed when several layers of procedural noise with different frequencies and amplitudes are combined.

Perlin noise is one of the most commonly used procedural noise methods. It was described initially by Ken Perlin in [17]. Perlin noise is a form of gradient noise, and many variations of Perlin's original implementation have been made since then. Perlin even published his improved version of the algorithm 17 years after the initial publication [18].

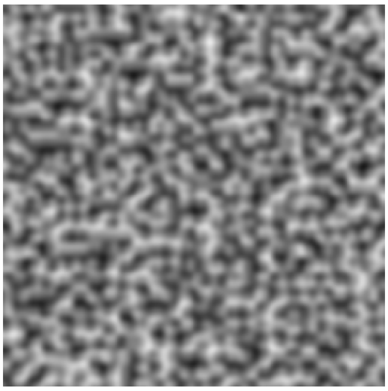


Figure 2.8: Example of perlin noise

2.4.3 Implicit Methods

Implicit terrain generation techniques can generate individual points without checking the data of other surrounding points. Methods of implicit generation usually rely on a self-contained mathematical model. For a heightmap, this means getting the height at a specific position when inputting the x and y coordinate, output a vector for vector displacement, and a density value for voxel terrains. Where the density value determines kind of material is rendered at input position. Saupe [19] presents a method to compute a random fractal function with 1-3 variables.

2.5 Path Generation

Path generation can be split into search-based approaches and non-search based approaches. Non-search approaches don't care what already exists in the terrain. The path is generated from one point to another, and everything else either has to be removed or merged. Search based approaches, on the other hand, search through the terrain for the optimal path based on restrictions.

2.5.1 Search-Based Approach

One example of a search based approach is the A* algorithm. Norvig and Russel [20] describe it as "the most widely known form of best-first search." It is based on a variant of the very well known *Dijkstra's algorithm.* The main difference is that A* does not search through every node in the set of nodes; it only adds nodes to the list as they are discovered. That makes the algorithm more efficient since it uses less memory and becomes more useful when finding a path through more massive graphs. This efficiency is essential when used in a procedural generation program since procedural worlds can get infinitely large.

2.5.2 Merge-Based Approach

When paths are generated without a search-based approach, it is necessary to merge the generated path with the terrain unless the terrain is chosen to be ignored. Interpolation is a method used to create data points within a specific range. When a road goes through a mountain, interpolation can be used to generate a smooth transition from the road to the mountain.

Bézier curves are a way of interpolating between several anchor points. Since Bézier curves can be extended to n-dimensions, it can be used to smooth out the transitions between several objects in the terrain. From Floater [21]: A Bézier curve of degree n is a parametric polynomial **p** given by the formula:

$$\mathbf{p}(t) = \sum_{i=0}^{n} \mathbf{c}_i B_i^n(u) \tag{2.1}$$

Where $\mathbf{c}_i$ are the anchor points, and B_i^n is the Bernstein polynomial:

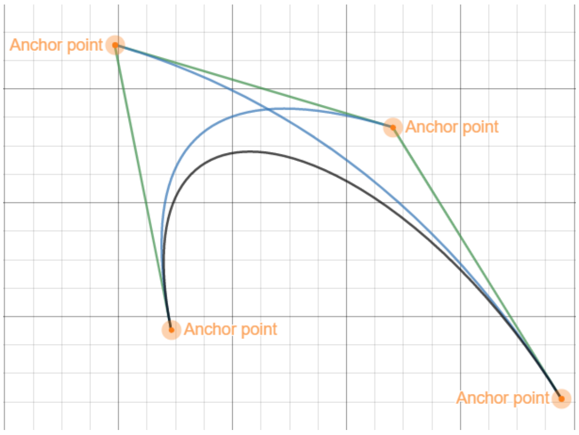


Figure 2.9: A cubic Bézier curve

$$B_i^n(u) = \binom{n}{i} u^i (1-u)^(n-i) \tag{2.2}$$

2.5.3 Midpoint Displacement

The midpoint displacement algorithm can be used to generate a path with pseudo-random offsets. It works by starting with a straight line. The midpoint of the line is found and displaced proportionally to the length to the midpoint. This procedure is repeated for the two new line segments until a sufficiently curvy path is generated.

The algorithm is usually used to generate fractal terrains. By expanding it to two dimensions, it becomes similar to the diamond-square method presented in section 2.4.1. However, rivers are naturally curvy, and roads are usually not entirely straight either. That makes the midpoint displacement algorithm suitable for generating paths as well.

2.6 OpenGL

OpenGL is a platform-independent graphics rendering specification developed by the Khronos Group [23]. By itself, it is a language-agnostic specification, and implementations have been developed for most common programming languages. Developers of OpenGL implementations can freely decide how functions are implemented since OpenGL only specifies what the output must be.

The term shader program used to describe the part of the computer program responsible for shading an object with appropriate levels of light, darkness, and color. Modern shader programs perform a variety of tasks in graphics rendering. Broadly shaders can be defined as code that runs on the GPU instead of the CPU.

When using OpenGL, there is typically a client program that runs on the CPU, and a shader program that runs on the GPU. Before the release of OpenGL 2.0 in 2004, the

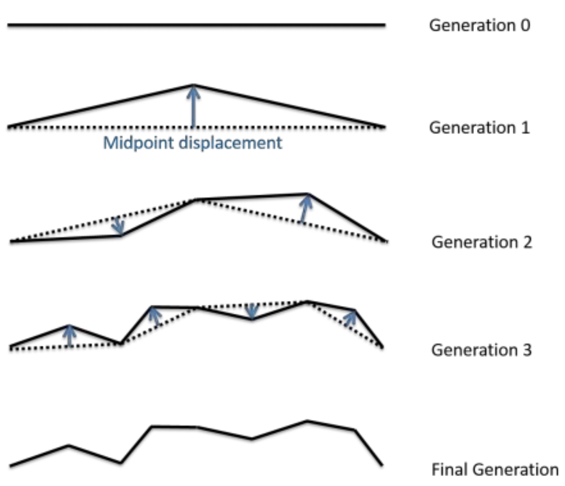


Figure 2.10: Midpoint displacement algorithm visualized. Image from [22]

shader program was fixed [24]. OpenGL 2.0 introduced programmable shaders, and the level of shader program customization has increased since then.

