Øystein Brox

Procedural Generation of Terrain and Roads

Master's thesis in Cybernetics and Robotics
Supervisor: Sverre Hendseth
June 2019

Master's thesis

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

NTNU
Norwegian University of
Science and Technology

Øystein Brox

Procedural Generation of Terrain and Roads

Master's thesis in Cybernetics and Robotics
Supervisor: Sverre Hendseth
June 2019

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

Problem Description

Procedural generation is a method of creating data algorithmically as opposed to manually. Procedural generation of terrain in the videogame industry has largely been handled by tweaking a random source in order to get a reasonable looking landscape. This thesis will instead opt to look at different aspects of the landscape, such as mountains, lakes, and cities, as “primary components”.

This should be the foundation for an experiment into an area of procedurally generated landscape rarely explored, namely the creation of roads. Roads introduce complexity both in the form of deciding the optimal path for the roadway and natural endpoints in the landscape.

The student shall:

- Research relevant procedural generation
- Specify a possible “proof of concept”
- Implement the specified “proof of concept”
- Evaluate the results based on experience gathered

Preface

This thesis is written as a part of my Master of Science degree in Cybernetics and Robotics at the Norwegian University of Science and Technology (NTNU). The work has been done during the spring of 2019 under the supervision of Sverre Hendseth. We have met almost every week to discuss the progress and what areas of research to pursue going forward. I would like to thank Sverre Hendseth for the all help and support he has given me throughout this thesis. Further, I would like to thank Christian Aschehoug and Edvard Rauø Vasdal for motivation throughout my final year of study. Lastly, I would like to thank all my friends and family, especially my parents, who have supported me throughout my education. To each and everyone, I am truly grateful.

NTNU provided me with a Dell workstation computer with windows 10 installed. The choice of game engine was made by me in the autumn as part of the specialization project. The game engine, Unity, is freely available for all non-commercial uses on their website. Some other components throughout the thesis are taken from external sources, and those will be clearly marked with citation as well as a URL. Several algorithms and parts of the code were inspired by online forums for code development, game development, and Unity's own community forums.

This thesis is inspired by ongoing research in the video game industry as well as professional interest from both myself and supervisor Sverre Hendseth.

Trondheim June 3, 2019
Øystein Brox

Abstract

This paper studied the possibilities for procedural generation of terrain and roads through the utilization of “primary components”. A proof of concept was implemented where the terrain was created through combinations of bell-curve shaped mountains. Several studies were conducted as to how to utilize this single component for creating realistic terrain. Roads were constructed through A* search based on the slope of terrain and location of cities and lakes. The implication of different cost functions and how to visualize the road was an integral part of the experiment.

Results of the experiment showed that combinations of simple bell-curves could create a realistic environment. Different methods for procedural placement of mountains were found feasible and could be used to great effect on different applications of procedural generation. Further, the use of cities and lakes as components meant the road could travel on a natural path from one city to another even if the entire terrain were procedurally generated. The results showed great promise for this type of procedural generation. The introduction and combination of components made a realistic environment and could create the foundation for significant steps forward in the industry.

Sammendrag

Denne oppgaven undersøker mulighetene for prosessuell generering av terreng og veier gjennom å utnytte "primære komponenter". En *"proof of concept"* er implementert hvor terrenget består av kombinasjoner av bellkurve formede fjell. Flere studier er utført på hvordan bruke denne ene komponenten til å skape et realistisk landskap. Veier er laget gjennom bruk av A* søk basert på helninger i terrenget og posisjonen til byer og innsjøer. Implikasjonene av forskjellig kostfunksjoner og hvordan visualisere veien var viktige deler av eksperimentet.

Resultatene fra eksperimentet viser at kombinasjonen av slike enkle bellkurver kan skape et realistisk landskap. Forskjellige metoder for prosessuell plassering av fjell viste seg å være gjennomførbare og kan brukes med god effekt på ulike applikasjoner av prosessuell generering. Videre førte bruken av byer og innsjøer som komponenter til at en vei kunne strekke seg fra en by til en annen selv om landskapet var generert prosessuelt. Resultatene viser gode utsikter for slik type prosessuell generering. Introduksjonen og kombinasjonen av komponenter førte til ett realistisk landskap som kan danne grunnlaget for store steg videre.

Table of Contents

Problem Description I

Preface III

Abstract V

Sammendrag VII

1 Introduction 1
1.1 Motivation and Objective . . . 1
1.2 Contributions . . . 2
1.3 Report Outline . . . 3

2 Background 5
2.1 Procedural Generation in the Video Game Industry . . . 5
2.1.1 Procedural Terrain Generation . . . 6
2.1.2 Procedural Road Generation . . . 7
2.2 Basics of 3D Graphics . . . 8
2.2.1 Meshes . . . 8
2.2.2 Visualization . . . 10
2.2.3 Triplanar Texture Mapping . . . 11
2.3 Mathematics . . . 14
2.3.1 Transformation Matrix . . . 14
2.3.2 Linear Interpolation . . . 14
2.3.3 Bézier curve . . . 15
2.3.4 Radial Basis Functions . . . 17
2.4 Algorithms and Data Structures . . . 18
2.4.1 Dijkstra's algorithm . . . 18
2.4.2 A* . . . 19
2.4.3 HPA* . . . 20
2.4.4 D* . . . 23

2.4.5 Distances 23
2.4.6 Heap 25
2.5 Noise 26
2.5.1 Perlin Noise 26
2.5.2 Blue-Noise Sampling 26
2.6 Unity Features 29
2.7 The Specialization Project 30
2.7.1 Meshes 31
2.7.2 Mountains 32

3 Specification of the Experiment **35**
3.1 Specification 35
3.2 Summary of the specification 37

4 Implementation and Results **39**
4.1 Improvements on the Pre-Existing System 39
4.2 Mountains 40
4.2.1 Mountains on Top of Existing Mountains 40
4.2.2 Mountain Ranges 43
4.2.3 Natural Unevenness Through Mountains 47
4.3 Other Components 48
4.3.1 Lakes 49
4.3.2 Cities 50
4.4 Roads 52
4.4.1 Choice of Pathfinding Algorithm 52
4.4.2 The Implementation of Roads 53
4.4.3 The Impact of the Cost Function 57
4.4.4 Saving or Replicating the Roads 61
4.4.5 Visualization of the Road 62
4.5 Player Character 66
4.6 Texturing of the Landscape 68

5 Evaluation **71**
5.1 Mountains 71
5.1.1 Additional Mountains 71
5.1.2 Mountain Ranges 72
5.1.3 Limits of Bell-Curves 73
5.2 Roads 74
5.2.1 Use of a Traditional Search Algorithm 74
5.2.2 Possible Improvements to the Current Search 75
5.2.3 Replication of Roads 76
5.2.4 Improved Visual Appearance 76
5.3 The Entire Experiment 77
5.3.1 Limited Number of Components 77
5.3.2 Procedural Placement of Components 78

6 Discussion **81**
6.1 Design of the Specification 81
6.2 Dedicated Time to Support-Components 82
6.3 Retrospective on the Experiment 83

7 Conclusion and Future Work **85**
7.1 Conclusion 85
7.2 Future Work 86

Bibliography **87**

Chapter 1

Introduction

1.1 Motivation and Objective

Consumers in the US spent almost 30 billion dollars on video games in 2017 alone. 65.000 people are employed in the over 2.700 game companies across America [1]. The industry is growing rapidly, and there is an ever-increasing demand for developers and designer to create new and exciting games. Simultaneously manual creation of content is becoming a significant bottleneck during development[2, 3].

Procedural content generation, PCG, is algorithmic creation of content. While this definition could span wider, all content discussed in this thesis centers around video games. Different variations of PCG been part of the game industry since the 1980s; despite this, the majority of content is still created manually. There are many different explanations as to the limited use of algorithmic creation of content; the primary reason might be its inherent difficulty.

Manually crafted content is, by comparison, easy to quality assure. A manually created world could be inspected, small changes to the appearance could be done and as such perfect the result. Such iterative improvements are impossible with a procedurally generated world. A small alteration of the algorithm could have a massive impact on the landscape. Also, the task of deciphering what part of the algorithm created the unwanted part of the world is non-trivial.

Additionally, generated content is always supposed to fulfill the required quality standard. Because of that, developers have to ensure that in all circumstances, the content generated is realistic and within their vision of the game. Further, many methods of procedural generation leave little room for determinism or domain knowledge after its creation. Traditional noise based landscape generation creates vast, varied worlds with both mountains and lakes; however, the game has no recorded information of where they are located. As

a result, the creation of roads or paths is difficult because the game has few ways of determining where points of interests are located.

This thesis will attempt to tackle the problem of procedurally generate a road in previously generated terrain. A specification for a proof of concept demonstrating such abilities will be presented and implemented. The experiments functions both as a study on the feasibility of such generation and a study on different aspects of the procedure.

1.2 Contributions

There is much research being done within the field of procedural generation. To clearly state the contributions of this thesis, the most important contributions are listed below.

Implementation of a procedurally generated terrain based on components: Every significant aspect of the game world presented in this thesis is based on a few selected components. The entire experiment and results are in themselves a feasibility study on the use of components as opposed to traditional noise.

Creation of mountain ranges: A study on how to combine mountains to create mountain ranges procedurally with the increased domain knowledge of components. Different approaches are implemented, compared, and evaluated on their feasibility and what sort of applications the different methods could be used for.

Creation of roads based on previously generated terrain: The increased knowledge of the world, since all features in the landscape are created from a set of primary components, is utilized in the creation of roads. A search algorithm is implemented and optimized for the creation of roads while the game is running. This function additionally functioned as a more in-depth study on how to generate and visualize a realistic road.

Evaluation of the results and suggested improvements: The proof of concept is evaluated based on what worked and areas sill open for improvements. Several ideas and prospects for future research and possible improvements to the current implementation are presented.

1.3 Report Outline

This thesis is composed of seven chapters that are structured as follows:

Chapter 1 contains a brief introduction and motivation for the topic, as well an overview of the most important contributions of this thesis.

Chapter 2 presents the necessary background for the features used in this experiment. This includes a literature review, basic information of game development as well as several algorithms either studied or implemented later. Unique features of Unity, as well as a short summary of the specialization project this thesis is built upon, is presented.

Chapter 3 contains the specification for the proof of concept implemented.

Chapter 4 explains the implementation in more detail as well as presents all the results gathered during the experiment.

Chapter 5 evaluates the results from the experiment. Several improvements and suggestions for further research on the topic are presented.

Chapter 6 discusses the experiment, proof of concept, and decisions made during the experiment.

Chapter 7 concludes the study as well as suggests some areas of research going forward.

Chapter 2

Background

2.1 Procedural Generation in the Video Game Industry

There has been done extensive research on several different applications of procedural content generation; however, by far, most research is connected to the video game industry. Within a single game, there are multiple aspects possible to generate procedurally. This section will first give a general overview of all the different aspects of a video game that could be generated procedurally. Next, different approaches to the generation of terrain and roads will be presented.

A taxonomy strategy presented by Hendrikx et al.[2] is to classify the content into five different categories. They could be structured as a pyramid with foundational building blocks at the bottom and more overarching abstract content at the top. The categories are as follows.

At the foundation is the *Game Bits*, this is the basic building blocks of a procedurally generated game. This includes the likes of textures, sound, vegetation, and similar foundational components.

The next level is the *Game space*. This entails both indoor and outdoor maps as well as bodies of water. Here the previous Game bits, some of which could be generated procedurally, have partially filled up a game world for the player to explore.

In a similar fashion, *Game systems* are the next level. This encompasses more abstract creations; such as simulation of a city or the relationship between vegetation and parts of the outdoor map.

Game scenarios are the next level of the pyramid. This part is often transparent to the player but dramatically affects how the game is played. This could, for instance, be generating new quests or puzzles for the player to solve.

The top layer is *Game design*. This layer encapsulates abstract features such as in-game rules or goals for the player.

Within all these different classes of content, several areas of research are performed. Nonetheless, some areas are more commonly used in video games than others. The area of procedural generation that has been explored the most is within the game space, namely map generation. In this context, *map* refers to the playable surface of the game. As such, there has been developed a lot of different methods for procedural terrain generation and of which some strategies will be explored in the next section.

2.1.1 Procedural Terrain Generation

The exact details on how the most popular games with procedurally generated content are created are impossible due to the secrecy that surrounds all major games. Nonetheless, for some of the games, the general principles of their design are known. Subsequently, this allows for publications and studies into their approaches. This allows their overarching design principle to be studied further.

There are several methods of procedurally generating terrain in video games. However, the most common methods are based on noise [4]. There are several noise-based algorithms, and they are used differently based on the demand of the application. The Simplex noise is especially suited for higher dimensions, while the Value noise is a fast and straightforward approach in only two dimensions. Perlin Noise is also a method with widespread use in the industry. This is a form of gradient noise that ensures smoother transitions from high to low values. For this exact reason, it is widely used for generating terrain and other applications such as simulating fire or clouds. The game Minecraft utilizes modified Perlin Noise in order to generate the vast open terrain seen in the game [4].

Taken a step further, approaches based on fractals, with different types of noise for each octave, have been put to use in some applications. This process creates several layers of finer detail and breaks up the unnatural smoothness a single iteration of Perlin Noise would generate. However, problems with computation time arise when there are several layers of recursion. As such real time of implementation of fractals in video games is very difficult [5].

On the other hand, lightweight approaches to procedural terrain in video games have also been implemented. In games such as Diablo III, a set of predefined tiles are shuffled and as such randomly generates a map for the player to explore [4]. The tiles themselves are handcrafted ahead of time, and naturally puts a theoretical limit on the number of possibilities, but with enough tiles, the chance of the exact same game world is minimal.

Research into entirely different methods of procedural terrain generation is also being attempted. While there is too many to mention all here, examples include the likes of evolutionary algorithms [6], cellular automats [7] and even deep learning based on satellite images from NASA [8]. All of those are academically interesting, but far from considered an industry standard.

2.1.2 Procedural Road Generation

The public knowledge on how roads are created in popular video games with procedural generation also suffers from their need for secrecy. This, in combination with most procedurally generated games simply avoid incorporating roads, makes the available material is somewhat limited.

However, there are still several academic researchers in the field that have published articles related to procedural generation of roads. A popular city generation tool based on procedural generation, called *Citygen*[9], has procedural generation of roads at its core. The basic concept, as presented in their paper *Citygen: An Interactive System for Procedural City Generation*[10], makes the city revolves around the roads. First, a set of primary roads are generated, then the surrounding is filled with secondary roads. This process creates different city blocks. Each road consist of a set of control nodes the path should attempt to follow, and a more detailed set of nodes that actually depicts the road. The latter path is based both on the aforementioned control nodes as well as other aspects such as the environment. The Citygen software also consists in large part user interface for the designer to augment and alter the road structure created through procedural generation. As such, it serves primarily as a foundation for the designer to improve upon as opposed to a final design.

This area of procedural road generation is by far the most common area of research. Either as with the aforementioned *Citygen* or as [11] to procedurally generate the road networks of a modern city or as is the approach in [12] where they attempt to model the road networks found in settlements in South Africa. They all share the property of trying to achieve complex and realistic networks of roads but put minimal efforts into generating the roads with respect to the underlying terrain. When the network of roads is created, the surrounding landscape is augmented to fit this structure.

A study that takes the approach of attempting to generate a single road based on the underlying terrain is [13]. In this experiment, they generated a road based on a weighted anisotropic shortest path algorithm from one point in the landscape to another. This was done in a continuous scene with the cost function taking into accounts aspects, including the slope of the terrain and obstacles such as mountains, lakes, rivers, and forests. In order to overcome the difficulties with computing an exact solution to the weighted anisotropic shortest path, uniform discretization into a grid was performed. A* is utilized on this grid-based graph in order to search for the best path. The paper also presents several other fundamental properties in order to create a more natural appearance of the path. First, in order to increase the number of discrete directions, the path is able to take, path segment masks were introduced where the algorithm could move several tiles at a time in order to incorporate more discrete directions. Further, they also considered curvature by creating a 3-dimensional discretization of the continuous domain in order to represent all the different positions with all the different orientations.

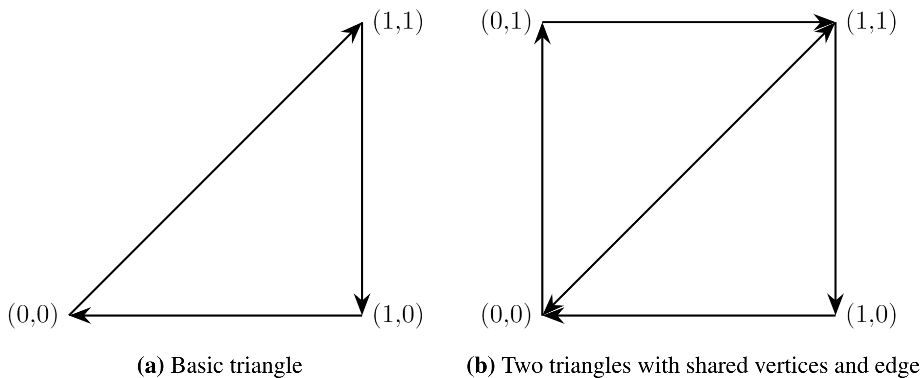
Finally, a road mesh is created based on clothoid splines, and the surrounding terrain mesh is altered slightly for the road mesh to fit on top of it.

2.2 Basics of 3D Graphics

2.2.1 Meshes

Theory from [14] with some alterations to better fit the purpose of this thesis.

Most 3D graphics consists of a series of vertices connected into different shapes referred to as polygons. These polygons are most commonly shaped like triangles or at least broken into a set of triangles. There are historically good reasons for most computers being optimized for graphics computation with triangles. The main reasoning behind the choice of triangles is that three vertices are the least amount that is required to span a plane in 3D space unambiguously. The vertices are connected clockwise orientation, as shown in figure 2.1a, but more about the reasoning behind this in section 2.2.2. Set of such polygons are combined into a *Polygon mesh*, also referred to as a *wireframe mesh* or simply *mesh*.



(a) Basic triangle **(b)** Two triangles with shared vertices and edge

Figure 2.1: Basic polygons used in video games

As with all digital images, mesh figures have different resolutions. Instead of pixels, the resolution is determined based on the number of vertices. With few vertices and long edges between them, the model ends up looking *"bulky"* and not true to the smooth surface it tries to represent. However, the rendering of a mesh with fewer vertices is significantly faster, and such the choice of resolution is an important aspect when optimizing a game. A comparison of different resolutions is shown in figure 2.2. This image is created by *Gabro Media*[15], and depicts two barrels with different resolution, i.e., the number of vertices.

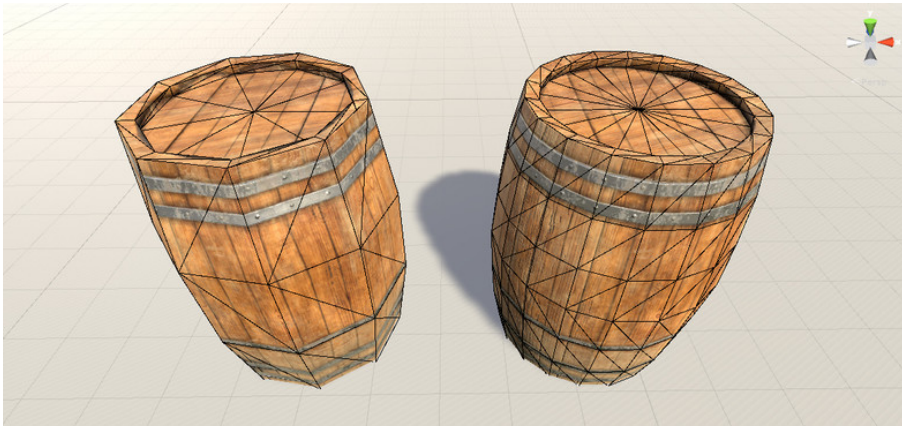


Figure 2.2: Two barrels with different resolution. Image from [15]

Meshes visualizing terrain faces a particular challenge; that they are huge. As most game engines render the entire mesh at once, and stores it in memory, game developers were faced with a problem. There was an apparent demand for relatively high-resolution terrain, but at the same time still be enough processing power and memory left for the rest of the game.