2.6.1 Vertex Buffering

OpenGL uses objects to classify what data should be rendered and how the data should be rendered. The three most common OpenGL objects are Vertex Array Objects(VAO), Vertex Buffer Objects(VBO), and Element Buffer objects(EBO). A VBO stores vertex data like vertex position, texture coordinates, and color. The VAO stores attribute data that explains how the data in a VBO is stored and how it should be rendered. A VBO must be ”bound” to a VAO for the data in the VBO to be rendered. The EBO stores the index information and is used when the same vertex is used in several triangles in the same VBO.

2.6.2 Shaders

OpenGL provides five different shaders as part of its pipeline. When shaders became programmable with the release of OpenGL 2.0, it had only two different shaders. The most recent addition, the Compute Shader, was added in OpenGL 4.3 (2012).

- **The Vertex Shader** handles the rendering of individual vertices. It takes in vertex attribute data specified by a vertex array object.
- **The Tesselation Shader** is an optional shader in the OpenGL pipeline. It deals with subdividing patches of vertex data into smaller primitives.
- **The Geometry Shader** is an optional shader that processes primitives.
- **The Fragment Shader** deals with color and the final depth value of each pixel on the screen.

- **The Compute Shader** deals with computing information. This stage of the pipeline cannot do rendering and is used for tasks not directly tied to triangles and pixels.

2.7 Mathematics

2.7.1 Transformations

The theory in this section is based on the book by Gravdahl and Egeland [25].

It is useful to have an understanding of fundamental transformation matrices to move objects around in 3D space. Transformation matrices are used to move from one coordinate system to another coordinate system.

The equation for the homogeneous transformation matrix can be seen in eq. (2.3), and in [25], equation 6.115.

$$\mathbf{T}_b^a = \begin{pmatrix} \mathbf{R}_b^a & \mathbf{r}_{ab}^a \\ \mathbf{0}^T & 1 \end{pmatrix} \tag{2.3}$$

The homogenous transformation matrix is made up of a rotation matrix $\mathbf{R}_b^a$, and a translation vector $\mathbf{r}_{ab}^a$. The elements of the rotation matrix depends on the axis of rotation, all of which can be found in eqs. (2.4) to (2.6), and in [25], equations 6.101 to 6.103.

$$\mathbf{R}_x(\phi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{pmatrix} \tag{2.4}$$

$$\mathbf{R}_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix} \tag{2.5}$$

$$\mathbf{R}_z(\psi) = \begin{pmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{2.6}$$

The translation vector is given by the translation along each axis and is presented in eq. (2.7)

$$\mathbf{r}_{ab}^a \begin{pmatrix} \mathrm{t_x} \\ \mathrm{t_y} \\ \mathrm{t_z} \end{pmatrix} \tag{2.7}$$

Chapter 3

Specification

3.1 Project Specification

The goal of the project is to develop a program for rendering procedurally generated terrain, and the program should also be *playable*. An end-user can input properties of the desired terrain, and even move around in the rendered world. The project will also attempt to texture the world to appear realistic, but this is not the main focus of the program. The appearance of the world should be sensible enough not to break immersion, but making the terrain look entirely realistic is not a primary goal. The software structure should be non-hierarchical, which means to build around design principles like *models creating models*, and non-search based object generators.

The input to the program will be a simple text file that lists properties of the terrain to be generated. The user will also be able to explore the rendered world by walking or flying. There is no requirement for the input file to specify all the different areas of the world, and the program should fill the world with variety where there is nothing specified. Lastly, some properties of the world should interact with each other. Roads crossing rivers create bridges, roads connect multiple towns, rivers flow downstream, mountains close to each other create mountain ranges. The input file will be on the form *mountain at position (10,10)*, *tree at position (25,5)*, etc.

One major research question is how to make the different properties of the world join together effectively. If a road and mountain are generated in the same position, does the road go over the mountain, or is it divided, creating a mountain pass? Rivers need to either search for a path sloping downhill or change the height of the terrain where it passes.

The project aims to move away from a hierarchical software structure. The order in which objects in the world are generated should not matter, and no algorithms should have to search through what has already been created. In general, the idea is to work on a shared state where objects generated later redefine what was created before or discard it if it no longer fits. A new object always has priority to avoid searching for a suitable place to create, and ideally, it will merge seamlessly with the terrain already there.

The project will use OpenGL for graphics rendering. Different ways of representing terrain will be explored. OpenGL is a low-level graphics specification, where different parts of the rendering pipeline are programmable. Different styles of representing terrain are to be studied and implemented. The programmable sections of the OpenGL rendering pipeline will also be examined to explore how they can be used to render variation in the procedural terrain.

3.2 Summary of Specification

The following features are to be implemented:

- Create a program for back end rendering of a procedurally generated world
  - The end user will be able to change the properties of the generated terrain by changing the input file
  - It will be possible to explore the generated world by walking or flying
  - The world should feel realistic enough to not break immersion
- Use a non-hierarchical software structure
  - Rivers and roads should not *search* the terrain
  - Use the *models generating models* principle
  - Implicit procedural techniques instead of explicit techniques
- Make properties of the generated world join together
  - Roads crossing rivers create bridges
  - Close proximity mountains create mountain ranges
  - Rivers must flow downhill
- Explore different ways of representing terrain using OpenGL
  - Heighmaps, vector fields and voxels
  - Meshes for model representation
  - Study the programmable shaders of OpenGL

Chapter 4

Design

4.1 Software Design

4.1.1 Textures

How textures are implemented makes a big difference in how the terrain looks. The most basic and intuitive implementation is to fill each square of the terrain with precisely one texture. If performance is a concern, the world could be made more vertex efficient by mirroring the textures along the edges of each polygon in the terrain. Both of these approaches do not attempt to hide the underlying ”squareness” of the texture. It would be highly inefficient for each polygon to have a unique texture, but another solution is to change the same texture slightly for each polygon using it. Lighting techniques can make each polygon look somewhat different, and if using seamless textures, the texture can be rotated slightly at each polygon.