The solution that was crafted was to split large objects into several meshes. If these meshes are aligned correctly, the landscape will appear continues from the players' point of view but will be built with several different meshes. This is shown in figure 2.3, where the blue and red mesh are placed adjacent to each other and as such *"sewn"* together. A similar process will be done later in the experiment. Note that along the seam, marked with violet in the figure, both meshes has vertices and edges directly on top of each other. They do not share vertices or edges as is often done within the same mesh.

Also, by splitting the game world into different "chunks" new possibility emerges. As each chunk of the terrain is split into different meshes, it is possible to alter them independently. This allows for different parts of the landscape to be drawn with different resolution quite easily.

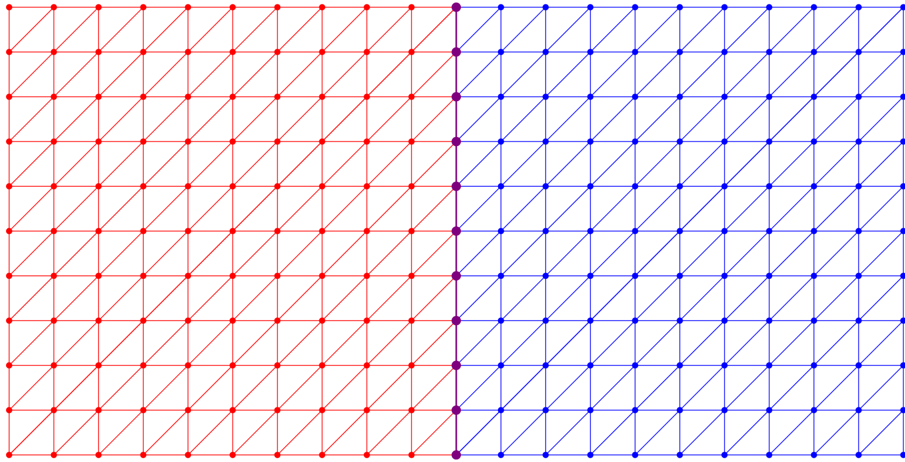


Figure 2.3: Two meshes sewn together

2.2.2 Visualization

Theory from [14] with some alterations to better fit the purpose of this thesis.

The triangular polygons described in the previous section have both a front and back. This is of great importance for visualization. This is determined based upon the rotation, i.e., the order vertices are listed in each triangle. The vertices in figure 2.1a are connected clockwise and as a result the polygon *faces* the reader. This distinction is important as a majority of modern game engines are optimized to not focus on the backside. This results in the mesh either not having a texture or more commonly appearing transparent if observed from the other side.

For a more realistic environment, a *texture* is applied to the surface of a wireframe-model. A texture is in its simplest form a single color. However, a single color is not very realistic, and as a result, more advanced techniques are used to make a game come to life. There are multiple techniques and methods for achieving a realistic environment, and it would be a thesis in its own exploring the different approaches. The most common way is to have a basic foundation texture as an image, for instance, a brick wall. Then create a new texture based on the same image to highlight where it is supposed to be elevation, and similarly to indicate how light is supposed to be reflected.

All of these textures are then used in combination with a *shader*. A shader is basically a unique type of script that makes the textures work as intended. It combines the textures and assigns them the correct properties to look seamless. The shader also handles a lot of how and where light is reflected and where to cast different shadows. All of this is what makes a 2-dimensional mesh surface look 3-dimensional when playing a game.

This is observable in figure 2.4. Created with Unity's standard asset textures and shader used for grassland. The mesh the person is standing on is perfectly flat. However, with the use of clever texturing, the grass appears to be three dimensional and gives a feel of resembling nature.

Figure 2.4: Flat grass mesh with a 3-dimensional appearance

2.2.3 Triplanar Texture Mapping

When applying an image texture onto a mesh, it has to be projected onto it from a given direction. With most surfaces, this process is straightforward as there is a clear direction in which to apply the texture. For a mostly flat mesh, like a lawn, a projection down from above would be natural. This is how the grass texture in figure 2.4 is applied. However, with more complex terrain, this one directional application of textures is problematic. For instance, in the case of a mountain, the image would be *dragged* along the mountainside and look far from nature. This exact scenario is shown in figure 2.5

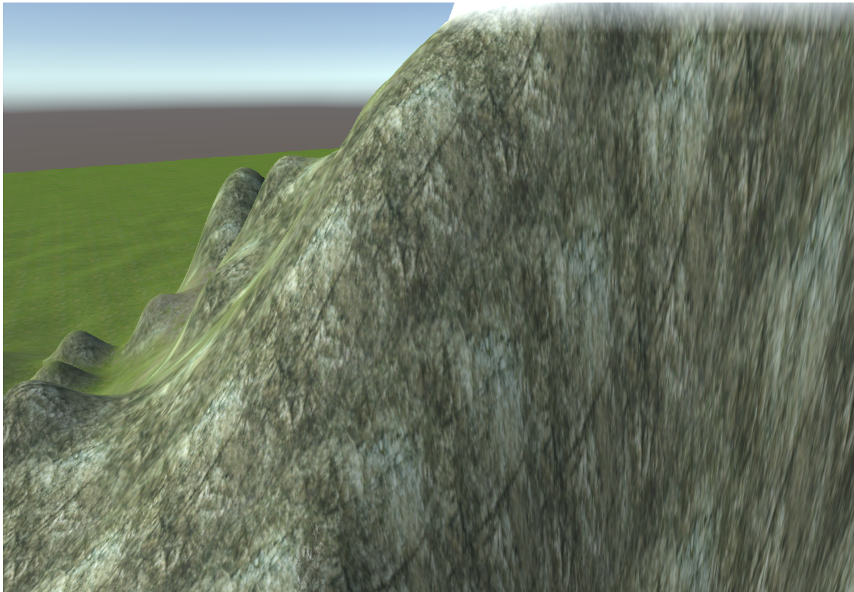


Figure 2.5: Mountain with texture applied in one direction

A popular strategy to levitate this problem is to always apply the same texture from all three directions (x, y, and z). This method is called tri-planar texturing, as it applies a texture in three directions only with different intensities. In several other applications, different textures could also be applied in different directions. However, as this will not be attempted in this thesis, no further emphasis will be put on this possibility. The concept of tri-planar texturing is also shown in figure 2.6.

Such texturing ensures that all sides of an object is correctly textured and has a natural appearance. This is utilized in figure 2.7 on the same mountain as one dimensional texturing was used on in figure 2.5 for comparison.

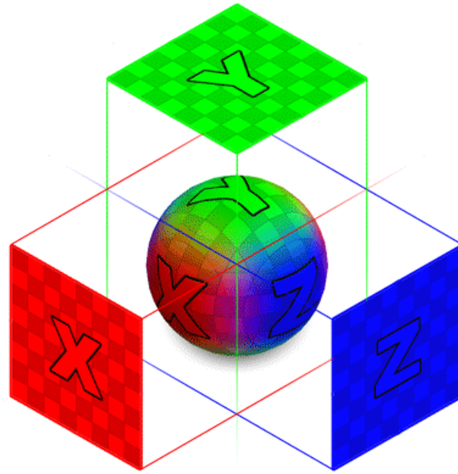


Figure 2.6: Principle behind triplanar texturing, image from [16]

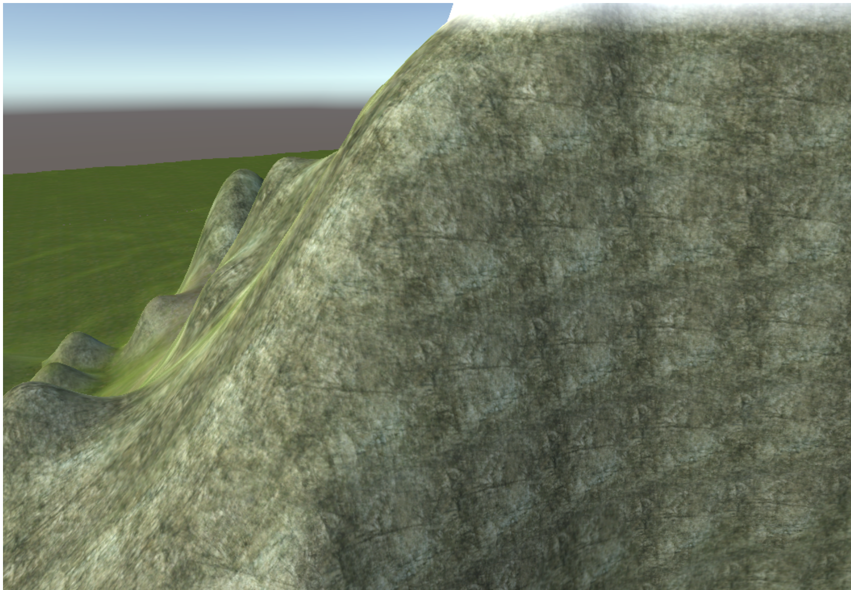


Figure 2.7: Triplanar texturing on a mountain

2.3 Mathematics

2.3.1 Transformation Matrix

Theroy based on [17].

Transforming from one frame to another could be done through a *Transformation Matrix*. The general homogeneous transformation matrix is given in equation 2.1, from [17] equation 6.115.

$$\mathbf{T}_b^a = \begin{bmatrix} \mathbf{R}_b^a & \mathbf{r}_{ab}^a \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{2.1}$$

The $\mathbf{R}_b^a$ element is in itself a matrix. Depending on the axis of rotation, the matrix is different. However, as all rotations will be done in two dimensions in this experiment, the matrix is simplified. This simplified rotation matrix is given in equation 2.2.

$$\mathbf{R}(\psi) = \begin{bmatrix} \cos\psi & -\sin\psi \\ \sin\psi & \cos\psi \end{bmatrix} \tag{2.2}$$

The $\mathbf{r}_{ab}^a$ term in equation 2.1 refers to the translation. As with the rotation, this will only be performed in two dimensions, and as such, the matrix is somewhat simplified. The matrix is presented in equation 2.3, where each term is the translation in the respective direction.

$$\mathbf{r}_{ab}^a = \begin{bmatrix} t_x \\ t_y \end{bmatrix} \tag{2.3}$$

2.3.2 Linear Interpolation

Interpolation is used in order to generate new data points within a specified range. For the instances used in this experiment, linear interpolation will be used between two discrete points. The formula for a given point (x, y) between (x_1, y_1) and (x_2, y_2) is shown below in equation 2.4. Linear interpolation is commonly abbreviated as *lerp*.

$$\frac{y - y_1}{x - x_1} = \frac{y_2 - y_1}{x_2 - x_1} \tag{2.4}$$

However, for many applications, including in this implementation, the exact point (x, y) is not known. Instead, there is a desire to find those coordinates based on the relative position between two points. This is the basis of how most mathematical libraries in programming languages incorporate lerp-functions. The t-term represents the percentage between the two points as a decimal in the range $[0, 1]$. As such naturally a t-value of 0 indicates $(x, y) = (x_1, y_1)$ and likewise $t = 1$ results in the endpoint for the possible range.

$$(x, y) = (1 - t) \cdot (x_1, y_1) + t \cdot (x_2, y_2) \tag{2.5}$$

2.3.3 Bézier curve

Theroy based on [18].

Bézier curves are in its essence curves created based on linear interpolation. In mathematical terms, they are polynomials of *t*, where *t* is fixed between 0 and 1(as in linear interpolation). A Bézier curve of order $n = 1$ is referred to as linear, $n = 2$ is quadratic and so forth. The exact formula of Bézier curves up to cubic curves $(n = 3)$ is shown in equation 2.6.

$$B(t) = (1 - t)P_0 + tP_1 \tag{2.6a}$$

$$B(t) = (1 - t)^2 P_0 + 2(1 - t)tP_1 + t^2 P_2 \tag{2.6b}$$

$$B(t) = (1 - t)^3 P_0 + 3(1 - t)^2 tP_1 + 3(1 - t)t^2 P_2 + t^3 P_3 tP_1 \tag{2.6c}$$

In theory Bézier curves could be extended to *n*-dimensions with the general formula given in equation 2.7. However, for this thesis, only cubic curves are utilized, and as such, the examples given are centered around those.

The equation for a linear Bezier curve, equation 2.6a, is exactly the same as the formula for linear interpolation, equation 2.5. Further, the quadratic curve is simply a linear interpolation between the linear interpolation between two and two points. This feature is extremely useful when implementing the Bézier functions in a program as it allows the usage of recursion.

$$B(t) = \sum_{i=0}^{n} \binom{n}{i} (1 - t)^{n-i} t^i P_i \tag{2.7}$$

In general, the Bézier curve will not pass through all its points, probably only the first and last point. As such, these points are sometimes referred to as *anchor points*, while the rest are referred to as the *control points* of the curve, see figure 2.8. In other words, the anchor points set the start and stop point, while the control points control the curvature. Examples of different curves based on the positions of anchor and control points are shown in figure 2.9.

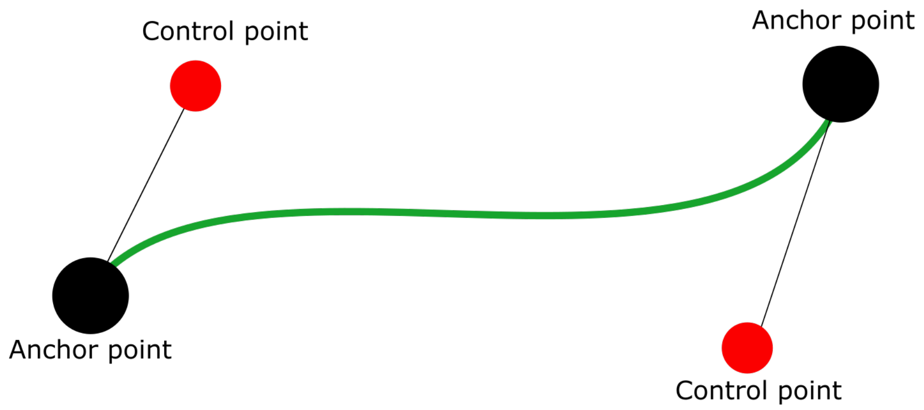


Figure 2.8: The basic principle of control and anchor points in Bézier curves

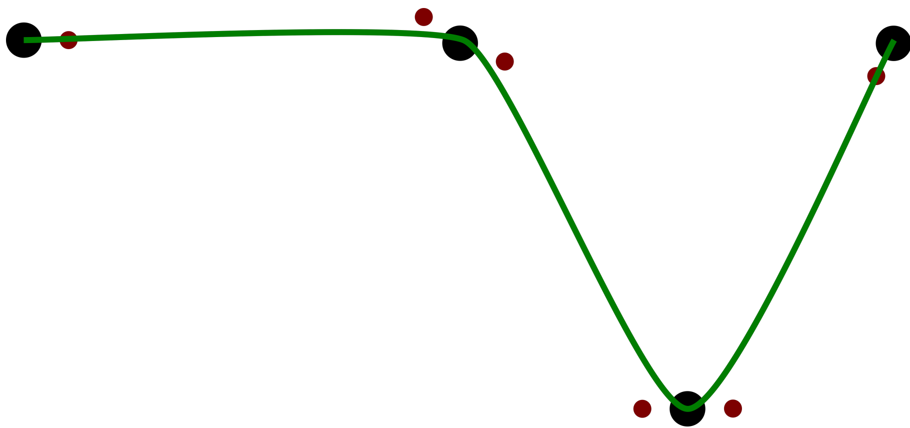


Figure 2.9: Path based on Bézier curves

2.3.4 Radial Basis Functions

Theroy based on [19].

Radial Basis Function, RBF, methods are used to approximate multivariate functions, typically through interpolation. There are many different implementations; however, for this experiment, only the Gaussian RBF will be used, and as such only, this case will be explored.

$$\varphi(P_0, P_1) = e^{-(\epsilon r)^2} \tag{2.8}$$

The Gaussian RBF takes two points and returns a value purely based on the Euclidean distance between them. The mathematical function is shown in equation 2.8. The distance between the points are calculated as the Euclidian distance $r = ||P_1 - P_0||$, but as mentioned earlier other distances could be utilized. The ϵ is known as the *shape parameter* and is used to scale the input of the radial kernel. This is illustrated in figure 2.10 where the Gaussian RBF with different ϵ parameter is plotted.

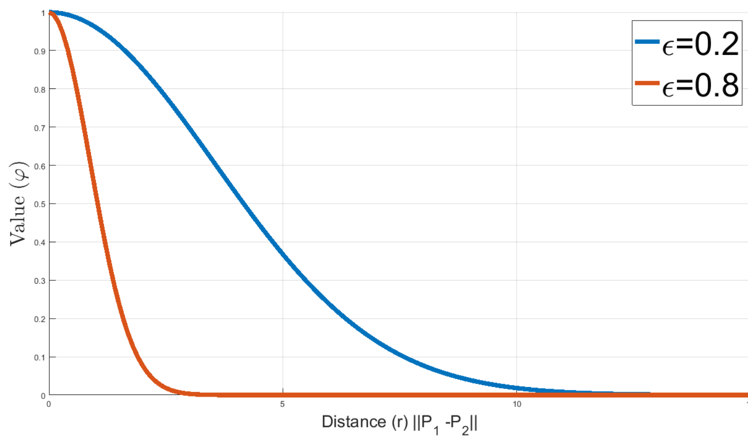


Figure 2.10: Radial Basis Function plotted with different ϵ parameter

2.4 Algorithms and Data Structures

2.4.1 Dijkstra's algorithm

Theroy based on [20].

One of the most famous search algorithms is Dijkstra's algorithm. The algorithms solve the single source shortest-path on a weighted graph problem.

Algorithm 1: Dijkstra's algorithm

```
   Input: Graph, Weights, Source
   Result: Solution or failure
 1 foreach Vertex v in Graph do
 2 |   distance[v] = Infinity
 3 |   previous[v] = undefined
 4 distance[Source] = 0
 5 Q = Set of all nodes in Graph
 6 while Q not empty do
 7 |   u = node in Q with lowest distance
 8 |   foreach neighbour v of u do
 9 |   |   Relax(u,v,w)
```

The process starts by generating a set of all possible nodes in the graph and sets their distance to infinity. Then while there are still node in the set Q, the algorithm loops through each of the neighbors of the current node and attempt to *relax* them. This is, in essence, check whether the distance to the node is shorter through the current node that the existing shortest cost to the node. This process is repeated for all nodes in the graph until the set of open nodes, Q, is empty. At this time the algorithm has complete knowledge, and the process of back tracing the *cheapest* route from the destination to the source node is trivial.

2.4.2 A*

Theroy based on [21].

The A* algorithm is, according to [21], the most widely known form of **best-first** search. It is based on a variant of Dijkstra's algorithm named *Uniform Cost Search*[21]. The major difference this introduces is to not create the entire open set, Q, at the start. Instead, it opts to add to the list when it discovers new nodes in the graph. This results in an algorithm that requires less memory and is able to find a path through larger graphs.

Algorithm 2: A* algorithm

```
   Input: Graph, Source, Goal
   Result: Solution or failure
 1 Initilize StartNode, OpenSet and ClosedSet
 2 while OpenSet not empty do
 3     Current = OpenSet.pop()
 4     if Current == GoalNode then
 5         return Solution(Current)
 6     foreach neighbour of Current do
 7         tentativeCost = Current.Cost + MovementCost(Current,neighbour)
 8         if neighbour not in ClosedSet OR tentativeCost ¡ neigbour.cost then
 9             neighbour.parent = current
10             neighbour.cost = tentativeCost
11             neighbour.Fcost = cost + heuristicEstimate(neighbour, Goal)
```

What makes A* a *best-first* search is the use of a *heuristic*. This heuristic is an estimate of the cost for reaching the goal, and as such, the algorithm does not have to explore the entire graph, but only the parts it deems necessary. As long as the heuristic is consistent, A* is ensured to be optimal on a graph.