A texture atlas was designed to access different textures. A texture atlas means putting every used texture into the same image file, and then the individual textures are chosen by specifying the coordinates of the texture in the file. The end atlas design supports 100 textures of size 512x512pixels, but only 11 of these spots were used in the end. A complete list of all textures and their name, when called by the input file, can be seen in fig. 4.1. All textures found on [26].

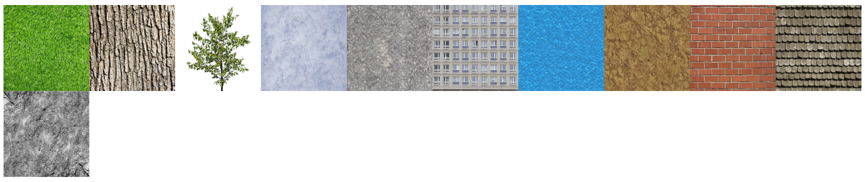


Figure 4.1: Texture names starting top left: grass, bark, branch, snow, road, apartment, water, dirt, brick, roof and stone

4.1.2 Using OpenGL

As mentioned in section 2.6, OpenGL is language-agnostic, and there exist implementations for most programming languages. For this project, C++ was chosen as the design language. One of the advantages of using OpenGL instead of a full game engine like Unity or Unreal Engine is that one gets to learn graphics programming on a much more fundamental level. That is one of the reasons C++ was chosen as well. Another advantage is that the OpenGL shader language resembles C++.

For OpenGL to work with C++, a couple of additional dependencies and useful libraries were used. The first is GLFW [27]. GLFW is a library that provides simple API for creating windows, contexts, and surfaces. It was used to create the window. The second is glad.c [28]. Glad provides OpenGL functions based on official specifications. It provides functions for creating OpenGL objects, binding, rendering, and much more. The third is OpenGL Mathematics (GLM) [29]. GLM has datatypes and mathematical functions useful for many OpenGL projects. In this project, the library was used for the camera movement and transformation matrices. The last extension used was GLEW [30]. Glew is an extension loading library and makes three first libraries work together.

4.2 Camera

After vertex coordinates have been processed in the vertex shader, they will be in normalized device coordinates. Normalized device coordinates are a small space where the x,y, and z values vary from -1.0 to 1.0, and any vertex coordinates that fall outside this range are discarded.

OpenGL has no concept of a ”camera,” but by using transformation matrices on the vertex coordinates of a rendered model, a camera can be simulated. For instance, if a model of a tree is loaded, but only the bottom half of the tree falls within the normalized device coordinates. A function that rotates every vertex of the tree downwards about a coordinate system oriented to ”look” straight at the normalized device coordinates can be used to simulate camera movement. If the function is implemented to rotate when the computer mouse is moved upwards, it will seem like a camera is being pointed upwards with the mouse’s movement.

4.3 Mountains

Changes in height are an essential part of rendering natural-looking terrain. Different kinds of mountains can vary a lot in their visual complexity. In this project, the method used to render mountains needs to be called multiple times to fill the terrain with height variety. The amount of information required to render a mountain should, therefore, be kept to a minimum.

To create a mountain range, a function that recognizes the proximity of two mountains can be used to issue the generation of several smaller mountains between the two original mountains.

4.4 Roads and Rivers

As described in section 2.5, path generation algorithms can be either search-based on non-search based. To keep to the principles of non-hierarchical software structures, the search-based algorithms are not suitable for this project. The advantage of search-based approaches is that a path that fulfills specific requirements can be found without modifying the underlying terrain. When using a non-search based approach, the specifications can still be fulfilled, but the terrain where the path is generated might have to be modified.

Rivers have stricter requirements than roads in that they have to flow downstream. The underlying terrain will have to be modified after the river is generated. A Bézier curve could be used to interpolate the terrain from the edge of the river outwards, making sure the anchor points go up first before flattening.

Chapter 5

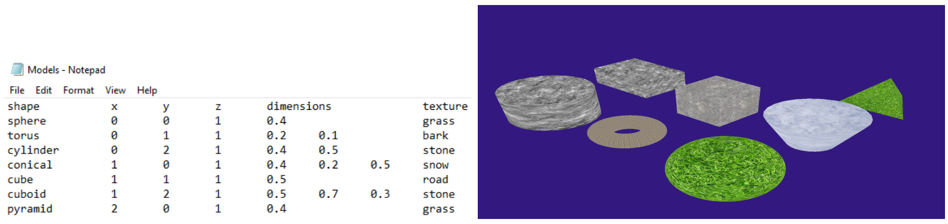
Implementation

5.1 Using the Program

The project was implemented as a Microsoft Visual Studio solution. The complete solution can be found on GitHub [31]. The additional dependencies for the preprocessor and linker are already taken care of if the source code is compiled with Visual Studio. Otherwise, the preprocessor needs to include GLEW_STATIC, and the linker requires glew32s.lib, glfw3.lib and opengl32.lib.

5.1.1 Input

The source code folder also includes the input file *Models.txt*. This file can be changed to add models to the terrain. It allows for the creation of basic models and geometrical shapes by specifying their position, dimensions, and texture.



```
shape       x     y     z     dimensions          texture
sphere      0     0     1     0.4                 grass
torus       0     1     1     0.2   0.1           bark
cylinder    0     2     1     0.4   0.5           stone
conical     1     0     1     0.4   0.2   0.5     snow
cube        1     1     1     0.5                 road
cuboid      1     2     1     0.5   0.7   0.3     stone
pyramid     2     0     1     0.4                 grass
```

Figure 5.1: Example of a Models.txt input, and the rendered geometrical shapes

Since each model has different dimensions, a complete table of every model and their expected input parameters can be found in table 5.1. The table also shows whether a model accepts a texture input or not.