The cost function in conventional A* graph search is presented in equation 2.9. The $g(n)$ term represents the cost of getting to the node in question from the start, the same as with Dijkstra's Algorithm. The $h(n)$ term is the heuristic estimate for the cost from the current node and until the goal. There are several metrics for estimating or calculating distance as is elaborated upon in section 2.4.5.

$$f(n) = g(n) + h(n) \tag{2.9}$$

2.4.3 HPA*

Theroy based on [22].

Hierarchical Path-Finding A*, or simply HPA*, is a proposed optimization to A* for faster computation on larger graphs. The key feature of HPA* is that large parts of the actual search are done ahead of time and as such is able to find a solution up to 10 times faster when needed [22].

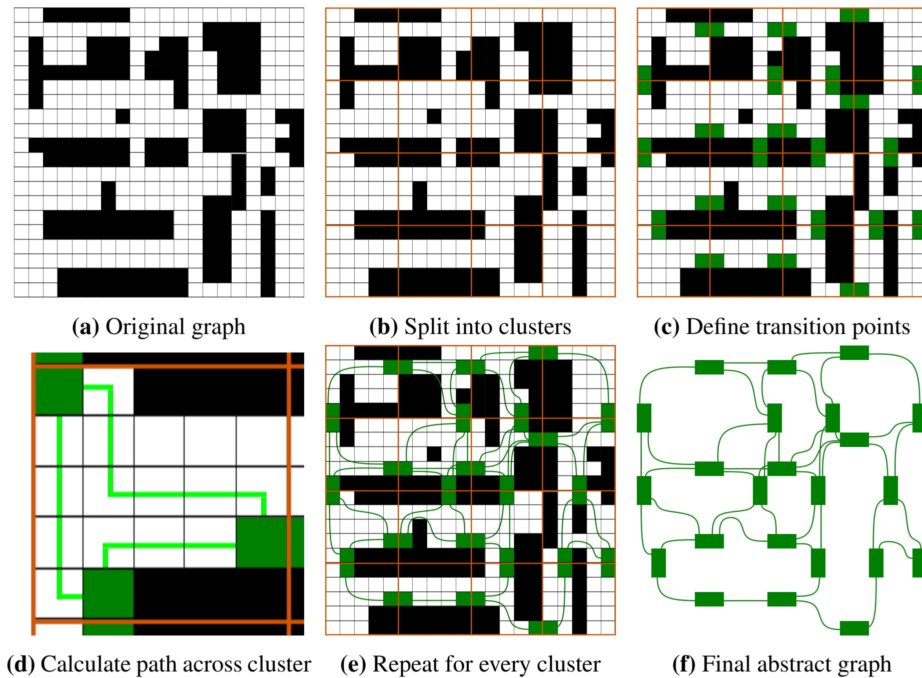
Algorithm 3: HPA* algorithms pre-processing

Input: Graph
Result: Abstract graph

1 Split map into smaller clusters
2 Identify entrances between neighbouring clusters
3 Compute traversal costs within each cluster
4 Put everrything together into an abstract graph

The pre-processing part of HPA* is done ahead of time, often in computer games when the map is created and as such way before any path planning is needed. The process is straightforward on a high level of abstraction. The first step is to split the map into smaller clusters. In theory, there could be several hierarchical layers and as such different size clusters, however, for simplicity, only two levels are exemplified here. Then entrances between them are set up: For large entrance areas, there could be implemented several transition points.

The next step is to calculate the traversal costs between all transition point within the same cluster. This is done with the use of conventional A* pathfinding, as explained earlier. When all costs across each cluster are computed, the graph could be put together once more. This new abstract graph consists of all the transition points and the costs of traveling between them. The entire process is illustrated in figure 2.11 for a simple graph.



(a) Original graph **(b)** Split into clusters **(c)** Define transition points

(d) Calculate path across cluster **(e)** Repeat for every cluster **(f)** Final abstract graph

Figure 2.11: Pre-processing part of HPA*

Algorithm 4: HPA* algorithms run-time processing

Input: Abstract Graph, Start, Goal
Result: Solution or failure

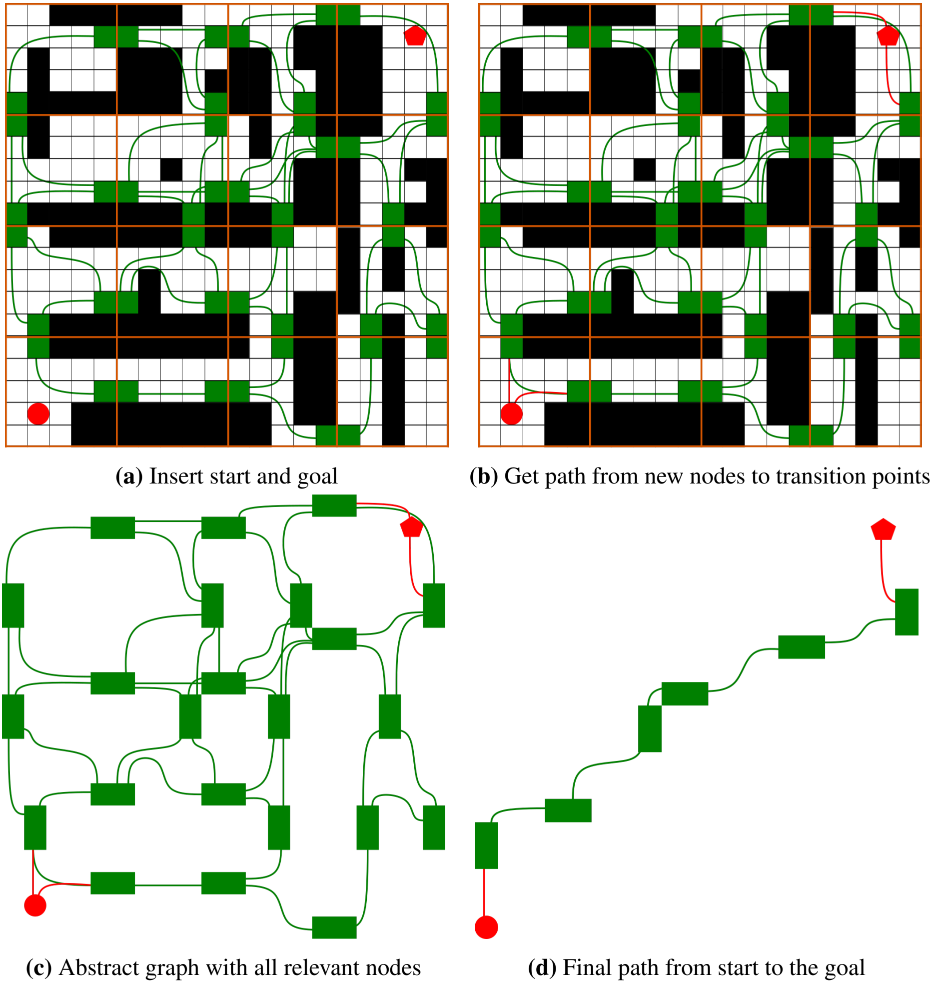
1 Insert start and goal node into the abstract graph
2 Find cost from start/goal to the entrances of their cluster
3 Calculate total costs based on previously computed costs through intermediate clusters
4 Refine steps in each cluster if necessary

When a new path from a start node to a goal node is in demand, the run-time part of the algorithm is called. First, the goal and start node are inserted into the graph created earlier. With the use of traditional A* the cost of traveling from those two nodes to the entrances in their respective cluster is computed. When this is done a simple graph spanning only a few nodes, including both the start and goal is created. The process of finding the optimal path in this graph becomes trivial, and as such, the path across the entire original map is also found.

Depending on the implementation of the algorithm, the exact path across each cluster might have to be refined. As the used transition points are known the usage of A* between only the necessary transition points is quite fast. This tradeoff between execution speed

and demand for memory is a design choice that has to be made for each use case. Regardless whether this refinement has to be computed or not, the HPA* is significantly faster than traditional A*.

As a natural consequence of utilizing a discrete subset of transition points between the clusters, a solution is not guaranteed to be optimal. According to [22], the paths that are computed with this algorithm is still within 1% of the optimal path. For many situations, this is well within an acceptable level.



(a) Insert start and goal

(b) Get path from new nodes to transition points

(c) Abstract graph with all relevant nodes

(d) Final path from start to the goal

Figure 2.12: Run-time part of HPA*

2.4.4 D*

Theroy based on [23] and [24].

Pathfinding with D* is common in both the video-game industry as well as in several robotic applications. There are actually three different versions of D* available, the original D*, Focused D* and D* Lite. While they all share the same property of being an incremental search algorithm based on A*, D* Lite is the most modern and commonly used today. Even the creators of the other two versions commonly use D* Lite [25]. Even though most properties described in this section are shared between all implementations, the focus will be with the D* Lite alternative. The major advantage of the D* algorithm, as opposed to the previous alternatives, is that it does not require complete knowledge to start the search. D* is able to start planning a route and improve it during its execution.

Algorithm 5: D* Lite algorithm

```
Input: Graph, Source, Goal
Result: Solution or failure
1 Calculate possible path based on LPA*
2 Agent starts to move along path
3 while Agent not at goal node do
4     Scan graph for changes in edge cost
5     if Edge cost changed then
6         Update all edge costs
7         Update all vertices
8         Recaclculate shortest path
```

The D* algorithm starts of using *Lifelong Planning A** to calculate a path to the goal based on the current knowledge of the graph. Then while the agent moves along this path, it scans for updates. This could, for instance, be new obstacles that were unknown with the first execution of LPA*. If some changes are detected the weights and costs are updated and the shortest path is recalculated based on the new knowledge. Here LPA* utilizes the already computed path when computing the new path, and as a result, this is significantly faster than to perform an A* search from the start. This feature makes D* excellent for pathfinding when part of the graph is either unknown or prone to changes.

2.4.5 Distances

In all the aforementioned search algorithms, the term *distance* has been used. However, there are different methods of measuring a distance in a grid. One popular method of calculating distance is through the *Manhattan Distance*, also known as *Taxicab geometry*. The formula for this distance metric is shown in equation 2.10. Examples of this distance

function implemented are shown in figure 2.13. An important aspect to remember is the lack of diagonal movement.

$$d(p,q) = |q_x - p_x| + |q_y - p_y| \tag{2.10}$$

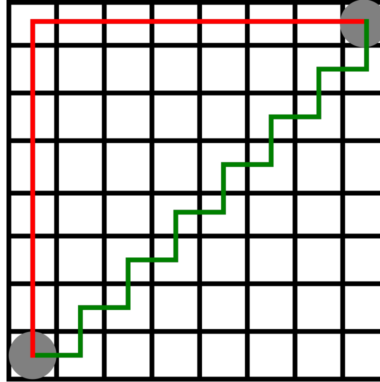


Figure 2.13: Manhattan distance, both red and green has a cost of 14

Another metric for calculating distances is *Euclidean distance*, the formula of which is given in equation 2.11. While this is more computationally expensive, it distinguishes the cost of diagonal movement from the Manhattan-movement presented earlier. Example of routes with their distance based on Euclidean distance is shown in figure 2.14.

$$d(p,q) = \sqrt{(q_x - p_x)^2 + (q_y - p_y)^2} \tag{2.11}$$

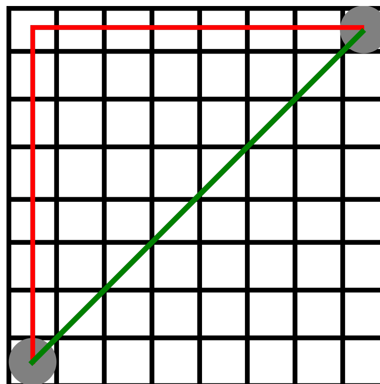


Figure 2.14: Euclidean distance, red = 14 and green ≈9.9

While not a metric, because of the triangle inequality, the *Squared Euclidean distance* is also utilized for computing distances. The formula is given in equation 2.12, and is simply the squared of the traditional Euclidean distance. This results naturally in significantly higher costs for all directions than the two approaches mentioned above but is faster to compute because of the removal of the square root operator.

$$d^2(p, q) = (q_x - p_x)^2 + (q_y - p_y)^2 \tag{2.12}$$

2.4.6 Heap

There are numerous different data structures available for storage, and one of them is through a binary tree or *heap*. The primary advantage of a heap is how fast it is to sort. Because of this property, it is commonly used in search algorithms. The basic principle is to ensure that the parent node always has a lower value than both of its child nodes. This is known as heap property. When this is done correctly, the graph looks like a binary tree, as shown in figure 2.15.

If a new element is inserted in the binary tree, it will take the spot at the bottom right. The ensuing sort algorithm will then compare it to its parent, and if it has a lower value swap the two. This process continues until the correct heap property is achieved. Since each item only has to be compared to its parent, the process is significantly faster than comparable methods for maintaining priority queues, especially one-dimensional lists.

When the A* algorithm needs the next item to explore, i.e., the lowest value, the top node in the tree structure is ensured to be the next node to be explored. After this element is popped from the queue, the last element is put on the top and then sorted down until the correct heap property is restored. Once again this process consists of considerable less computation than would have been needed that with a regular *list* structure.

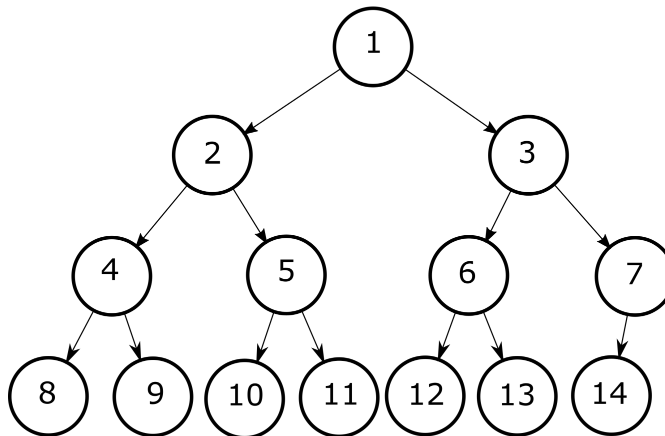
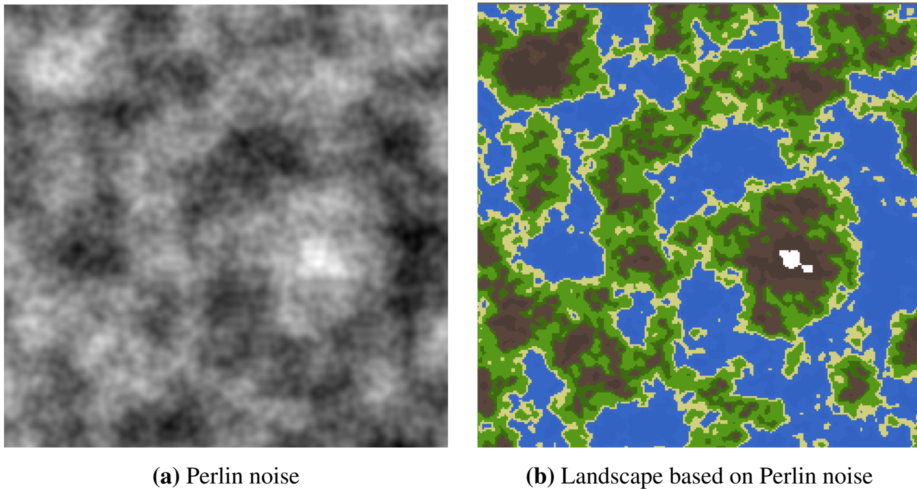


Figure 2.15: Heap

2.5 Noise

2.5.1 Perlin Noise

Perlin noise has been used for many different purposes, from the creation of sound and image textures to the height of the terrain and smoke effects [26]. The basic principle of Perlin noise is to let each point be dependent upon nearby points. By far the most common is to utilize this in either two or three dimensions, but in theory, it could be extended to any dimension. The result is a gradual transition between high and low values. An example of 2-dimensional Perlin noise is shown in figure 2.16a, where darker colors represent lower values.



(a) Perlin noise **(b)** Landscape based on Perlin noise

Figure 2.16: How video game landscape could be created based on Perlin noise

This noise could be interpreted as heights of the terrain and as such, be the basis of a procedurally generated landscape as is shown in figure 2.16b. This technique of noise based procedurally generated landscape is quite common in the industry today as it is a fast and simple method. Minecraft is one of the more popular games utilizing this type of landscape generation [4].

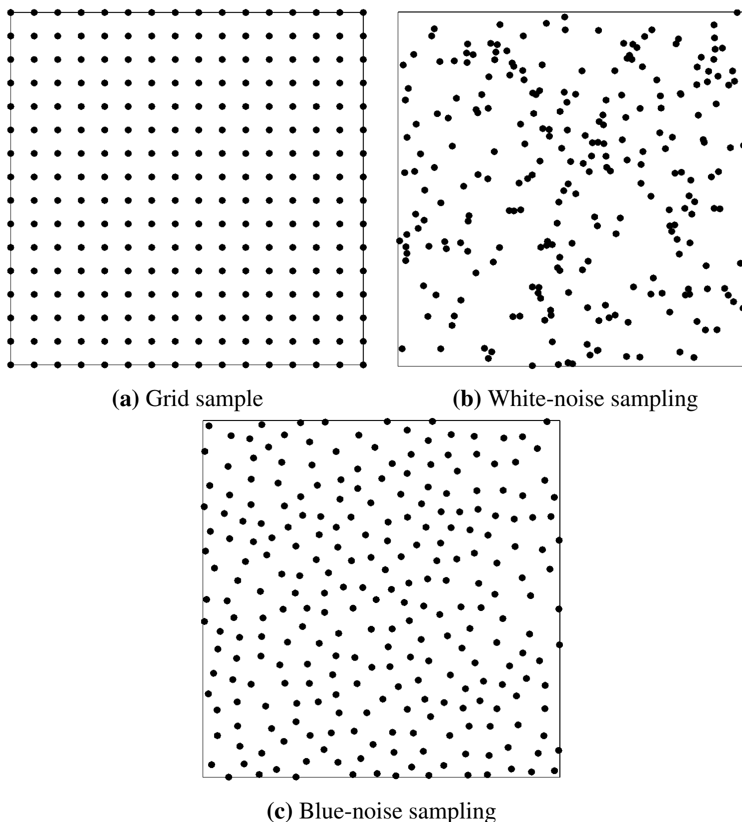
2.5.2 Blue-Noise Sampling

Noise is excellent when something needs to be sampled at random. However, the total randomness in white noise has its drawback. The sample points could be grouped together and as such, not give an accurate representation of the sample data. Another drawback is the number of redundant sample points if they are grouped together.

The opposite approach, grid-based samples or simply regular sampling, would solve part of the problem with the white-noise based approach. This utilizes a fixed grid with consisting spacing between each sample point. This mitigates the problem of sample-points being grouped together. However, this approach could suffer from aliasing. Besides, with the usage in this experiment, namely distributing components, this approach would not appear natural.

Blue noise sampling is a form of middle ground between the two approaches mentioned above. This is still based upon the randomness of noise, and as such, get the more natural looking results. While at the same time ensuring sample points are approximately evenly spaced out. This is done through the introduction of *restrictions* on each sample point, for instance, a minimum distance between each point. As such, it generates a randomized uniform distribution where no sample point is located on top of another.

A comparison of how the different sample points could be spread is shown in figure 2.17. Each of the different methods has exactly 256 sample points. However, the distribution of them is quite different.



(a) Grid sample **(b)** White-noise sampling

(c) Blue-noise sampling

Figure 2.17

There are several ways to categorize blue-noise sampling, according to Dong-Ming Yan et al. [27]. This could be done based on aspects such as the sampling domain, the shape of the sampling primitives, the properties of the sampling results, to mention a few. However, for this situation, the most beneficial would be to classify them based on the style of the algorithm used. According to the same study, there are three major types of blue-noise algorithms. 1. Poisson-disk sampling; 2. relaxation-based sampling; and 3. patch/tile based sampling.

In this thesis, the classic *Poisson-Disk sampling* method was used, and as such, it will be the only method that will be explained more in depth. The traditional method of such super-sampling was first proposed by Cook [28] and was called *dart-throwing*.

The algorithm is as straightforward as the name implies and is shown in algorithm 6. This was quite an inefficient algorithm with a complexity of $O(n^2)$. However, while keeping the same principle of generation at heart, much research has been done in the field since then, and faster algorithms have since been developed.

Algorithm 6: Poisson-Disk sampling through dart throwing

Input: samplingDomain, sampleRadius, threshold
Result: Set of sample points

1 ContinuousMisses = 0
2 **while** $ContinuousMisses$ ¡ $threshold$ **do**
3   Generate random sample point within the sample domain
4   **if** *sample point condlicts with existing* **then**
5   ContinuousMisses++
6   **else**
7   ContinuousMisses = 0
8   add sample point

A better implementation of Poisson-Disk sampling could be based on the paper *Fast poisson disk sampling in arbitrary dimensions* [29]. The modifications presented in this paper allows for the generation of N-dimensional blue noise sample in $O(N)$ time. As all usages in this experiment will be two dimensional, the presented optimizations will be illustrated as such.

The primary reason for the faster computation presented is a grid-based approximation. The available space for samples to be located at is divided into cells, where each cell is bounded by $r/\sqrt{n}$, where *r* is the radius of the sample points and *n* is the number of dimensions. In the 2-dimensional space used in this experiment each cell is then $r/\sqrt{2}$ in diagonal. This ensures wherever a point is located within the cell, it is guaranteed to cover it entirely, and as such only space for one point per cell. Further, this also ensures the radius of the point will never be larger than a given block of cells. As such, the dart throwing algorithm only has to look at discrete cell blocks close to the new point as opposed to compute the Euclidean distance to all points.

2.6 Unity Features

Unity is one of the largest game engines on the market today. According to their website, at least half of all mobile games are powered by Unity, and more than 60% of AR/VR content is made with their game engine [30].

The editor window in Unity is as shown in figure 2.18. On the left-hand side is the hierarchy of all *GameObjects* present in the game. GameObjects and their properties will be explained in more detail later. On the bottom is the explorer window where all files are located. To the right are the components, and other details, of the selected GameObject shown. Examples of such components are shaders, textures and custom scripts attached to that specific GameObject.

In the middle is a preview of the game. Unity has set up its axis system in such a way that directly up is the y-axis, and as such gravity pulls objects in the negative y-direction. The x-axis is to the right in the figure, and the z-axis is forward into the screen. As a result, a flat terrain mesh will span out the xz-direction.

While the mesh, as explained earlier, constructs the shape of an object this is not enough for it to properly appear in the game. Unity also requires a couple of other properties set up for it to be displayed. Additionally, a custom shader or texture often will be involved to give the mesh the desired appearance. For this exact purpose, Unity has created what they call *GameObjects*. These objects combine all the required components for a mesh to correctly appear in the game. While there are special GameObjects such as light sources and the Camera(which the player *looks through* while playing) that do not consist of a mesh, this experiment will pay little attention to such GameObjects. As such GameObjects will, for the most part, be centered around objects in the game a player is able to interact with.

The *Unity Assets Store* is a marketplace where all sort of assets are available for free or to purchase. Assets could be everything a developer would need from basic textures and shaders to automatic city generation tools. The external assets that will be used later are primary prefabricated models. These are finished GameObjects with mesh, textures, and sometimes scripts included and as such is easy to include in the game.

A useful feature of GameObjects in Unity is their *parent* property. When a parent object is moved, all their child objects are moved the same distance. This ensures that their relative distance and appearance is kept intact. There are many applications where this is useful. For instance, in a racing game, all the wheels of a car could be separate objects, and as such, allow them to be removed or altered individually. However, while they are attached to the car, they follow the rest of the car perfectly without individual movement set on each tire. Additionally, in the hierarchy view in the Unity editor window, all the child objects can be minimized and as such, allow for a better overview.

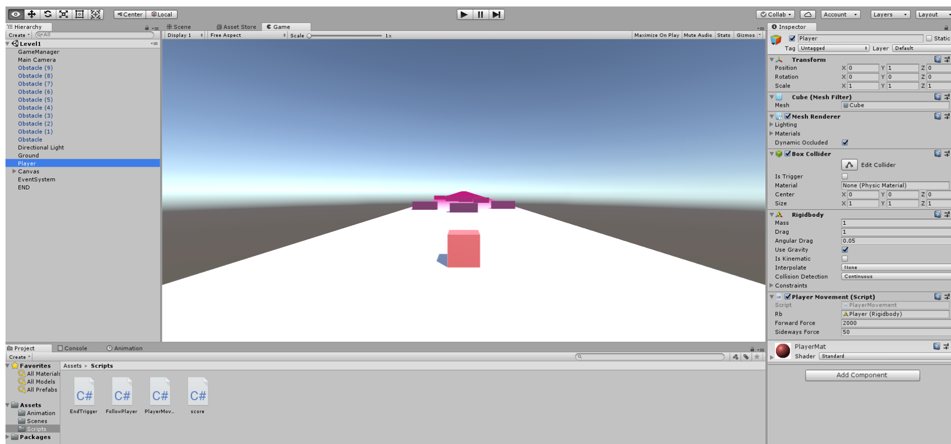


Figure 2.18: Layout of the Unity editor

2.7 The Specialization Project

The experiments in this thesis are in large parts based on basic features developed by Øystein Brox in [14]. In this section, a summary of the results from that study will be presented. Parts of the approaches and algorithms have been altered during this experiment, and all significant changes will be addressed in section 4.1.

The most important features developed in that thesis were the procedural generation of meshes, and the ability to rescale them in run time. Further, the experiment implemented algorithms for augmenting such meshes to the shape of bell-curved mountains.

2.7.1 Meshes

Algorithm 7: Create new MeshData

Input: DistanceDownscaler, Size
Result: new meshData object

```
1  Calculate offset i X and Z direction;
2  Calculate steplength based on distance downscaler parameter;
3  Calculate number of vertices per line (VertPrLine);
4  vertexIndex = 0;
5  for z = 0; z < size; z += steplength do
6      for x = 0; x < size; x += steplength do
7          Add new Vertex at coordinates: (OffsettX + x), 0, OffsettZ + z) ;
8          Add UV
9          if not on outer/positive edge then
10             Add Triangles with the following VertexIndex:
11             (vertexIndex, vertexIndex+VertPrLine+1, vertexIndex+VertPrLine);
12             (vertexIndex + VertPrLine + 1, vertexIndex, vertexIndex+1);
13         vertexIndex++ ;
14 return MeshData;
```

The algorithm starts by calculating the offset i x and z direction. This is done in order to center the mesh. The reasoning behind this is for easier abstraction when converting from the relative positioning of each mesh to actual global coordinates. The difference is shown in figure 2.19.

Further, the *step length* and *VertPrLine* has to be calculated. However, these values are only used when downscaling the meshes, as will be explained later. For now, consider vertices per line to be equal the size and step length to be one. A counter, *vertexIndex*, is created in order to keep track of each vertex and the position in the array of the current vertex. Then the algorithm loops through and creates all the vertices and connects them correctly together.

Unity has its own *Mesh* class that the meshData needs to be converted into in order to be rendered by Unity. However, the underlying data types of the Unity *Mesh* class is identical to the data types chosen in *MeshData*. As a result, the process of generating an instance of the Unity class is simply setting the parameters equal to one another, i.e., $Mesh.vertices = MeshData.vertices$.

Rescaling meshes, i.e., change its resolution, is essential in game optimization. When a player is far from an object, they are unable to see the fine details, and as such, there is no need to render the objects with such level of detail. The mesh generation algorithm also allows for such rescaling. The basic principle of how it is done is to create a new mesh of lower resolution and discard the old mesh instance.

In algorithm 7 it is not the *size* that has to be decreased, but rather the *steplength*, which is

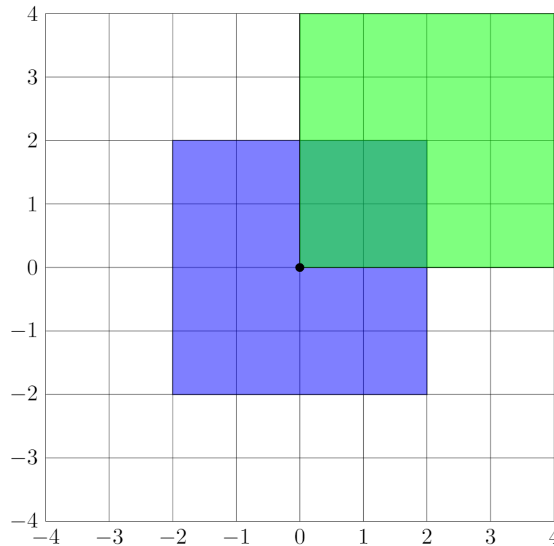


Figure 2.19: Effect of *offset*-parameter. Blue is with, and green without. Both are located at the origin

based on the *distanceDownscaler*. The size determines the overall size of the mesh, which needs to be consistent to avoid gaps between meshes. The factor of downscaling, i.e., the number of vertices skipped, is nontrivial. In order to ensure the meshes stay adjacent, without the need to redistribute all vertices within the mesh, a step length equal to a factor of the number of edges is required. As all terrain meshes in this experiment are 251x251 vertices, i.e., 250 edges in each direction, the step length is either 1, 2, 5, or 10.

2.7.2 Mountains

The creation of a consistently shaped mountain was implemented with the two-dimensional Gaussian function, see equation 2.13. The main advantage of this function is its simplicity, in combination with the opportunities for adjustments. The height of the peak, as well as the steepness, could be altered with ease.

$$f(x, y) = A \exp(-(\frac{(x - x_0)^2}{2\sigma_x^2} + \frac{(y - y_0)^2}{2\sigma_y^2})) \tag{2.13}$$

In equation 2.13, the *A* term decides the height of the peak of the mountain. The steepness of the bell curve is adjusted with the standard deviation term, σ. Examples of the different results can be observed in figure 2.20 where the one-dimensional Gaussian function is plotted in Matlab with different parameters.

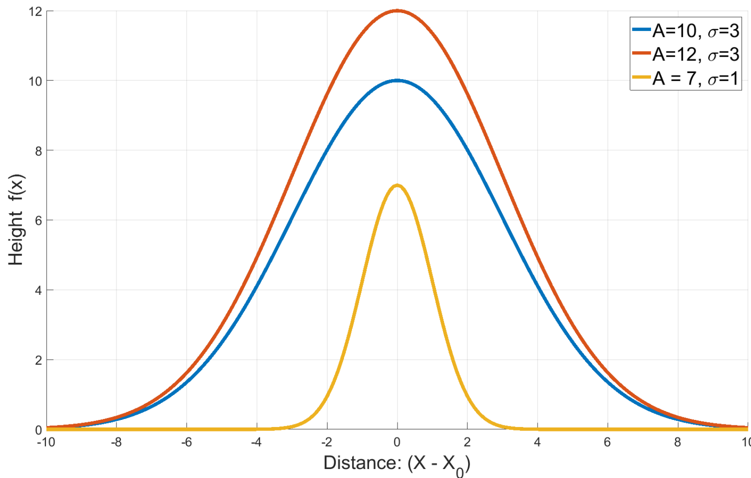


Figure 2.20: Shape of a mountain

Algorithm 8: Adding a new mountain

Data: list AllMountains
Input: peakCoordinates, chunkOfPeak, radius, height

1 StandardDev = radius/3
2 affectedChunks.Add(chunkOfPeak);
3 **if** *radius outisde chunkOfPeak* **then**
4   affectedChunks.Add(chunk that is affected);
5 AllMountains.Add(this mountain);

Algorithm 8 is responsible for creating one new instance of the Mountain-class and appending it to the list of mountains. It has to be called once for each mountain as part of the setup process. This function also calculates some additional parameters for ease of access later. The standard deviation in the experiment was set equal one-third of the radius. For almost every application, this is a decent choice of standard deviation. This is based on the reasoning as after three times the standard deviation the height is almost negligible. This is also observable in figure 2.20. Then the coordinates of the chunk containing the peak are added to the list of all chunks that are affected by this mountain, i.e., the meshes that the mountain has to be drawn into. Then the algorithm checks which, if any, other meshes have to be augmented and appends their coordinates to the list. Finally, the mountain initialization is complete, and the mountain is added to the public list of all created mountains, named *AllMountains*.

Algorithm 9: Draw mountains on a mesh

```
Data: list AllMountains
Input: meshData, chunkCoordinates, distanceDownscaler
Result: augmented meshData object
1  foreach mountain in AllMountains do
2      if mountain.affectedChunks contains chunkCoordinates then
3          myMountains.Add(mountain);
4  foreach mountain in myMountains do
5      Vector2 meshOffsett = chunkCoordinates - mountain.chunkOfPeak
6      radius = radius of this mountain with respect to distance downscale
7      standardDeviation = standard Deviation based on distance downscaler
8      Calculate the start coordinates in the mesh, startZ & startX
9      Calculate the stop coordinates in the mesh, stopZ & stopX
10     for z = startZ; z < stopZ; z ++ do
11         for x = startX; x < stopX; x ++ do
12             meshData[height at coord x,z] += Equation 2.13
13 return meshData;
```

For augmenting a mesh algorithm 9 is utilized. First, it loops through all mountains created by algorithm 8 and checks if any of them affects the current mesh. All relevant mountains are appended to a new list the algorithm then loops through. The algorithm then loops through all x and z coordinates of the mountain in order to augment all vertices. The new height of each vertex is its original height and then increased with the amount computed by equation 2.13. This allows for mountains to overlap each other. This was experimented with in this thesis. However, it was not attempted to procedural place new mountains as will be attempted in section 4.2.1.

Chapter 3

Specification of the Experiment

3.1 Specification

The overarching goal of the experiment is to create a playable *proof of concept* based on procedural generation. This includes a controllable character as well as texturing that conveys a realistic appearance. However, as this is not the main focus of the experiment, these results are not to be comparable to most modern games. Their objective is to be realistic enough not to break immersion and as such be fit for demonstration. The primary goal is to explore different techniques for procedural generation.

The proposed experiment is to be based on the work done and presented in section 2.7, and consists of two primary areas of research. They are *mountains as primary components* and *procedural generation of roads*. Together they should create the basis for what could be an open world explorer game. While the parameters of each primary mountains are to be specified ahead of time by the developer, all other aspects of the mountains are to be created procedurally while the game runs. Additionally, all content must be replicable. In other words, if a player plays a level, with certain terrain, the system must be able to reproduce the exact same terrain next time if the player wishes.

The mountains are to be based on the procedures explained in section 2.7.2, however, they are to be taken a step further. While each primary mountain is to be specified ahead of time, they are to be done so with as little metadata as possible. Only the most essential information to create a mountain is to be specified. This primary mountain-object is to be the basis of further studies into its applications.

First, there will be a study on how to spread several smaller mountains on top of a primary mountain. Different strategies on how to distribute them are to be implemented and compared. Likewise, procedural creation of mountain ranges is to be explored. The basis of this study is to connect two primary mountains, created ahead of time, with new pro-

cedural generated mountains. Different methods of determining the position of each new mountain are to be studied and compared.

The last area of research regarding mountains is a study into utilizing the mountain component to recreate the natural unevenness of the ground. In other words an attempt to mimic natural bumps in the terrain through incredibly low mountains.

The other major area of research is procedural generation of roads. With the aforementioned terrain created by mountains, a study on how to generate a natural path for a road is to be performed. The start and stop coordinates of the road are to be specified beforehand, as with the mountains. At the start and stop of the road, cities are to be located. They will serve as natural methods of terminating the roads. Additionally, lakes will be created as primary components similarly to the mountains. They are also to function as impassable obstacles for the road.

The path the road should follow is to be generated based on a grid search algorithm. As this process has to be performed after the world is generated, and as such, in practice when playing the game, a study on several modern search algorithm is to be performed. This in order to find a suitable algorithm. One of the studied algorithms is to be implemented and function as the basis of a more in-depth study on road generation.

Different cost functions are to be studied and implemented in order to better understand the challenges of procedural generation of roads based on terrain. Additionally, a study on the trade-off between temporary storage of unused roads and the process of recreating them when needed is to be performed. Lastly attempts to make the road appear realistic in the game, in order to not break the immersion of the player, is to be implemented.

3.2 Summary of the specification

The following features are to be implemented with the ability to be replicated if desired:

- Create a mountain based on a few predetermined hyperparameters
  - Explore different strategies to procedurally located several smaller mountains on top of the aforementioned primary mountain
  - Explore different strategies to connect two primary mountains into a mountain range procedurally
  - Explore whether small mountains could be the basis for simulating natural unevenness of the terrain
- Procedural generation of roads based on the terrain
  - Ensure the roads follows a natural path from one city to another based on the height of the terrain while avoiding lakes
  - Study different modern pathfinding algorithms and chose one for implementation
  - Study different designs of cost functions and how they affect the paths
  - Explore the benefits and drawbacks of recreating a road as opposed to saving it
  - Utilize different techniques to make the road appear realistic in the game
- Implement a fully playable character
- Create a shader that conveys the terrain in a realistic way for better visualization

Chapter 4

Implementation and Results

4.1 Improvements on the Pre-Existing System

While the basis of mesh creation and rescaling were already done, some augmentation and improvements have been performed. Several minor alterations have been done to improve the code itself, but the most significant change is the introduction of heightmaps. This heightmap is a two-dimensional float array, where each float represents the height of the corresponding vertex. There were several reasons for the implementation of such software architecture.

First and foremost, a heightmap is significantly easier to use. The heightmap always represents the mesh at full resolution. As such, when augmenting the terrain, there is no demand for recalculating what vertices to augment based on the resolution. Additionally, by this split, the system would be easier to make multi-threaded since different levels of protection could be put on the mesh and heightmap.

Lastly, while it increases memory usage, it significantly lowers the computational demands of the system. Previously algorithm 9, which augmented the mesh in the shape of a mountain, had to be called each time a new mesh was created. With an underlying heightmap, this algorithm only has to be called once per mesh since the terrain is kept at full resolution in the heightmap. When a new mesh is in demand, algorithm 7 as described earlier is utilized, only instead of size, a heightmap is the input parameter. Further, the height is set equal the height in the map and not zero as was done originally.

Further alteration to the algorithm for creating meshes was also performed. In an effort to increase the maximum resolution of each mesh, the global size was drastically reduced while maintaining the same amount of vertices. A noticeable side effect is increased computation due to a significantly higher number of meshes being visible at the same time. However, this augmentation allows for higher detail in the terrain.

Finally, algorithm 9 was augmented to start its iteration at either the start of the mountain or the heightmap, depending on what is most beneficial. This improvement made the process significantly faster for mountains spanning across multiple chunks.

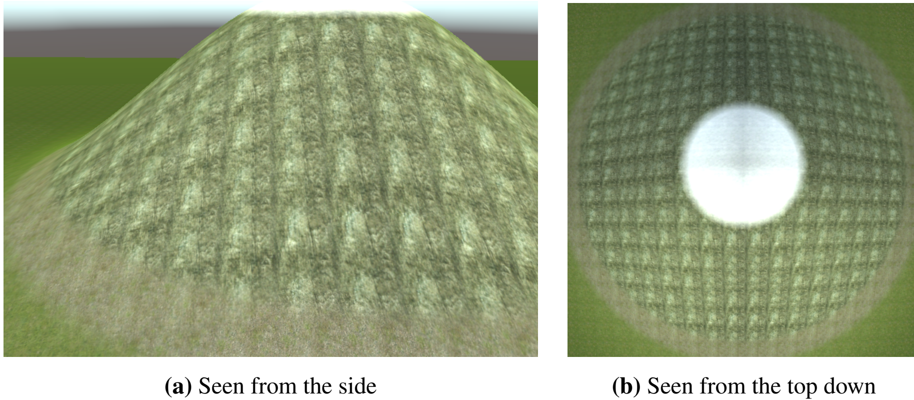
4.2 Mountains

The creation and combination of *mountains* was, as described in chapter 3, a fundamental design principle. The necessary information to create and replicate, each mountain should be kept at a minimum. This laid the basis of creating a mountain based solely on the essential information. As a result each mountain is generated based on three parameters; *coordinate of the peak*, *height* and *radius*.