Model	Position			Parameters				Texture
Sphere	x	y	z	Radius				Yes
Torus	x	y	z	Inner radius	Outer radius			Yes
Cylinder	x	y	z	Radius	Height			Yes
Cube	x	y	z	Length				Yes
Conical	x	y	z	Top radius	Bottom radius	Height		Yes
Cuboid	x	y	z	Length	Width	Height		Yes
Pyramid	x	y	z	Length				Yes
Road				Start x	Start y	End x	End y	Yes
Mountain	x	y		Height	Radius			No
Tree	x	y						No

Table 5.1: Complete list of models and their input parameters

5.2 Rendering Mesh Data

In this project, a mesh is implemented as a structure with vertex positions, texture data, and index data. To be rendered, the mesh data needs to be transformed into objects that OpenGL can understand. The mesh structure is transformed into a model object. In the model, the vertex positions and texture coordinates become vertex buffer objects(VBO), and the indices become an element buffer object(EBO). Every model also has a vertex array object(VAO). Lastly, all created models send their data to the renderer object. When the renderer calls the OpenGL function glDrawElements, all the data of the currently bound VAO gets rendered. The renderer object binds every VAO once every application loop to draw every model created.

5.2.1 Terrain Model

In section 2.3 different terrain models were introduced. For this project, the terrain type implemented was heightmaps. The implementation could easily be changed to a vector displacement implementation since the heightmap already stores length and width coordinates. Still, there are no functions that change the offset of the length or width. Voxel grid representation is not supported as an input datatype, but with the input file's creative use, a voxel terrain can still be rendered, see section 6.3 and section 7.1.3.

5.2.2 Spheres

Song Ho Ahn [32] was used as inspiration to create the code for creating spheres made up of triangles. Their method starts with an icosahedron, a convex polyhedron with 20 faces. Then the midpoint of each edge line is extruded such that the distance from the center to the polyhedron equals the radius, subdividing the original triangle into four new triangles. This procedure can be repeated until the polyhedron reaches the desired ”roundness.”

Unless otherwise stated, the spheres in this project are created with a subdivision of 4, creating spheres made up of $20 * 4^4 = 5120$ triangles. Subdivisions of up to 10 were

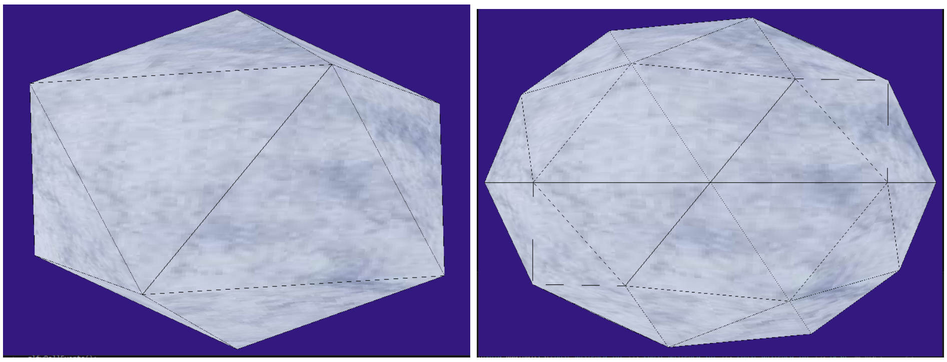


Figure 5.2: Spheres with subdivision 0 and subdivision 1

tested. At subdivision 10, the program failed to allocate the space required by the sphere array. This limitation will be further explored during testing in section 6.1.

5.2.3 Textures

Two different texture implementations were explored. One implementation with four texture coordinates per square in the terrain. The other with an average of two texture coordinates per square. The second implementation works by mirroring the texture in the adjacent square. The first implementation allows for smoother transitions between textures but has higher computer requirements for equal performance.

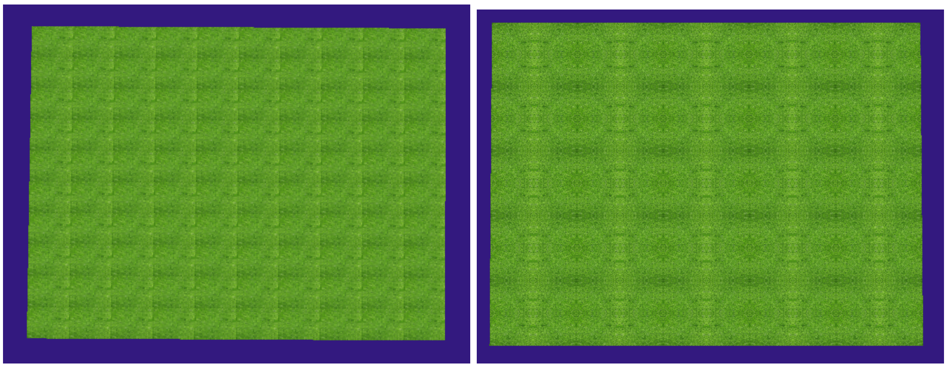


Figure 5.3: 10x10 grid implemented with 4 texture coordinates per square(left) and with two texture coordinates per square(right)

5.3 Mountains

The rendering of mountains was implemented using the two-dimensional Gaussian function. This function works well for our purpose because of its simplicity. Input to the function is amplitude and standard deviation, matching the minimum requirements for a mountain, peak height, and radius.

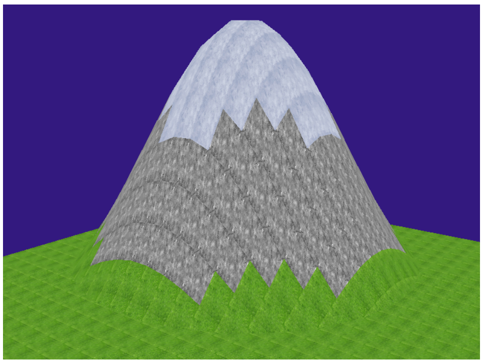


Figure 5.4: Basic bell curve mountain

Figure 5.4 shows an example mountain with height 20 and radius 10. How the mountain gets textured is based on the height of each square. Bottom squares get textured with grass, middle squares with stone and the top squares with snow.