The algorithms for creating mountains were as described in section 2.7.2 with the augmentation presented earlier. In addition, the mountains were specified in an external XML file as opposed directly in the script. This allows for a change in scenery without changing any part of the code itself, as well as being more scalable. Further, in theory, another program or script could handle procedural placement of mountains in the landscape. This is an exciting aspect of component-based procedural generation that will not be attempted in this experiment.

4.2.1 Mountains on Top of Existing Mountains

The basic curve-shaped mountain is far from anything in the real world. As such a combination of several mountain components was utilized in the specialization project [14] in an attempt to break the smooth mountainsides. In this thesis, the experiment will be taken a step further by exploring different strategies of populating a mountain with several smaller mountains. Two main approaches have been implemented and compared in this experiment, white noise, and blue-noise sampling. For both methods, all new mountain peaks will be located within the radius of the primary mountain.

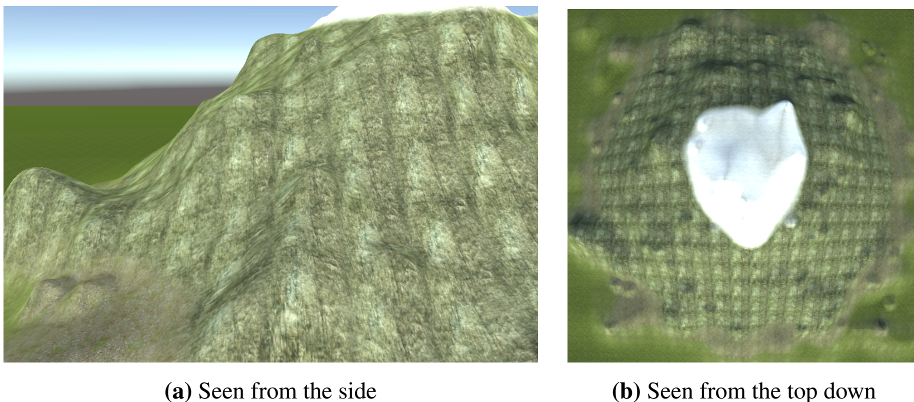


(a) Seen from the side **(b)** Seen from the top down

Figure 4.1: Original mountain without any additional smaller mountains

First, with the complete randomness of white noise, the result did not appear machine-like in an almost grid-like fashion. The implementation randomly generated a point within the original mountain radius, and randomly gave it a height within 8% primary mountain height. This added randomness to the height of the new mountains drastically increased the natural appearance. The general appearance of the mountain was greatly improved by the addition of several smaller mountain components.

The intuitive drawback of several mountains being located on top of each other also turns out to be less of a problem that it might appear at first. Even when two smaller mountain peaks are located almost precisely at the same spot, the accumulated extra height is not unnatural compared to several real mountains. As a result, this side effect might even be beneficial. There has to be implemented a way of determining when to stop adding new small mountains, for instance, a maximum number, but apart from that, the white noise approach has many advantages. Example of a primary mountain with several additional small mountains located on top of it is shown in figure 4.2.

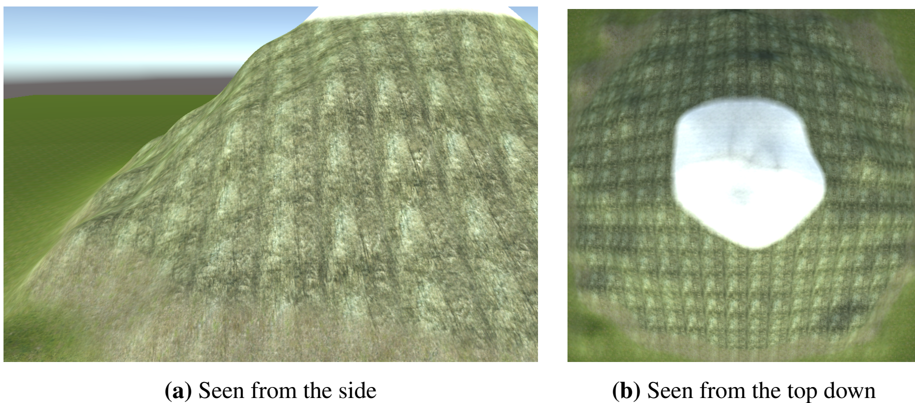


(a) Seen from the side **(b)** Seen from the top down

Figure 4.2: Mountain populated through white-noise

Another method attempted was to utilize blue-noise sampling, as described in section 2.5.2. The sample domain was a two-dimensional plane with spans out the two times the radius of the mountain in each direction. The coordinate of each sample point was converted from the local domain to the global frame through the transformation matrix in section 2.3.1. There was no rotation, and as such, the equation was simplified to a simple translation. The translation performed ensured that the sample domain was mapped directly on top of the mountain.

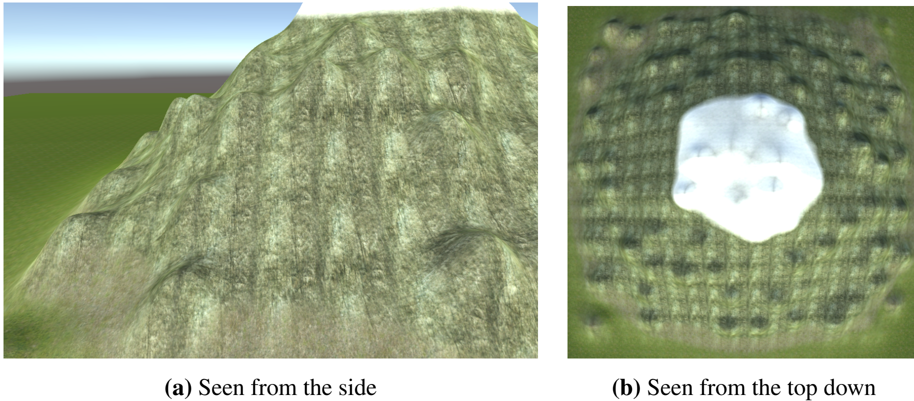
This approach ensures the new mountains is spread quite evenly and that most of the original mountain will be covered in new mountains. While this at first sounds pretty advantageous, it turned out to be not as ideal as the aforementioned white noise based approach. Since all new mountains were evenly spread the height must be kept quite low. If not the mountain would become unrealistic as it had smaller mountain peaks evenly spread all over the sides. As such blue noise sampling could only be used to disrupt the smoothness of the original mountain and not to create new exciting spots to explore. The Same mountain with new mountains based on blue noise sampling is shown in figure 4.3



(a) Seen from the side **(b)** Seen from the top down

Figure 4.3: Mountain populated through a single layer of blue-noise

In an attempt to improve the outcome based on blue noise sampling, a second layer was added. This time the height interval of each new mountain was kept while the radius was reduced. This resulted in a great combination of larger mountains to break the smoothness with smaller and sharper mountains to create interesting slopes and point of interest. This is observed in figure 4.4, where the mountain shown in figure 4.3 has a second layer of mountains put on top.



(a) Seen from the side **(b)** Seen from the top down

Figure 4.4: Mountain populated through two layers of blue-noise

All the different approaches were based on noise and as such reproducibility is inherently an issue. The implemented solution was to design a deterministic seed. There are several different methods for ensuring that the seed for each mountain is consistent between each time the game starts, while at the same time different for each mountain. The latter had to be ensured, so not all mountains became identical. The implemented solution was to augment each seed based on static properties of the mountain. In other words, a mountain with a given set of hyperparameters will always be identical, while a small change will drastically change the outcome.

4.2.2 Mountain Ranges

Once the process of creating single, natural looking mountains was complete, they could be used for new studies on component-based procedural generation. The next phase of the proof of concept attempted to generate vast and realistic looking mountain ranges. The implemented solution was to utilize two primary mountains located ahead of time and then connect them procedurally. Three different approaches were attempted in order to generate this; Linear interpolation, blue noise sampling, and white noise.

Figure 4.5: The primary mountains that spans the mountain range

First attempted were the method based on linear interpolation. This ensures that the entire void between the two mountains is wholly covered. This is pretty essential as if there are too large gaps between two mountains in a mountain range it simply seizes to be a mountain range. With this in mind, in combination with the computational efficiency of linear interpolation, there are some definite advantages.

However, the linear interpolation approach has a significant drawback as well, the lack of a natural appearance. Few things in nature are as perfectly evenly separated as those mountains, and this could lead to a break of immersion. There are possible steps to break the evenness of linear interpolation, and those will be discussed further in section 5.1.2.

For all the new mountains that make up the mountain range, new small mountains as presented earlier could be utilized. This would, for obvious reasons, improve the outcome. However, this would also make it harder to observe the effects, and as such compare, the results from this experiment. For this reason, the shape of all new mountains in the mountain range will be smooth bell curves.

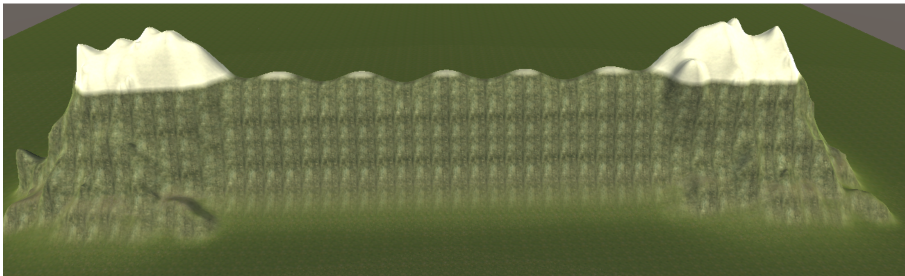


Figure 4.6: Mountain range based on linear interpolation

Another solution for the placement of mountains is through blue-noise sampling. The first dimension of the sample domain was the distance between the mountains minus each of the radius. This ensured the mountains would fit between them while not having their peaks directly on top of the originals. The other dimension enabled the mountains not to be located just in a straight line. Based on tests, a realistic range was the average radius of the mountains. As such, each new mountain peak could be located about half the radius of a mountain in each direction. The area new peaks could be located is shown in figure 4.7.

Additionally, the radius of the mountains in the game was twice as big as the radius set in the Poisson disc algorithm. This ensured they overlapped to some degree and as such not break the range while still spaced out.

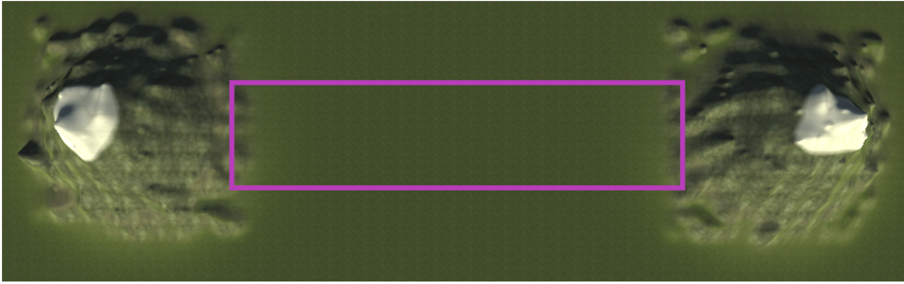


Figure 4.7: New mountain peaks are located within the purple area

In order to transform the coordinates from the sample domain to the global coordinate system of Unity, a transformation matrix was utilized. The rotation matrix had to be included for this experiment, and as such, the entire formula given in equation 4.1 had to be implemented. The angle, θ was between the desired angle of the mountain range and the first mountain, while translation ensured moved the entire frame to the location of the first mountain. This ensured the general layout of the mountain range were as indicated in figure 4.7 despite any angle between the mountains.

$$x_a = Cos(\theta)x_b - Sin(\theta) \cdot y_b + t_x \tag{4.1a}$$

$$y_a = Sin(\theta)x_b + cos(\theta) \cdot y_b + t_y \tag{4.1b}$$

This approach had some distinct advantages compared to the aforementioned linear interpolation. First and foremost, the more natural spread of the mountains. With the spacing between each mountain altered and maybe even more importantly, they were not all on a straight line; the appearance is a lot more realistic. Despite this, the approach has a significant drawback as well, the chance of gaps in the mountain range. Since the method is based on randomness, there is, albeit incredibly small, chance that the placement of mountains make the number of continuously rejected samples pass the threshold prematurely. There is also a concern with regard towards reproducibility that has to be addressed through a nonrandom seed the same way as the earlier section explained. Example of a mountain range based on blue noise sampling is shown in figure 4.8.

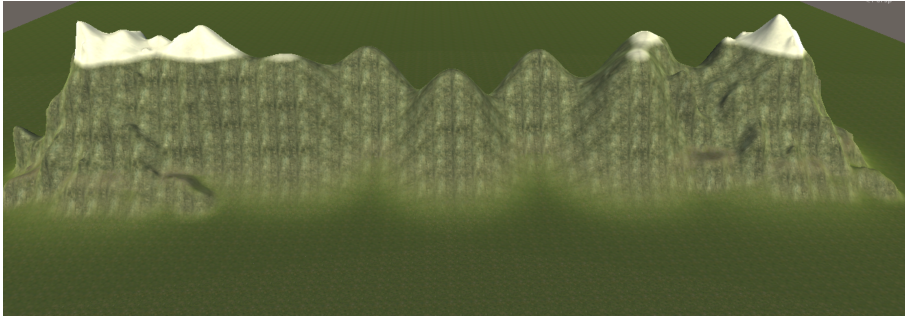


Figure 4.8: Mountain range based on blue-noise sampling

The final implemented approach was with the *total* randomness of white noise. Implementation wise it was done the same way as blue noise sampling with the new mountain peaks to be located within a similar bounding box.

The primary advantage of white noise placement is the lack of repetition and as such a more natural appearance. However, as shown in figure 4.9, there are some obvious drawbacks. Since the mountains are not kept at a minimum distance between each other, there is a high possibility that two mountains would overlap far more than what is natural. The issue of knowing how many mountains to create is also nontrivial in this case. With linear interpolation, the number of mountains to be distributed is easily computed ahead of time. Blue-noise sampling fills the space between the two primary mountains and is finished when it can fit no more. However, with this white noise approach, there is far from any guarantee that the entire mountain range is full after a given number of new mountains. As a result, this method is far from ideal.

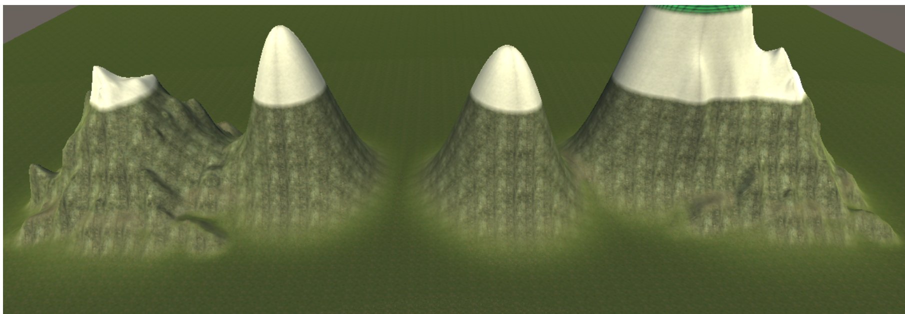


Figure 4.9: Mountain range based on white noise

4.2.3 Natural Unevenness Through Mountains

The last significant phase of research on mountains as primary components was the attempt to mimic natural unevenness in the terrain. Based on the previous experiments, the most feasible approach is the utilization of blue noise sampling to spread small mountains across the terrain. A grid-based approach would more than likely have an unnatural appearance. While white noise could easily result in too many on top of each other and as such almost create a new primary mountain.

The same blue noise sampling algorithm, as used earlier, was utilized across the entirety of each mesh. Each mountain was given a random height, within a pretty low interval. This ensured a more varied terrain as opposed to if each bump in the terrain were identical. The new more varied terrain is shown in figure 4.10. The new grassland is undoubtedly more realistic as few natural areas are entirely flat.

Figure 4.10: Unevenness created through mountain components

For comparison, an attempt with the usage of Perlin noise was also performed. However, instead of shaping the entire terrain, the noise was only used for the smaller alteration of the terrain. Unity has a built in Perlin noise generator, and as such, the implementation for comparison was relatively quick. The terrain augmented through Perlin noise is shown in figure 4.11. Based on the figures presented here, the Perlin noise approach gave slightly

sharper edges. However, this is certainly modifiable based on the hyperparameters of both methods. With Perlin noise as well, a consistent seed would have to be generated for consistent terrain. In total, the result for both methods are quite identical, but a more in-depth discussion on the addition of smaller will be given in section 5.1.1.

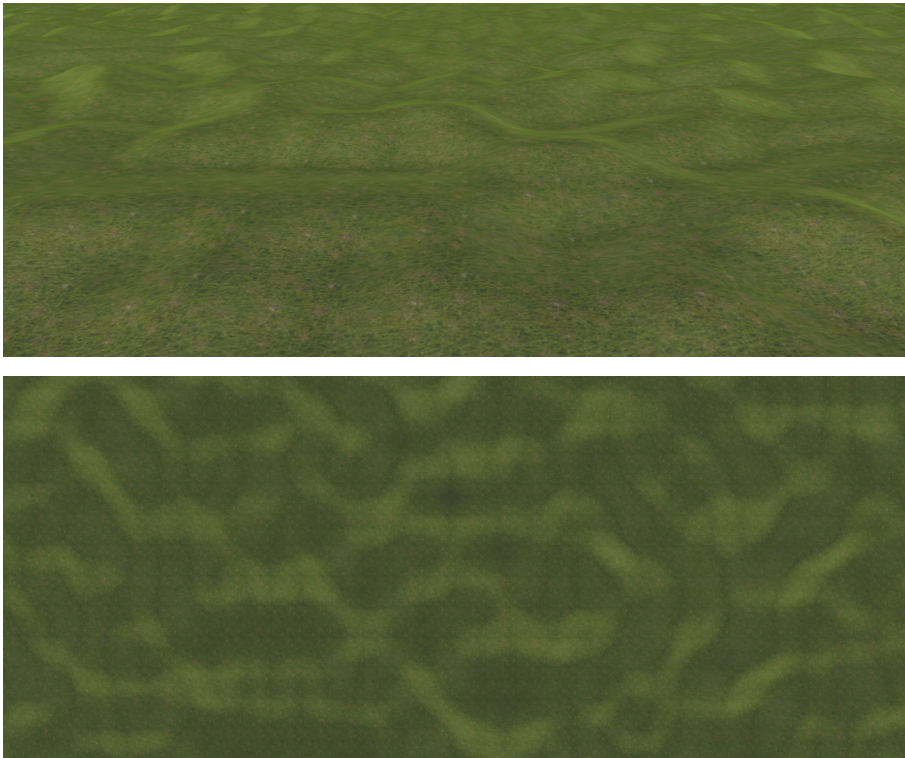


Figure 4.11: Unevenness created through Perlin noise

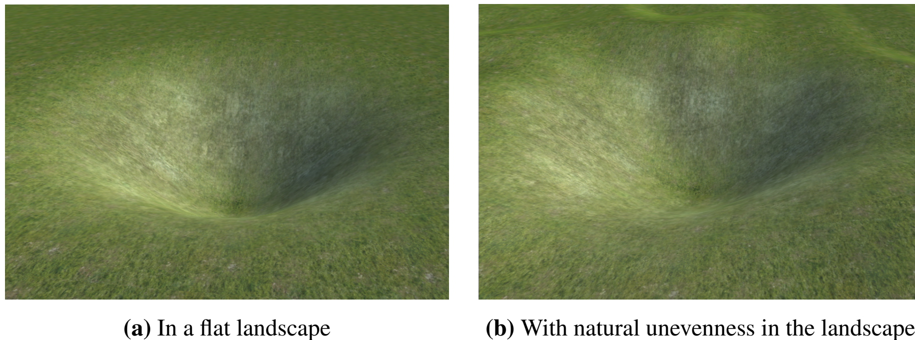
4.3 Other Components

As presented in the specification for this experiment in chapter 3, additional components were to be introduced before any roads could be generated. They were implemented in a similar fashion as the primary mountains with the essential information given ahead of time — likewise, they were also specified in an XML-file for more straightforward augmentation and future scalability.