5.4 Roads and Rivers

Roads were implemented using a midpoint displacement algorithm. The input file describes the two ends of the road, and by displacing a series of points between the two ends, a more curvy road is generated. Functionality specifically for rivers was not implemented, but if the texture for a road is changed to *water*, the path created will resemble a river. Figure 5.5 shows a primary river with midpoint displacement active, and a road with midpoint displacement inactive. Unless otherwise stated, every road will be generated with midpoint displacement activated.

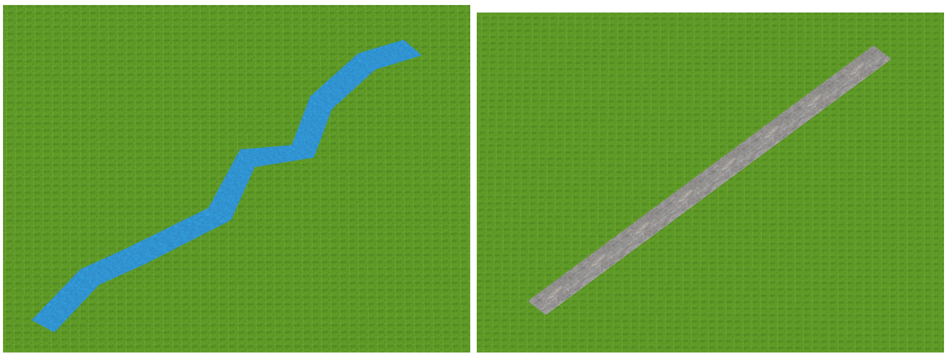


Figure 5.5: Basic river and road

Chapter 6

Results

6.1 Testing

During the implementation of the sphere model, it was noted that the program failed to allocate space for a sphere with subdivision 10 or more. A sphere with a subdivision of 9 has $20*4^9$=5 242 880 triangles. While a sphere with subdivision 10 has four times as many.

Each triangle is made up of three vertexes. Every vertex has three floating numbers for vertex position, two floating numbers for texture coordinates, and one unsigned int for index value. In c++ floating, numbers and unsigned ints have a size of 4 bytes. This means each vertex requires 24 bytes, and the total capacity needed for a sphere of subdivision 9 is 377 487 360 bytes.

After checking the limit for the number of triangles in one sphere, the maximum amount of spheres of subdivision four was tested. Microsoft Visual Studio provides diagnostic tools that show the time elapsed since the diagnostic session started, the amount of process memory used, and CPU usage as a percentage amount of all processors. The resultant allocation time, peak process memory at the end of allocation time, and the stable process memory usage after allocating time for an increasingly considerable number of spheres is shown in table 6.1. NA means the allocation did not finish.

The computer specifications used for the experiments are listed in table 6.2.

6.2 Demo

The input file will be filled with cubes, cuboids, mountains, and other models to showcase what kinds of worlds can be rendered. The project is designed to render a procedurally generated world. Since no procedural generation software was developed, this section will showcase what it looks like when the input file is filled with random models.

By using the apartment texture on cubes or cuboids, something resembling a building can be rendered. These buildings can be clustered to make something resembling a small

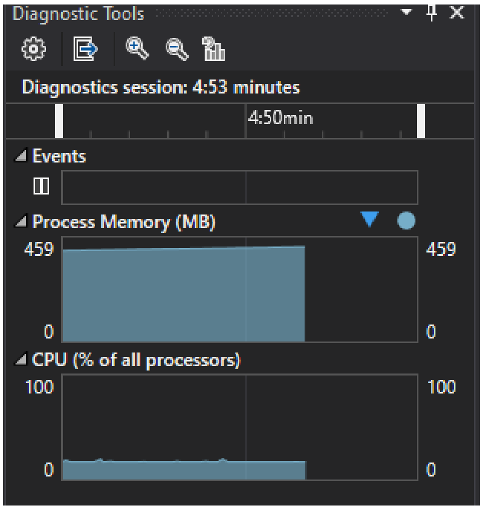


Figure 6.1: Example of the diagnostic tools provided by Visual Studio

Spheres	Triangles	Allocation time	Peak	Stable
1 000	5 000 000	0:56	311.5MB	214.3MB
2 000	10 000 000	1:54	384.2MB	290.5MB
4 000	20 000 000	3:49	531.5MB	336.5MB
8 000	40 000 000	7:41	821.2MB	628.3MB
16 000	80 000 000	15:26	1.4GB	1.2GB
18 000	90 000 000	18:10	1.7GB	1.7GB
20 000	100 000 000	NA	1.8GB	NA
32 000	160 000 000	NA	1.8GB	NA

Table 6.1: Testing allocation time and process memory usage for higher amount of spheres

GPU	Gigabyte GeForce RTX 2060 CC
CPU	Intel Core i5-9600K
Memory	16GB DDR4 3200MHz
Storage	960GB M.2 SSD
Buid Year	2019

Table 6.2: Computer Specifications

town. The small towns are spread out to pseudo-random locations to give the terrain some variety, and then the towns are connected with roads from their center. An example town can be seen in fig. 6.2, and a small world with multiple towns and mountains can be seen in fig. 6.3.

The mountain model by itself can be used to give the terrain lots of height variety. Figure 6.4 shows many mountains near each other with varying height and radius.