4.3.1 Lakes

The creation of lakes consists of two separate challenges. First, a pit has to be excavated in the landscape, and secondly, it has to be filled with water.

For the first challenge the algorithm used for mountains, Algorithm 9, was used with only minor changes to the variable names. The process of including additiional smaller mountais on top, described in section 4.2.1, was not implemented. This decision was based on several factors, but most importantly, because the bottom of the lake would rarely if ever, be visible. As such, any steps towards creating a more lifelike floor of the lakes would yield little effect in this experiment.



(a) In a flat landscape **(b)** With natural unevenness in the landscape

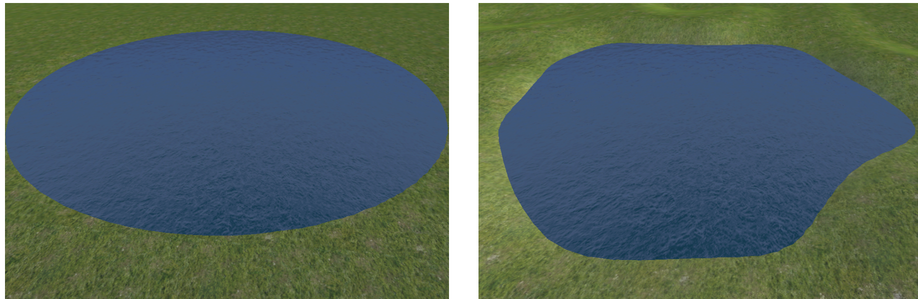
Figure 4.12: Pit of the lakes

The second challenge is to add water to the excavated pit. The process of representing realistic water has been a long-standing challenge in computer graphics [31]. There are several different approaches to representing water, everything from a primitive flat surface towards more advanced particle and physics-based methods. The latter methods yield more realistic results and are interactable in all dimensions. The basic principle is to track either the water particles themselves or the entire body of water either through *Euclidean* or *Lagrangian* methods[32]. While such a lake would be more realistic, and interactable for the player, it would both be significantly harder to implemented and more computationally expensive. Based on the implementation time and computational cost, a simpler option was chosen.

The water implemented in the game is a simple flat surface with precomputed motion. As a result, the lakes look relatively realistic in the landscape, figure 4.13, while maintaining a low computational cost. More complex modeling of the water would allow the player to interact with the lakes in a natural way, but that was not within the scope of this experiment. The precomputed water chosen was part of the available water assets in *Unity Technology*s available package of *Standard Assets* [33]. Additionally, as Unity Technology created the water asset, the process of including it was much more straightforward. Only a few adjustments were necessary for the asset to be integrated flawlessly in the scene.

The planechunk where the center of the lake was located became the parent for the water.

This ensured that when the player was sufficiently far away from the chunk, and the chunk was removed from the visible scene, so was the water. Both objects were still stored in memory, but as an optimization feature for the GPU, they were no longer rendered on screen.



(a) In a flat landscape **(b)** With natural unevenness in the landscape

Figure 4.13: Lakes in the terrain

4.3.2 Cities

How to include cities in the implemented experiment had to be a deliberate choice. For a city to be immersive, the player has to able to move freely within the city and look at different buildings. Such an implemented city would require a lot of computational resources and take up a large portion of the in-game world. As a method to combat this, many games split a city into two distinct parts. There is the outside boundary, typically a wall, that is visible from afar. However, while the player is located outside the city walls, the entire city is empty. When the player interacts with the gate a new scene, inside the city, appears while everything on the outside disappears.

As the primary focus of the implementation is to explore cities connected on a large scale, such optimization was the foundation of the implementation. Further, the intended design made no emphasis on what was happening within the city walls, and as such, the creation of a living, breathing city was not attempted. Nonetheless, the basic design still allows for such expansions to be attempted in the future.

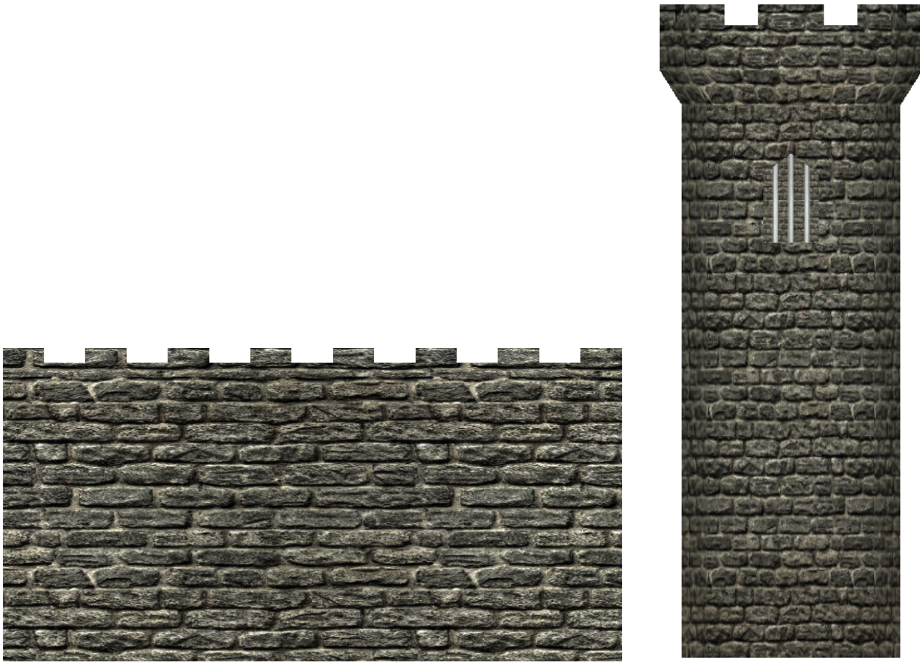


Figure 4.14: Wall and tower models used for cities

The basic structure of the city is based upon two prefabricated assets created by *AquariusMax* and made available freely on the Unity Assets Store [34]. Both prefabs are shown in figure 4.14. The basic algorithm of creating the city perimeter is first to determine each *corner* of the city. At each of the aforementioned spots, a new instance of the prefabricated tower is created. Then walls are constructed between all of the towers.

However, the wall between two towers has to consists of several instances of the prefabricated wall model. If simply one copy of the wall were put in there and stretched to fit between two towers, the result would not be realistic at all. Instead, linear interpolation is used to spread out a computed number of wall pieces evenly. This ensures each instance of the prefabricated wall is not stretched or distorted so much it loses realism. Finally, all the individual pieces of the city are given a common *parent* GameObject. Through this single object, the entire city could be moved around or rotated. Additionally, this object is given the PlaneChunk as a parent similarly to the lake.

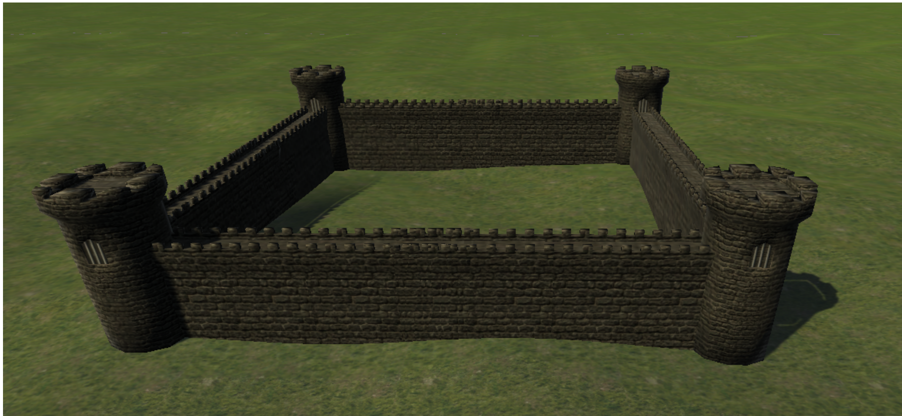


Figure 4.15: A city located in the landscape

4.4 Roads

4.4.1 Choice of Pathfinding Algorithm

The path of each road was determined based on a search algorithm in this experiment. There are a vast number of different search algorithms available, and as such research had to be done in order to narrow down the options. Based on research on both classical pathfinding algorithm as well as state of the art in the video game industry, four algorithms were selected. All of those are presented in greater detail in chapter 2; *Dijkstra's algorithm*, *A**, *D** and *HPA**.

Based on the demands in this experiment, there are a couple of properties that are especially desirable. First and foremost, the algorithm has to efficient as all roads are to be computed while playing the game. Secondly, the path has to be optimal, or at least close enough to appear realistic. Lastly, the path calculated by the algorithm has to be reproducible. In other words, the road needs a consistent path between each time the exact same level is played.

Dijkstra's algorithm is well known for finding the optimal path through a graph. However, it is quite computationally expensive. The breadth-first approach clearly loses out to the other best-first approaches with respect to the performance in a computer game. As a result, Dijkstra's algorithm was dropped, and the focus was on the other alternatives.

D* has some unique features that are well suited for procedurally generated terrain. The ability of the algorithm to explore new areas and not require complete knowledge of the terrain is well suited for such an experiment as this. However, for this specific implementation, the algorithm would either have complete knowledge of a selected chunk of terrain or none at all. Also, since it is used for generating roads, and not move an agent, there would never really be any discoveries. The only way around it would either be for the algorithm

to explore new areas on its own, or halt the road creation until the player discovered new areas.

If the algorithm were allowed to explore and create the road in the process, the result could end up looking far from realistic. In theory, the road could be heading straight towards a lake, then when the algorithm discovers it, do a complete 180° turn. While this is realistic behavior for a character exploring, such a path for a road is unheard of. In all realistic scenarios, the entire relevant part of the map would be explored before a road could be constructed. When this is the case for D* the entire path would be constructed in the first iteration of LPA*, which is identical to regular A*.

While similar, HPA* and A* has some distinct differences. HPA* computes a lot of the required pathfinding ahead of time and as such is incredibly fast when required. The speed of which it is able to compute a path is ideal for roads that are to be generated while the game is already running. The path created is not optimal. However, it comes quite close. As long as the transition points between the clusters are consistent and not too far apart, the road would be both replicable and optimal enough for realism.

The major drawback with this algorithm is the required computation ahead of time. While in many games, this could be done by the developer once before the game is sold, that is not possible in this experiment. As all the terrain is created after the game starts, all transition points and internal traversal costs have to be computed after the game starts as well. This would require much unnecessary computation for relatively few roads.

With this in mind, the traditional A* was chosen for implementation. The algorithm fulfills the essential requirement of creating an optimal path quite fast. An additional benefit of utilizing A* is several other modern pathfinding algorithms are based on A* and as such a lot of the results of this experiment is transferable to other algorithms.

4.4.2 The Implementation of Roads

The roads are created while the game is played based on a few parameters set before the game is started. The basic data required for the creation of a road in the experiment is two points that are to be connected, as well as a parameter for the size of the road. The last parameter could, in theory, be decided by the game; however, for the simplicity of testing different approaches, it was decided ahead of time. Additionally, each road is given a unique ID for the scripts to tell all the different roads apart. This information creates the foundation of which to build a road.

The next layer of necessary data for roads are the path computed by the A* algorithm. A custom class was implemented in C#, and each instance is one road with all the computed data and necessary information. In this class the road is also split based on what mesh it resides in. For each chunk there is a list of arrays containing all the computed coordinates of the road within its domain. As a road, in theory, could swirl back and forth between two chunks in order to avoid obstacles, there was an apparent demand for supporting several *parts* of the road connected to the same specific chunk position key. As such, a list

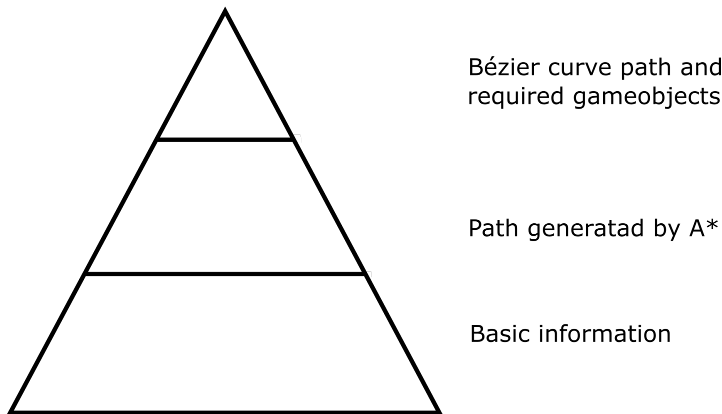


Figure 4.16: The layers of data structures that make up a road

of arrays was chosen in order to allow for several parts of the road belonging to the same chunk while at the same time ensuring the different parts are not connected directly.

The top layer is the road as it is represented in-game to the player. This contains the mesh, texturing, and all other necessary components. This is by far the largest in term of memory required to save everything. The details of the visualization and how to make the road look as realistic as possible is explained further in section 4.4.5.

Algorithm 10: Road algorithm, called each time a mesh is updated

Input: ChunkCoordinate, Resolution

```
1 if Possible to compute a new road then
2     Put all required meshes together
3     Call A* with this graph
4 Fetch all roads affecting me
5 for Roads in dictionary, but not present in the world do
6     for each array in the list of road parts in the current chunk do
7         Create new Belizer curve for my road
8         Create mesh based on the points
9         Create Road GameObject
```

Every time a mesh is either created or resized algorithm 10 is called for ensuring all roads are created and shown as they are supposed to be. First, the algorithm attempts to create new roads based on this either new mesh or new resolution. In the implementation, two different roads are included. First, large highways that are to be created and visualized as soon as possible, i.e., as soon as all the required heightmaps are created. The second type of roads were smaller country roads, and those only appear when a mesh is at full resolution. As a result, they were first created when a player is close enough for one of the affected meshes to be drawn at full resolution. While they, in theory, could be

created earlier, this was done as part of the experiment explained in section 4.4.4. The only difference in appearance of the roads is the width of the mesh, as will be explained in section 4.4.5.

If the algorithm decides the path of a new road has to be computed, the affected heightmaps are put together and sent to the A* algorithm. Deciding what parts of the landscape to include in the search is a nontrivial decision. Since a road could span a large part of the game world, the number of affected meshes could be substantial. A large number of meshes contain so much information it would affect both memory usage and execution time.

Simultaneously if too little is included, the road might be forced to take a sub-optimal path through part of the landscape since the algorithm does not have enough information to find a way around the obstacle. The compromise implemented is shown in figure 4.17. A rectangular bounding box was drawn from the mesh of the start coordinate to the mesh with the goal. While the road might be forced to take a sub-optimal path with this implementation, the majority of times the result were satisfactory.

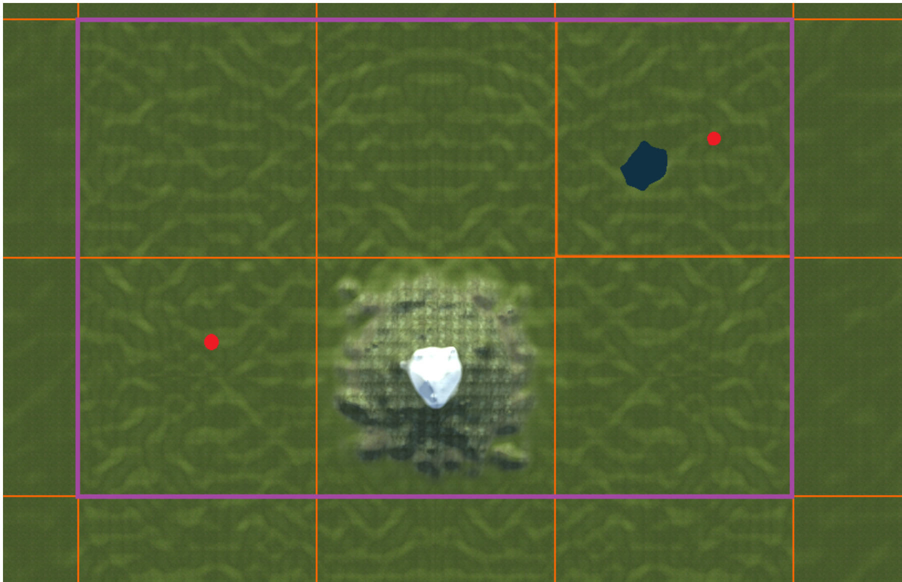


Figure 4.17: Meshes used when computing the path from one spot to another

The next part of the algorithm ensures new roads are made visible in the game world. First, the algorithm fetches a dictionary with all road segments in the current chunk. The same list of array structure as mentioned earlier, however now it only contains a list of arrays of roads in the current chunk and the *key* is the unique road id. Then for all roads present in the dictionary, but not previously created a gameobject for, steps are taken. First Bezier curves are created for the road to appear more realistic, as will be elaborated upon in section 4.4.5. Then a mesh is created for the road and finally all Unity properties like

instancing a GameObject and so forth is done. The PlaneChunk is set as the parent for the road for it to benefit the same way as the other components.

As the paths of all roads are calculated while the game is being played, the path had to be calculated as quick as possible. The first implementation of A* utilized the native *List* structure as well as the native *Sort()* and *CompareTo()* methods. The result of which caused the game to noticeable slow down even when small roads were created. If attempts to create larger roads, spanning several meshes as the one proposed in figure 4.17, the game would freeze for a long time.

The road proposed in that figure spans a total of six meshes and as such the graph for A* to traverse is a 751x501 sized graph. The algorithm, with eight directions of movement and height cost as introduced in the next section, were unsatisfactory slow. At the end of the execution, the final path had a total of 632 nodes, but the closed list consisted of approximately 114.000 nodes. On average the time required to sort the open list over 100.000 times in order to generate that path was just over 1 minute and 5 seconds. Despite optimization through a better heuristic, as will be discussed in section 5.2, another improvement had to be done.

As the primary bottleneck in the A* implementation came from sorting a list for each new node explored, this part of the algorithm had to be improved. The solution crafted was to implement a custom *heap* structure as explained in section 2.4.6. The major benefit of a heap is how fast it is to sort the elements to ensure the top node is the next node to explore. This is done many times with A* an as such there was a drastic improvement in execution time. In fact, the same path now only took just under half a second to compute on average.

The times of both list and heap implementation is shown in table 4.1. The entire algorithm was identical apart from the method to store and sort the nodes.

Table 4.1: Runtime of A*

Iteration	List	Heap
1	01:05.5460391	00:00.4991450
2	01:05.5607871	00:00.4946141
3	01:05.4853817	00:00.4944727
4	01:04.7960551	00:00.5105309
5	01:05.6756614	00:00.4987889
6	01:03.7897712	00:00.4913272
7	01:05.1062002	00:00.5168159
8	01:04.9826804	00:00.4775795
9	01:05.1495346	00:00.4903011
10	01:05.6332940	00:00.5016448
Avg.	01:05.1725	00:00.4975

4.4.3 The Impact of the Cost Function

Even with the same algorithm, and other details as described earlier, the road could be vastly different. To better study how to generate a natural looking road through a pathfinding algorithm, several different cost functions were implemented.

There are two main aspects of the A* algorithm that is being modified. The first aspects are which nodes are considered *neighbors*. This ties in to which directions, and possibly how far, the search algorithm could move each iteration. The second part is how to handle height differences in the terrain.

The classic grid search utilizes four directions of movement. Which metric used for computing distance traveled in such an application is equivalent as they all yields the same cost. The primary advantage of such a design is its simplicity and the speed of computation. Since no diagonal movement was required, the Manhattan distance was used as its computationally faster. An example of a road created with four directions of movement is shown in figure 4.18. As is quite clear, the appearance is far from realistic, even when the path is attempted to be smoothed out. At the root of the problem is the lack of diagonal movement and as such, sharp turns are made by the algorithm in an attempt to get an optimal path.

(a) Simple straight road based on the points

(b) Bezier road created based on the points

Figure 4.18: Road created with 4 directions of movement

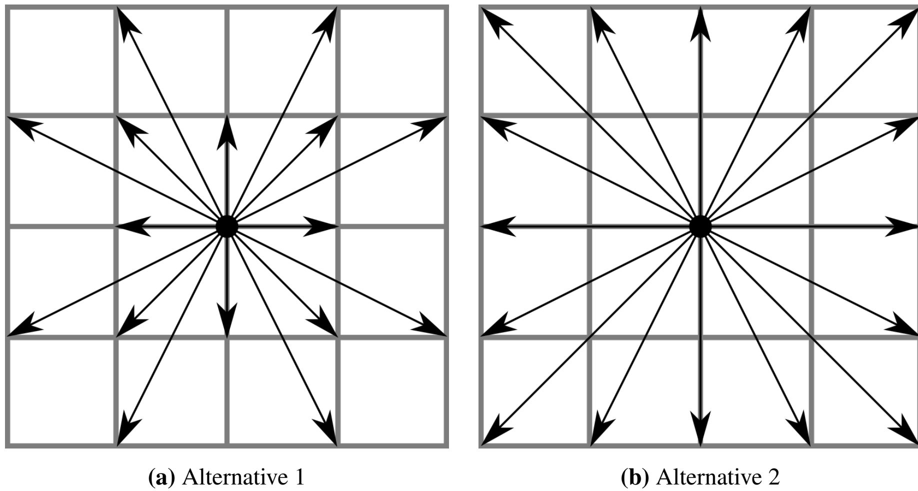
Based on those results, the number of neighbors was expanded to allow for travel in eight directions. In order to actually incentivize diagonal movement when beneficial, Manhattan distance could no longer be used, as the cost of up then right is equal to moving diagonally up-right. In an effort to reduce the computational cost of square operations, an approximation was used on the Euclidean distance. For movement directly in one direction the *cost* is set equal to 10, and for diagonal movement, the cost is set static to 14. This is based on the approximation of the Euclidian distance shown in equation 4.2.

$$Diagonal\ Euclidian\ distance = \sqrt{10^2 + 10^2} = 14.1421357... \approx 14 \tag{4.2}$$

After the introduction of the diagonal movement, the road looked significantly better. The jagged structure of the previous road is gone for a much simpler diagonal road as is shown in figure 4.19.

Figure 4.19: Road allowed diagonal movement

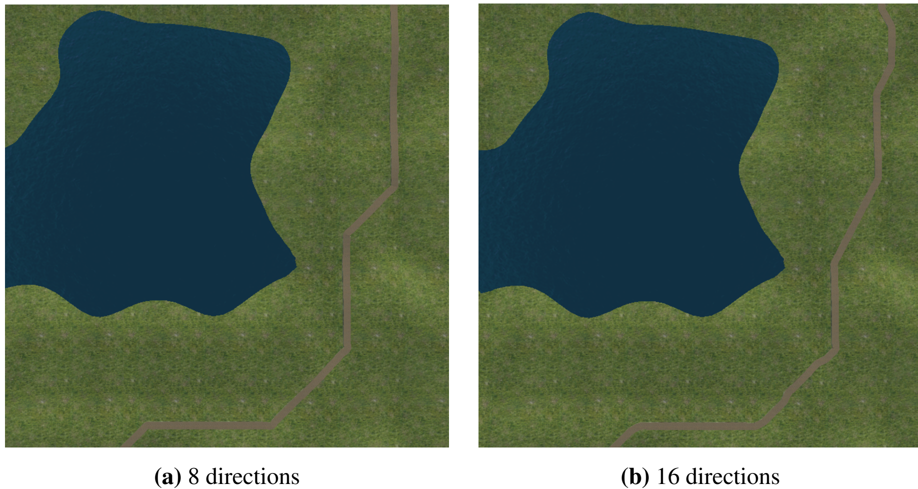
This worked quite well for flat areas without any obstacles. However, when additional components were included in the landscape, the limits of movement in only eight directions became more visible. As such another implementation with double the possible number directions were implemented in an attempt to study further how to find a smooth and natural looking road. There are two different methods of implementing the additional eight directions, as shown in figure 4.20. The latter method was chosen, where the agent always moves two tiles. This on the basis of simplicity as well as since its a road spanning a relatively large stretch, the implication of skipping every other tile would be minimal. A comparison of eight and 16 directional movements when avoiding a lake is shown in figure 4.21



(a) Alternative 1

(b) Alternative 2

Figure 4.20: Alternatives for movement in 16 directions



(a) 8 directions

(b) 16 directions

Figure 4.21: Road avoiding a lake

Skipping tiles as was done in the 16 directional cost function also has some inherent benefits. Firstly the computation necessary for finding a path is significantly reduced as the graph, and open/closed lists become significantly smaller. Additionally, this reduces the effect of small terrain variations and ensures that the road is not extremely jagged as the road is only able to turn after multiple tiles.

The obvious drawback is the loss of detail and as such a nonoptimal road. Even worse is if the pathfinding algorithm misses small lakes and rivers because it steps over them, and as such, creates an unrealistic road.

A different strategy that could be especially useful when ascending a mountain or get winding roads for some other reason is to penalize the road for turning. This would increase the unnatural parts of attempting to find the absolute most efficient path through uneven terrain, while not losing information as with skipping tiles.

The major drawback, however, is the A* algorithm does not inherently support such augmentation to the cost function. One way to include direction as a parameter was to expand each node in the graph to consist of one node for each direction, and as such, keep track of their different cost on that basis. However, this would be a major increase in the complexity of the algorithm. A more crude way is to simply increase the $g(n)$-cost based on the direction from the previous node. This breaks the fundamental rules of the A* algorithm and as such is not guaranteed to give a path even close to being optimal, or realistic for that matter. As such an attempt of penalizing turns in order to ensure longer straight stretches of road, were not attempted.

Up until this moment, all roads are constructed on relatively flat terrain and as such the cost of height differences are kept at a minimum. When the path has to traverse a mountain range or travel up to the summit, the method of penalizing height differences becomes apparent. The implemented method was to add a height penalty when computing the *g-cost* of a node. The height was not included in the heuristic in order to ensure it remains admissible; in other words, does not overestimate. The added cost of height cannot be based directly on the vertical difference, as shown in figure 4.22.

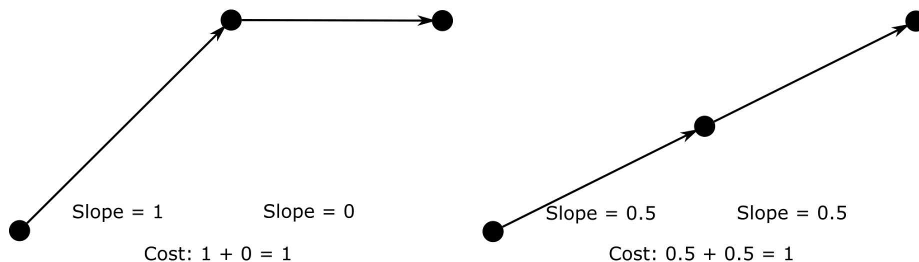


Figure 4.22: Height penalty purely based on difference in elevation

Such an added cost would only penalize the total vertical movement, regardless of the horizontal. In such a case, the algorithm would try to avoid any vertical movement, and if it had to move vertically, the steepness was irrelevant. Instead, the implemented height cost was based on the squared height difference, as shown in figure 4.23. This ensured the algorithm allowed for changes in elevation as long as the slopes were not too steep.

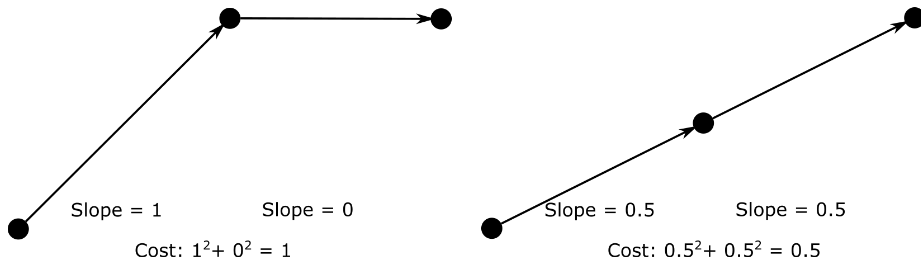
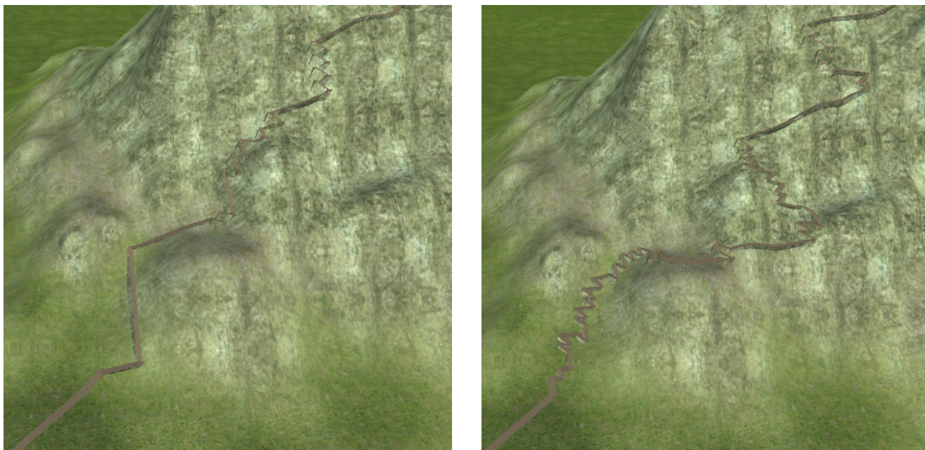


Figure 4.23: Height penalty based on the slope

Additionally, the slope value computed were multiplied by a weight to determine the cost of the difference in elevation. Several different weights were implemented and compared, and two examples are shown in figure 4.24. Increased penalty on slopes meant the roads took a more winding path up the mountainside. A universal correct weight is hard to determine, if even possible. As the terrain is generated procedurally, it is impossible to account for all eventualities. However, a further discussion on possible improvements are presented in section 5.2.



(a) Path with a low weight

(b) Path with a high weight

Figure 4.24: Different paths when the weight on height is changed

4.4.4 Saving or Replicating the Roads

The ability to replicate the entire landscape on a macro level was an integral part of the intended implementation. The same concept could also be applied on a micro level. When the player is far from away from a PlaneChunk containing a road both the chunk and road is obviously not visible. However, both components still exist in memory. Based on

the fast computation of new roads, a short study was performed in order to explore the possibility of saving less and instead generate new if or when needed.

As the highways stretched far across several chunks, the focus was put on the smaller country roads. They were mostly contained within a single chunk and higher probability of the player moving far away from. There are two different levels of deleting and recreating the road. The first is simply just to remove the Bézier path and meshes, the top layer in the pyramid. The second is actually to remove the path generated by A* algorithm and as such go back to where the game was when first started. There are two aspects that have to be explored for both of the approaches, first and foremost the time it takes to recreate the road, and secondly the memory usage by storing it.

The road used for comparison stretches diagonally across almost an entire mesh, with no obstacles. The average time for A* to find this path was 00.0075 seconds, and as such more than fast enough for the player not to notice. Additionally, some post-processing had to be done to ensure the road was within a single mesh, and if not split it into correct parts. As such, the entire process of creating, splitting and storing a new A* path was approximately 0.03 seconds.

The process of creating Bézier path and meshes were on average about 0.02 seconds and as such the entire process for generating a road was done in less than 0.05 seconds, more than acceptable to happen in the background when playing the game.

The benefit must also be linked to the actual reduction of memory usage. The exact space required for each component is challenging to analyze, but with the Unity Analyzer, some estimates are possible. The entire road mesh spanned across a single chunk uses approximately 1 MB. The mesh consists of multiple vertices per point along the Bézier path, and as such, it is quite logical that the underlying data-structures uses even less memory. As such, it is a choice regarding whether the reduction of memory is worth the computation of having to redo it later.

4.4.5 Visualization of the Road

In order to visualize the road in the game in a natural way, several steps had to be taken, the most basic of which is to choose *how* to visualize it. There were two primary options, either create a new road mesh or ”paint” the road on the existing mesh through a texture. Despite requiring more computer power, the former is the most common option. The road as a separate mesh, and GameObject in Unity, as a result, allow for greater control. The object could be scaled, rotated, altered resolution, or even hidden in the game world, which under given circumstances, could be beneficial. On top of this, by having it at as a separate mesh, it is much easier to determine if a player is located on the road or beside it. Such information is essential in many games when determining how fast a character, car or similar is able to move. Even though no such features are to be implemented in this experiment, the roads will be separate meshes as this is the industry norm.

Figure 4.25: Cross section of the road mesh on the surface

The road mesh has two distinct parts. It is the surface road mesh that the player is supposed to walk on. This is the only mesh affected by what type of road is created. Additionally, there are side meshes on both side, almost like side skirts to the road. They ensured the road got a realistic appearance of a 3-dimensional dirt road. Additionally, it ensured at no point in the game would the player be able to observe a floating road because of slight differences in the elevation of the terrain.

The road had to be textured in some way for it to appear realistic. For the road mesh on top, a texture created and made available for free on a blog was utilized [36], the texture itself is shown in figure 4.32. The side meshes had to be textured as well. One strategy is to give them a sand-like texture and as such, make the entire road appear like it is made from sand and dirt. This could give quite a realistic appearance; however, in the experiment, it gave some unwanted side effects. The road did not always perfectly followed the terrain, and as such, the side skirt could be rather large. Especially when a road was going up a mountain, the side meshes became too large for a realistic appearance of dirt.

The other strategy was only to make the top mesh look like dirt, while the side to appear as part of the terrain. This ensured that the gap between terrain and road became quite harder to spot. Based on the earlier mentioned problem with the sides becoming rather large at times, this was the implemented solution. For the majority of situations, this texturing gave a natural appearance.

(a) Path computed by A*

(b) Straight road mesh between points

Figure 4.26: Road based directly on points from A*

The path created by the A* algorithm was just a set of points the road was supposed to pass through, as shown in figure 4.26a. The simplest method of creating a road mesh was simply for each point in the calculated path create two vertices. Both spaced out form the point based on the direction of the road as to ensure the width of the road is consistent. The width is determined based on what type of road it is, highway or country road. Such a simple road mesh is observed in figure 4.26b.

The road, however, appears less than realistic. The straight road segments with sharp corners are rarely seen outside of cities in real life. There are multiple techniques for path smoothing utilized in different applications. Several curve fitting algorithm could be utilized in an attempt to smooth the entire path of the road. This approach, however, had some inherent problems that made it less than ideal — the path A* had chosen where the optimal allowed path through the terrain. With too much smoothing, the road could traverse less than ideal terrain or even worse cross obstacles that it should not cross.

Instead, the chosen design was to ensure the road crosses every point decided by the pathfinding algorithm, but interpolate more points between them shaped as curves. In this experiment, this was done through Bézier curves and interpolation of evenly spaced points along the new Bézier path.

Each point in the original path was set as an anchor point, and two control point was created for each anchor point, apart from the first and last as they only required one control point. All control points were placed based on the direction of the road with a predefined offset. Examples of how the control points were set with respect to the anchor points are shown in figure 2.9 from chapter 2. All control points were always located directly opposite another with the anchor node in the middle. This ensured a smooth turn through the corners. A Bézier road based on the same original path as figure 4.26 is shown in figure 4.27

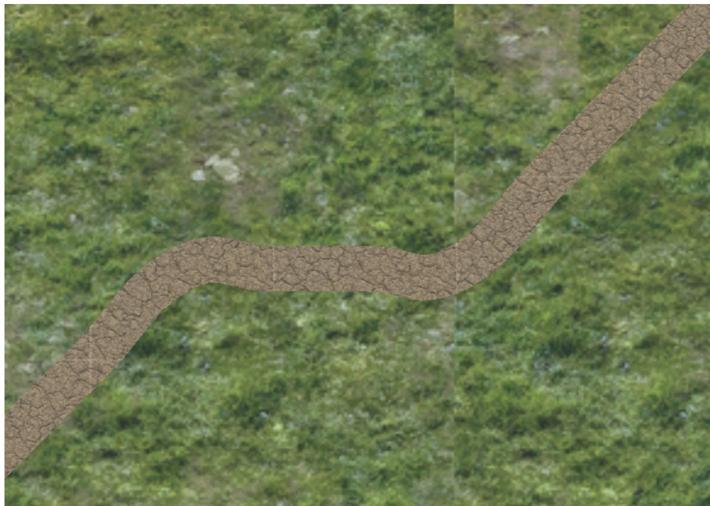


Figure 4.27: Road mesh based on Bézier curves

Steps were also taken in the direction of augmenting the terrain to fit the road. A part of all road construction in the real world is to flatten some part of the terrain to make an even foundation for the road itself to reside upon. In this experiment, the height surrounding each road was altered based on a radial basis function, as were explained in section 2.3.4. This process ensured a flat surface for the road to be *built* on top of.

Comparison with and without terrain augmentation is shown in figure 4.28 and 4.29. As is observed, the Radial basis function worked quite well for shaping the terrain to fit the road. Since the resolution of the mesh still was relatively low, the ϵ had to be high as not to affect a large area surrounding the road.

The low number of vertices that could be affected by the road also caused problems for the road when it performed sharp turns. If the algorithm tried to flatten the terrain under a winding road up a hill, there could be one part of the road attempting to raise the terrain while another tried lowering it. An example of this is shown in figure 4.30. Further discussion on how to improve this process is done in section 5.2.

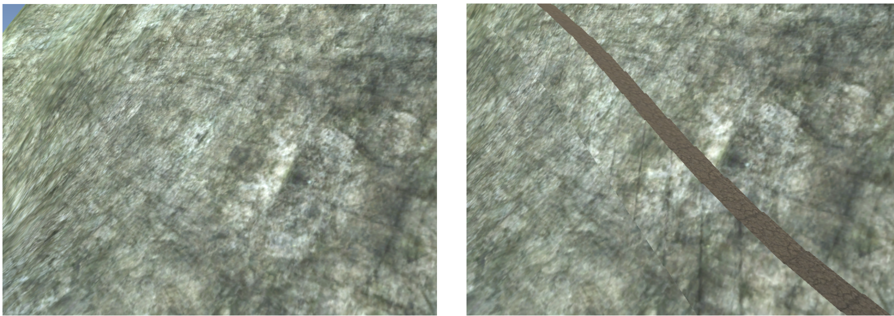


Figure 4.28: Original terrain and road

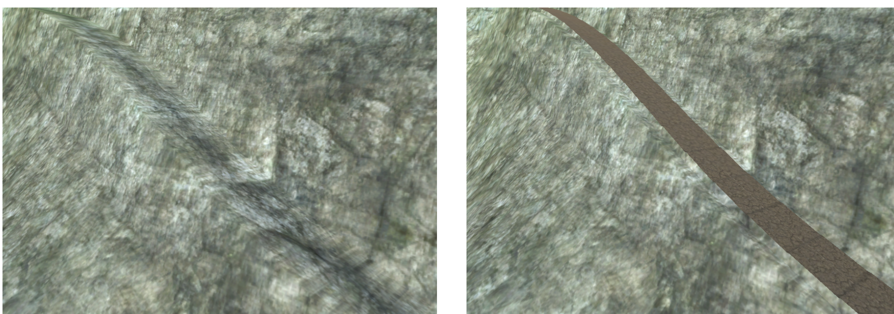


Figure 4.29: Road and terrain after augmentation

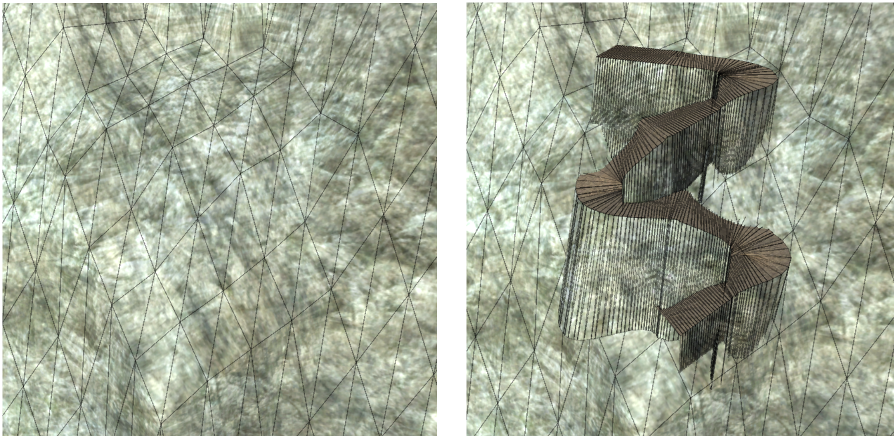


Figure 4.30: The problem with RBF on terrain with low resolution

4.5 Player Character

For a playable *Proof of concept* to be fully implemented, a character had to be included. While not an integral part of the experiment, a realistic player model is essential for the experiment as a whole. Unity in itself is rarely used for such modeling so a third party software like Blender would probably have to be utilized. While this is certainly possible, it would claim a significant part of the available time in order to model and animate a humanoid character. Instead, a prefabricated model from the Unity Asset Store was downloaded and implemented in the game. The model of a knight created by the Unity user *3DMaesen* is shown in figure 4.31 and is available at [35].