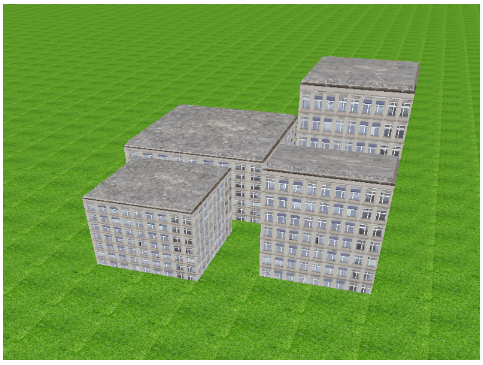


Figure 6.2: Using cubes and cuboids to render buildings

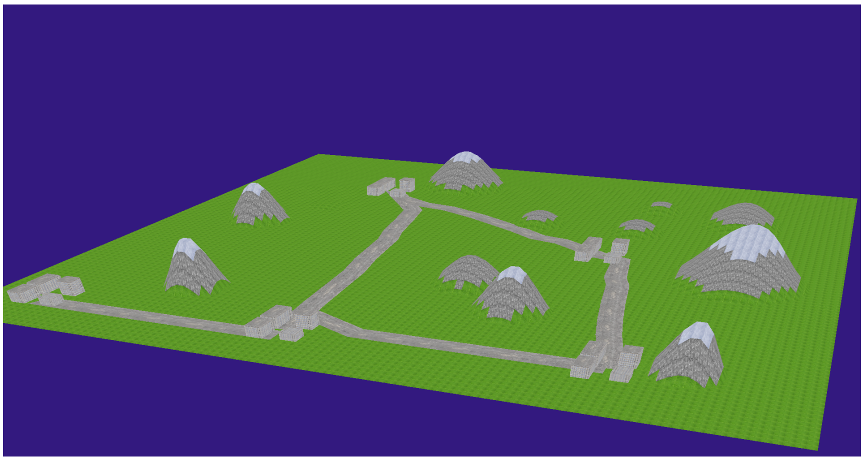


Figure 6.3: Rendering multiple small towns and mountains

6.3 Possibilities

Even though voxel terrain isn't supported as a data type, it is still possible to render a voxel terrain using a pre-generated voxel grid as input. Figure 6.5 shows how cubes or spheres can be used to generate a voxel grid.

There are no restrictions on how the geometrical models may be used, plenty of things that have nothing to do with terrain can be rendered. The program can be used to render many different models for demonstration or visualization. Figure 6.6 shows how conical and cylinder models can be used to render a rocket.

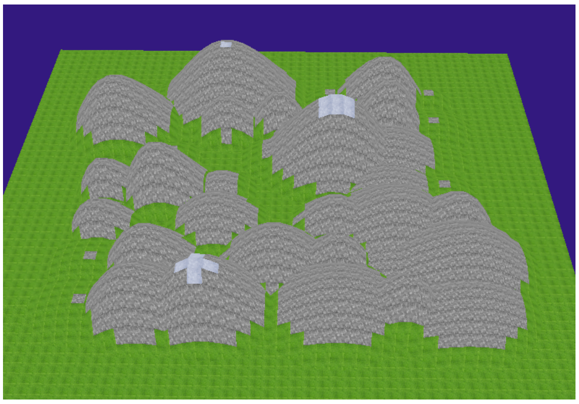


Figure 6.4: Many smaller mountains

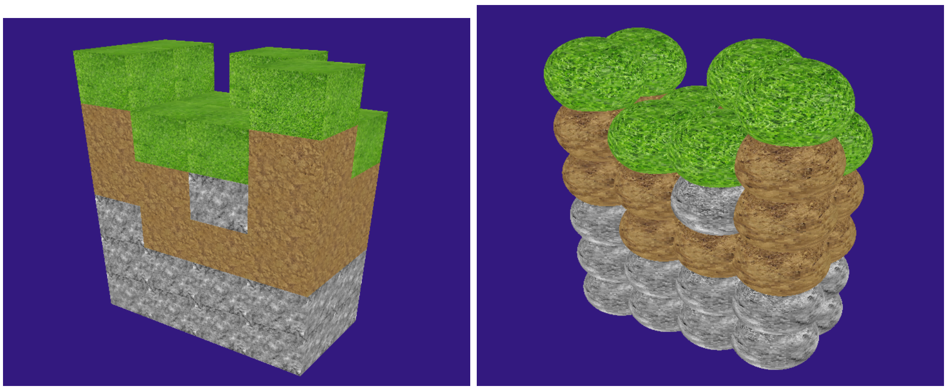


Figure 6.5: Using cubes and spheres to render a voxel grid

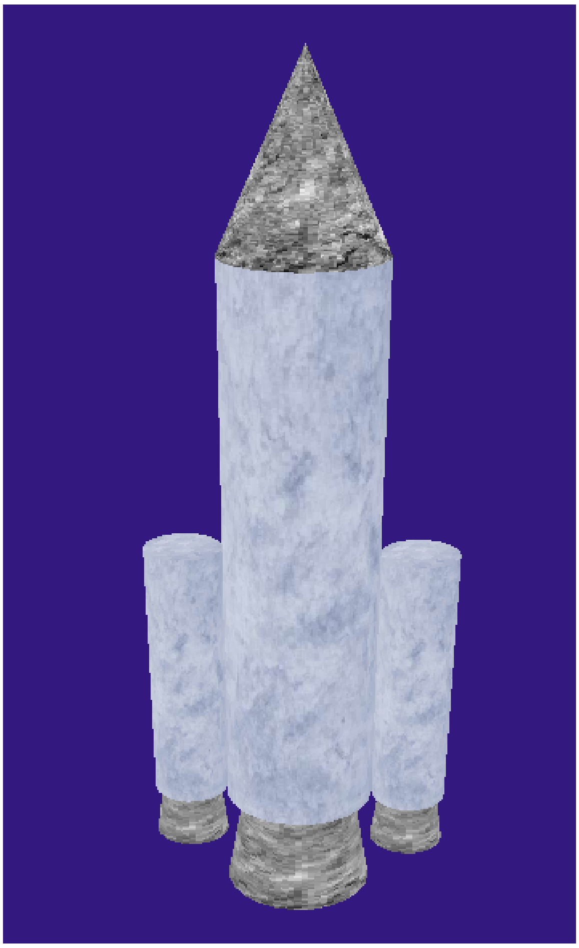


Figure 6.6: Rocket model

Chapter 7

Discussion

7.1 Results

7.1.1 Limits

The purpose of the limit tests was to find out when the program fails, and what makes it fail. After the initial realization that a sphere with subdivision 10 made the application fail, the proposed test was aiming to figure out if there was a limit was in the number of triangles able to go rendered or if it was a storage allocation issue.

A sphere with subdivision 10 has 20 971 520 triangles, and when the allocation failed, the amount of process memory used was 1.8GB. Spheres with subdivision four consists of 5120 triangles, which means 4096 spheres with subdivision 4 have the same amount of triangles as one sphere with subdivision 10. Looking at the results in table 6.1, we see that rendering many triangles is not the problem. The problem is when process memory usage reaches 1.8GB. The program always fails when reaching 1.8GB.

When rendering a sphere with subdivision 10, it takes the program less than 2 minutes to reach the peak process memory usage. When generating 18 000 spheres with subdivision 4 takes more than 18 minutes to reach the same process memory usage. The extra time is taken to read the input file. An input file with 18 000 lines had to be read, to render the 18 000 spheres. To change subdivision to 10, it is merely a change of a constant in the code. A more suitable test would have been to generate the 18 000 spheres without an input file by simply requesting 18 000 spheres directly in the system, but no such analysis was performed.

It is not clear what the process memory usage represents. The highest amount of triangles tested for multiple spheres that did not crash was 90 000 000. This equals more than 6GB of memory to store the vertices. A completed sphere of subdivision 10, will use 1.5GB of storage. Since the subdivision method is recursive, the spheres of the smaller subdivision are still using memory while the last sphere is being subdivided. However, the triangle total for all spheres subdivision 1-10 is still less than 90 000 000. Why the program crashes at specifically 1.8GB of process memory is still a mystery.

7.1.2 Demo

For demonstrating a proof of concept, the demo section fulfills its purpose. The variety of the rendered terrain is limited by the few models that have been implemented. Still, when using the input format correctly, the program does render both procedurally and non-procedurally generated worlds. Another limiting factor is how the geometrical models were used. The cubes, cuboids, and pyramids can be used to create basic building structures, and the conical can be used to make straight trees. The project has plenty of options in models and textures, and how creatively these options are used will make a better-rendered world.

The implementation of the tree model could have been done better. Trees were one of the first models with implementation, but because of the inability to rotate the basic geometrical shapes, and difficulty in getting texture orientation to work better. Trees were put on hold and never finished. Ideally, the trunk and the thicker branches would be implemented using the conical model, and thinner branches would use the branch texture to fill the tree with leaves, but this implementation was not finished.

7.1.3 Possibilities

Lastly, the extra possibilities section shows that the input can be used in creative ways to render terrain models like voxel terrains, even when voxel terrain is not an implemented datatype. The geometric models can also be used to render models that have nothing to do with terrain representation. The creativity from the user of the program the only limiting factor in that respect.

7.2 Future work

7.2.1 Models

In the future, more models could be implemented. Right now, cubes, cuboids, and pyramids are used to make buildings and houses. The next step would be to implement a ”building” model or a ”city” that automatically makes buildings and cities using the basic geometrical models. Other natural terrain models could be added as well. Some suggestions are a ”waterfall” model, a ”bush” model, and more complex models like animals and insects.

The models also suffer from the inability to rotate. The rendered models always have the same orientation, which hinders an end-users ability to combine the basic geometrical models into more exciting objects.

7.2.2 Terrain Models

The base of the terrain is a heightmap terrain, and it is possible to render a voxel terrain using a voxel grid in the input. It would be interesting to implement the ability to choose between heightmap or voxel when rendering a world filled with models like mountains, trees, and cities.

7.2.3 Textures

The current texture implementation looks a bit bland. It is effortless to see the edges of the textures in each square of the terrain. A texture implementation that tiles the same texture in a way that makes it look less similar would make the grass look a lot better. A texture implementation that interpolates between two different textures would make the texture transition between grass, stone, and snow on the mountain look smoother.

7.2.4 Mountains

Mountains are implemented to be as simple as possible, but no real mountains look like this. The mountain slope should look more like fBm, to look more like real mountains. The input to the mountain function doesn't even have to change to make this happen. The mountain can still gradually slope down using the two-dimensional Gaussian function, but multiply the height by some approximation of fBm.

7.2.5 Roads and Rivers

The roads can be improved in two significant areas. The first is the corners. It is undeniable that the roads are currently made up of triangles; it is visible when the road turns. These edges can be made more round to make it look more like a real road. The second improvement would be in how the road is generated. The midpoint displacement algorithm makes the roads more erratic than initially planned during design. Several attempts were made to change the displacement variable to make the roads look better, but in most cases, the roads either look too straight or not straight enough.

The major problem with the river implementation is the lack of merge implementation. When there are height changes in the terrain, the river implementation does not always attempt to flow downwards. If there were code implemented to make the different models of the world merge, the shape of a river generated with midpoint displacement would still look close enough to a real river.

7.2.6 Merging

The lack of merging implementation impacts every model. When several models are rendered in the same position, there is no attempt to combine them. They are rendered on top of each other. Implementing merging functionality turned out to be a lot more complicated than initially hoped when the project was started. A proper implementation would make every object in the rendered world look a lot better, and is the most exciting area for future research.

7.3 Development Process

7.3.1 Software Quality

Many software design decisions were made based on experience from a past OpenGL project. The decision to split the rendering process into a mesh, model and renderer objects came as a result of changing the code in several places when rendering new objects in past projects. Most implementations based on prior experience lead to good results, while a few did not pan out.

7.3.2 Missing Features

Initially, the use of different shaders to change the look of the was desired. When implementing the renderer class, the idea was to have multiple renderers with different shaders loaded so that each model could be rendered by a different renderer to showcase the impact of shader code. Creating multiple renderers were tested but did not work as intended. In the test, the two renderers each had a set of models, but the last renderer ended up superseding the first renderer. Nothing from the first renderer ended up on the screen. Without a smart way to solve this problem and no desire to create a new renderer class, multiple renderers were abandoned.

In its place, the renderer class was modified to contain two shader programs within the same renderer object. This implementation was used to render the black outlines of triangles while also rendering the full texture on the triangle; one example of this feature being used can be seen in fig. 5.2. The function was mostly used during the development of models to make sure the triangles were correct. It can be hard to judge without a black outline around the textures.

Chapter 8

Conclusion

In this thesis, a program for rendering procedurally generated worlds has been specified and implemented. The project focused on the rendering of individual models, and on the end-user being able to customize what kind of world is rendered. Geometrical shapes were implemented to be used together in more complex objects. Basic mountains and paths were implemented to give the terrain height variety, rivers, and roads.

The project showed how basic geometrical shapes could be used to build more complex models like buildings and trees. And by structuring the models in a grid, a different terrain representation can be made as well. The texturing of the models is easy to change and can be used to give the rendered world a great variety.

The generation of roads and rivers with the midpoint displacement algorithm gave decent results. When small towns were used as start and endpoints, the world became more connected.

The project shows great promise to be used as a foundation for future work. More studies can be done on the design level, in how textures are implemented and how the shaders of OpenGL are used. On the implementation level, more models can be implemented, and further functionality to manipulate the models' orientation and dimensions can be explored.

Merging the different parts of the terrain is the most exciting area for future research. It would give the rendered world a lot more variety and make the result look a lot more realistic.

Bibliography

[1] E. P. C. F. Brenner P.de Castro, Rosilane R. da Mota, “Level design on rogue-like games: An analysis of crypt of the necrodancer and shattered planet,” *SBC – Proceedings of SBGames*.

[2] M. S. Stichting Mathematisch Centrum. Nethack. [Online]. Available: https://www.nethack.org/

[3] M. J. N. Noor Shaker, Julian Togelius, *Procedural Content Generation in Games*. Springer, 2016.

[4] J. V. D. V. A. I. Mark Hendrikx, Sebastiaan Meijer, “Procedural content generation for games: A survey,” *ACM Transactions on Multimedia Computing Communications and Applications*.

[5] K. O. S. C. B. Julian Togelius, Georgios N. Yannakakis, “Search-based procedural content generation: A taxonomy and survey,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172–186, April 2011.

[6] RPGamer. The elder scrolls iv: Oblivion interview with gavin carter. [Online]. Available: https://www.webcitation.org/6872MtzO8?url=http://www.rpgamer.com/games/elderscrolls/elder4/elder4interview.html

[7] L. Stornaiuolo. Games and semantics. [Online]. Available: https://www.researchgate.net/figure/A-3D-landscape-from-The-Elder-Scrolls-IV-Oblivion_fig3_266287203

[8] 123RF. Old wood tree background texture pattern. [Online]. Available: https://fr.123rf.com/photo_17432852_old-wood-tree-background-texture-pattern.html

[9] M. Widmark. Terrain in battlefield 3: A modern , complete and scalable system. [Online]. Available: https://gdcvault.com/play/1015415/Terrain-in-Battlefield-3-A

[10] E. Games. Landscape outdoor terrain. [Online]. Available: https://docs.unrealengine.com/en-US/Engine/Landscape/index.html

[11] C. McAnlis. Halo wars: The terrain of next-gen. [Online]. Available: https://www.gdcvault.com/play/1277/HALO-WARS-The-Terrain-of

[12] U. Terrains. Ultimate terrains. [Online]. Available: https://uterrains.com/gallery/

[13] B. Mandelbrot, *The Fractal Geometry of Nature*. W. H. Freeman and Company, 1982.

[14] F. K. Musgrave, *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann, 1994.

[15] F. K. M. Manuel N. Gamito, "Procedural landscapes with overhangs," *10th Portuguese Computer Graphics Meeting*.

[16] L. C. Alain Fournier, Don Fussell, "Computer rendering of stochastic models," *Communications of the ACM*.

[17] K. Perlin, "An image synthesizer," *ACM SIGGRAPH Computer Graphics*.

[18] ——, "Improving noise," *ACM Transactions on Graphics (TOG)*, vol. 21, no. 3, pp. 681–682, July 2002.

[19] D. Saupe, *Point Evaluation of Multi-Variable Random Fractals*. Springer, 1998.

[20] S. R. Peter Norvig, *Artificial Intelligence: A Modern Approach*. Pearson, 2020.

[21] M. S. Floater. Bézier curves and surfaces. [Online]. Available: https://www.mn.uio.no/math/english/people/aca/michaelf/papers/bezier.pdf

[22] T. Mazurkevic. Noise and fractals. [Online]. Available: https://github.com/Falmouth-Games-Academy/comp250-wiki/wiki/Noise-and-fractals

[23] K. Group. Opengl overview. [Online]. Available: https://www.khronos.org/opengl/

[24] ——. History of opengl. [Online]. Available: https://www.khronos.org/opengl/wiki/History_of_OpenGL

[25] J. T. G. Olav Egeland, *Modeling and Simulation Form Automatic Control*. MARINE CYBERNETICS, 2002.

[26] 123rf. Textures. [Online]. Available: https://jp.123rf.com/

[27] M. G. Camilla Löwy. Glfw. [Online]. Available: https://www.glfw.org/

[28] Dav1dde. Glad. [Online]. Available: https://glad.dav1d.de/

[29] g truc. Opengl mathematics. [Online]. Available: https://glm.g-truc.net/0.9.9/index.html

[30] M. M. Nigel Stewart, Milan Ikits. Glew. [Online]. Available: http://glew.sourceforge.net/

[31] H. Borge. Proceduralcities. [Online]. Available: https://github.com/Howiezi/ProceduralCities

[32] S. H. Ahn. Opengl sphere. [Online]. Available: http://www.songho.ca/opengl/gl_sphere.html