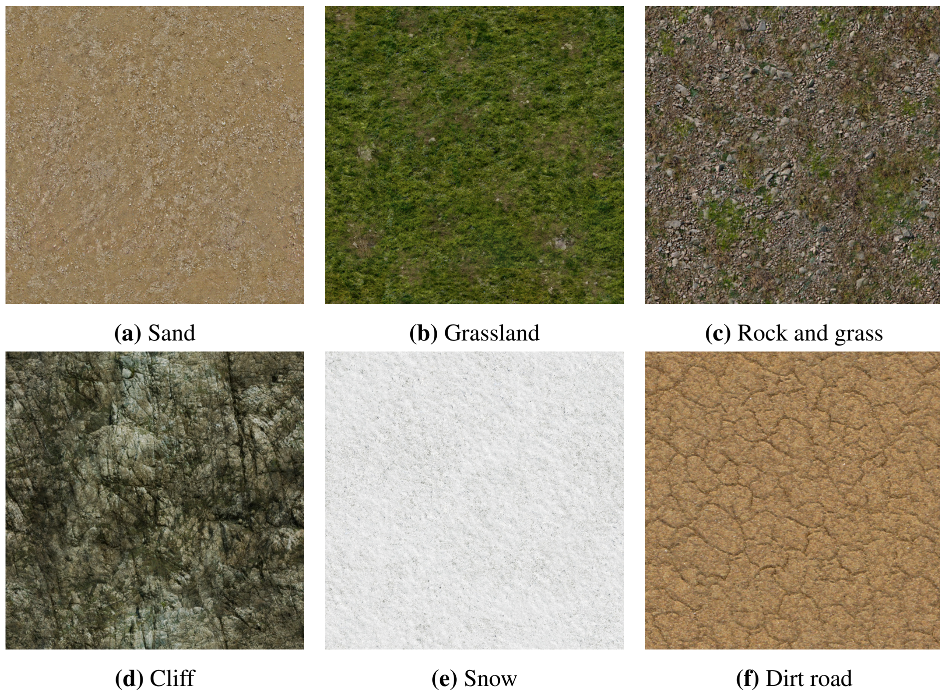
This prefabricated model did, however, not include any animations or scripts for realistic movement. As such, this had to be created before the experiment would be complete. As with the model itself, it is certainly possible, but difficult and time-consuming to create from scratch. Instead, another approach was chosen. The previously used Unity Standard Assets package [33] included several models with movement. This included the likes of cars, spaceships, but most importantly, a controllable humanoid model with animations. This character created by Unity is the one shown in figure 2.4 in chapter 2.

As such, the character control scripts from this futuristic looking model were transferred to the knight model. Only minor additional twerking and customization was required in order to have a fully controllable third person knight in the game. The movement animations of the knight are not entirely accurate as to how a person would move in full armor but is more than sufficient for this proof of concept.

Figure 4.31: Model of the knight implemented

4.6 Texturing of the Landscape

For any game, the texturing of the environment is essential to the overall appearance and as such experience. A major difficulty when texturing a terrain that is generated procedurally is the difficulty to anticipate different natural occurrences. Unless the textures themselves are also procedurally generated in runtime, there has to be a discrete set of textures included ahead of time. This defined number of texture has to be utilized in different ways across the entire continent for every sort of terrain and natural phenomena. For this project, five distinct textures are utilized for different areas of the terrain. Images were used as a base for the textures as they create the easiest and most realistic terrain. The different textures are shown in figure 2.4. All images were included in the Standard Asset bundle from Unity, apart from the snow and road image. Those are taken from a blog that creates seamless image textures used for all sorts of applications [36].

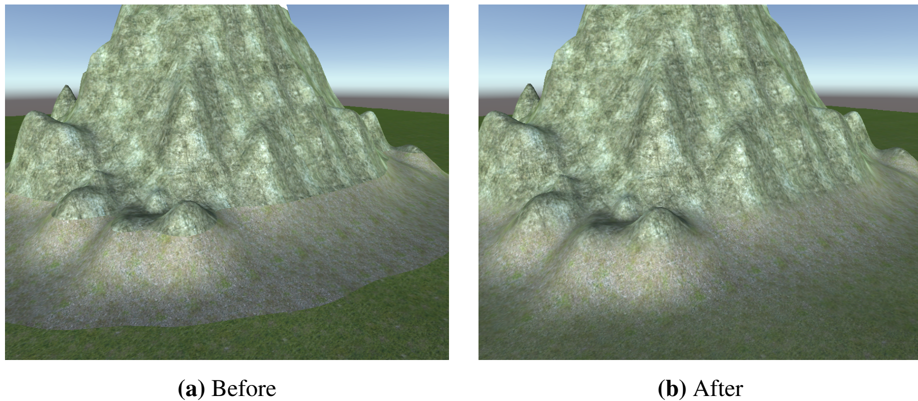


(a) Sand **(b)** Grassland **(c)** Rock and grass

(d) Cliff **(e)** Snow **(f)** Dirt road

Figure 4.32: Different image textures implemented

All these textures were combined in a custom shader written in Cg, C for graphics. Each represents a height-segment and as such span from the sand on a beach to snow on top of a mountain. Triplanar texturing was utilized, as explained in section 2.2.3. This ensured a realistic appearance of both mountainsides and the flat grassland surrounding it with the same shader.



(a) Before **(b)** After

Figure 4.33: Difference when interpolating between textures

A simple shader with image textures based on height is shown in figure 4.33a. However, the sharp edges between textures are not ideal. To levitate these unnatural transitions between terrain types the shader was somewhat augmented. Instead of setting a strict border between the different terrain images shown earlier, smooth transitions were implemented. The amount of each texture was computed through linear interpolation in the border region. This ensured a gradual transition between the grassland and the mountain, as shown in figure 4.33b.

While such an implementation certainly fulfills its intended purpose, it is not very robust. If for instance there is an altered elevation of the landscape and as such the foot of a mountain is not at sea level. Even with relatively small alteration, the transition between grassland and mountainside could end up in quite an unnatural spot. This is illustrated in figure 4.34.

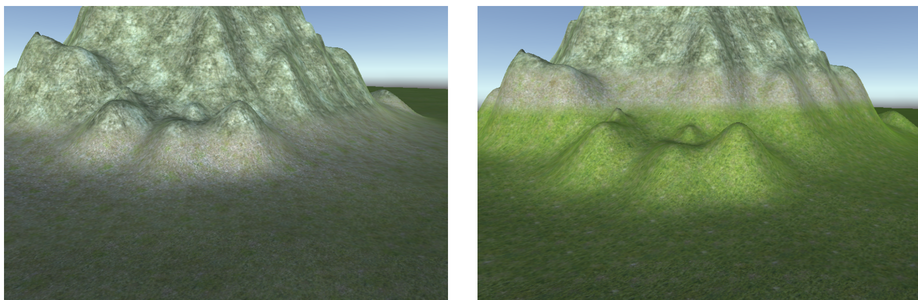


Figure 4.34: Problem with texturing solely based on height

Based on this premise of increased robustness, the shader was augmented once more. Instead of pure *height*-based texturing, the shader was to include the *slope* as well. The intensity of the textures became a weighted variable based on height and slope, and whether the texture encouraged slopes or not. For textures like in figure 4.32c the intensity was

increased with a steeper slope, while the opposite is true for the snow on top of the mountains.

No one solution is the perfect shader for procedurally generated terrain. Likewise, there are always ways to improve or fine-tune the details of such a shader implemented. Aspects as where to set the height segment for each texture and how hard to penalize slopes always has an element of empirical tuning to them. Nonetheless, the important aspect of this implementation is to create a robust shader that clearly puts across the correct visualization of the underlying terrain mesh. The final terrain visualized with the aforementioned shader is shown in figure 4.35.

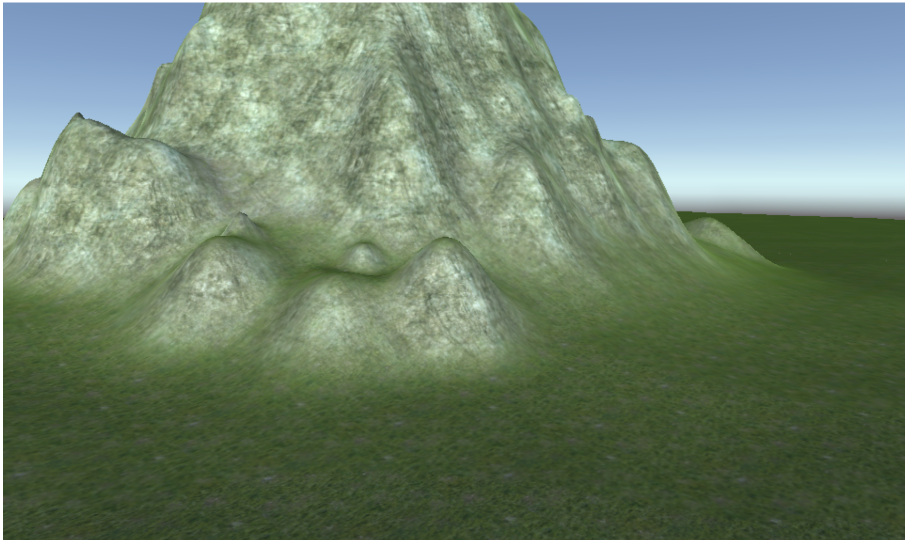


Figure 4.35: Texturing based on height and slope

Chapter 5

Evaluation

The implementation and experimentation throughout the creation of a *proof of concept* resulted in a better understanding of procedural generation. As the experiments carried out primarily has to be evaluated based on visual appearance and personal opinion about realism, a universal agreed upon solution is improbable. Nonetheless, below are sketched possible improvements as well as some ideas on other implementations that could work.

5.1 Mountains

5.1.1 Additional Mountains

The overall result of combining multiple mountains was rather realistic. Despite in its essence, just being unique combinations of a Gaussian curve, a vast and varied landscape was created. The study could be taken even further by attempting even more unique combinations of different mountains. Further tuning of size, radius height, and degree of overlapping could easily have yielded even better results.

The usage of blue-noise sampling ensured quite even spread of sample point across the domain. However, all points were of the same size. When utilized for sampling, this is usually not an issue; however, for the spread of mountains, this is not ideal. One method to counteract this effect is to adjust the radius of the components added slightly. Through only a minor adjustment of the radius, a more varied landscape is achievable. The obvious drawback of such a solution is too much or little overlap, and as such, this would not be ideal or scaleable for that matter.

The ideal solution would be to create a blue-noise sampling algorithm that inherently allowed for different sample radius. This, in turn, could lead to another study into the effect

of different radius and ideal ranges with respect to the terrain or primary mountains. As long as the execution time is kept to a minimum, such an improvement would be excellent.

Another significant possible improvement to the implementation would be how to handle procedurally generated mountains. With the current implementation, they are all stored in their entirety, with all metadata. For most mountains, this is not necessary. The data of all primary mountains, as well as others that significantly affect the landscape, could be useful. However, the many smaller mountains shaping the countryside or slope of a mountain are not needed after the terrain is augmented. After that process is done, as long as the mountain does not span into an unexplored mesh, the instance will never be called upon again.

A proposed improvement would be to either immediately delete each instance that will not serve a purpose again. This would require a small increase in computational time, but could greatly reduce the memory usage. Another approach would be to augment the heightmap directly and not even save the mountain instance. The latter approach will need to handle mountains spanning multiple meshes. If that is done correctly, the improved implementation will perform significantly better than the current version.

Even without the aforementioned improvements, the advantages of components were apparent. The visual appearance was just as realistic as traditional noise based methods, just with the added metadata provided through components.

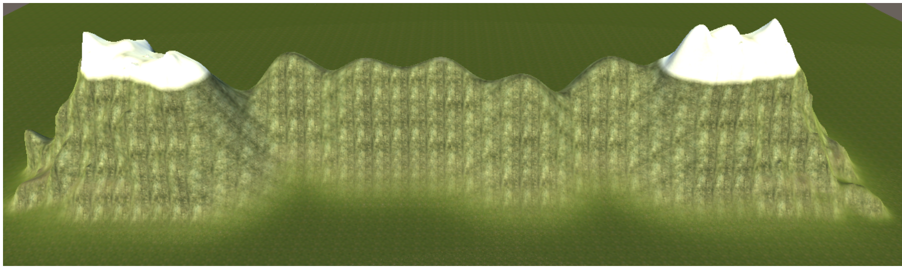
5.1.2 Mountain Ranges

The best method for populating a mountain range is quite subjective and dependent on the intended use of the result. Blue noise sampling generated far more natural placement compared with linear interpolation. This improved appearance comes at the expense of certainty of filling the entire gap. Depending on the intended use, this might or might not be acceptable. If limited to the approaches presented and explored in this thesis, linear interpolation is the only solution that guarantees no gaps in the mountain range.

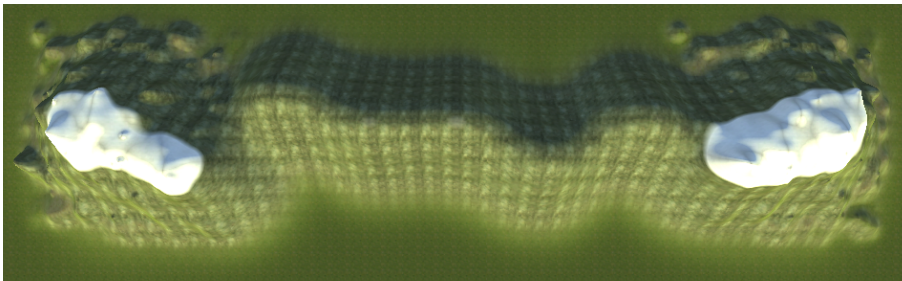
There are still steps that could be taken in order to improve the result of linear interpolation. A simple solution is to introduce a small random displacement of each new mountain. The displacement could be both back and forth on the interpolated line between the two primary mountains, and in the offset direction used with blue-noise sampling. The range of displacement is possible to control and as such, still, ensure the new mountains are not too far apart. There would have to be conducted a more thorough study into how much displacement and how it is best performed before a conclusion could be drawn. Still, a simple experiment with linear interpolation and a small offset made the mountain range more realistic. A mountain range with such an offset is shown in figure 5.1, and could be compared to the mountain ranges shown in figure 4.6 and 4.8 from chapter 4.

Regardless of the implemented strategy for the creation of a mountain range, it becomes substantially more realistic with the inclusion of smaller mountains. A mountain range

where all of the new primary mountains were given additional mountains to break their smoothness is shown in figure 5.2.



(a) Seen from the side



(b) Seen from the top down

Figure 5.1: Mountain range based on linear interpolation with offset to each new mountain

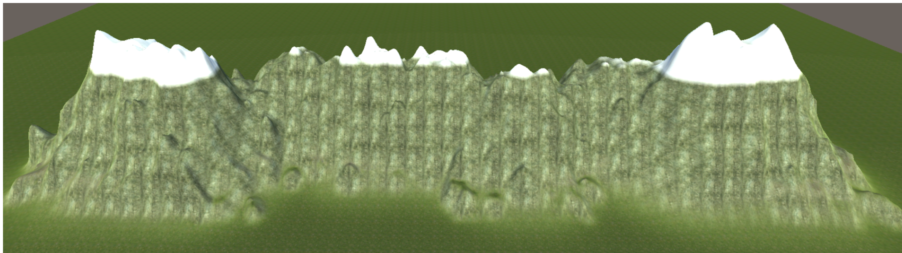


Figure 5.2: Mountain range covered in additional smaller mountains

5.1.3 Limits of Bell-Curves

While the two-dimensional Gaussian function was able to create the foundation for a natural looking mountain, it has its inherent limits. Even with additional smaller mountains scattered across the slopes, there was still a distinct symmetry. Additionally, it suffers the same major drawback as most procedurally generated terrain, the lack of natural rock shelters.

The visualization of such natural phenomena is quite tricky in procedural generation. The core of the problem becomes apparent when attempting to incorporate such phenomena in a heightmap. Two different vertices are supposed to be located at the exact same position in the xz-dimension but at a different height. Further, the polygons of that part of the mesh are supposed to face in a different direction than the others. Finally, the number of vertices required in a single mesh becomes unpredictable. This is especially problematic since Unity have a hard limit on the number of vertices in each mesh.

Due to the difficulties presented, such features are rarely seen in procedural generation. Techniques and algorithms on how to visualize such features without any intervention from a human designer could be a thesis on its own. Nonetheless, component-based, as opposed to noise based, procedural generation has some distinct advantages. Since most parameters are known, limits could be placed on such features. Additionally, the size of caverns or rock shelters would be known parameters for the algorithm creating meshes.

Based on those aspects, components could be a significant advantage in the future when including such natural phenomena. However, for any conclusion to be drawn, significantly more experiments and research has to be performed.

5.2 Roads

5.2.1 Use of a Traditional Search Algorithm

The basis for how the path of the road was decided in this experiment was through a search algorithm. The choice fell on A* as the other pathfinding algorithms studied were not optimized for the use intended in this experiment. HPA* and especially D* is popular among game developers today, but their demands are different. The majority of times a path is needed in a video game it is for some *agent* to move. Either as troops in strategy games or cars in racing, the path has to be calculated quickly and might need to be updated as quick.

Despite the use of an older algorithm, the execution time was not impacted significantly. The roads could be generated, and possibly regenerated, within what felt like the acceptable time when testing the implementation. As such, the limit was not with the execution time of the algorithm, but rather with the result. The only persistent drawback of the search was the requirement of complete knowledge of the terrain before any search could be performed.

In an ideal situation, the general shape of the path could be computed first. Then the details of the smaller bends in the road could be calculated when the player approached. Such an algorithm could, in theory, utilize the best aspects of both HPA* and D*. The general shape of the road could be planned like a high-level cluster from HPA*. Landscape consisting of components could be utilized to great effect in such a hypothetical scenario. High-level pathfinding only based its path on the location, and size, of components such as mountains and lakes. Then the exact path across the different clusters are gradually improved with increased knowledge as the player explores the world.

Alternative approaches without the use of a traditional search are, in theory, also a possibility. A direct line between the cities could be the start and then force the road in a left or right arc depending on components located in its path. This is a new solution to combat the difficulties mentioned above with a search algorithm. However, the process requires there to be a limited number of components or be able to traverse some components at a cost. With an increasing number of components or complexities in the landscape, such an approach would probably either result in quite an unrealistic path or be just as computationally demanding as a traditional grid search.

5.2.2 Possible Improvements to the Current Search

Without a custom HPA* search algorithm as proposed earlier, there are still several possible improvements to the implemented A* algorithm. The first of which is the possibility to include the height as part of the heuristic. The best possible slope form the current node to the goal should ensure the heuristic still does not overestimates. With such an improvement, there might be possible to reduce the number of nodes that had to be explored. Additional metadata could also be incorporated as part of the search. Since all major aspect of the terrain is based on components, their metadata could be utilized. Despite not directly affect the cost function, this information could be used to determine what meshes to include.

The current solution always fetches the same part of the terrain, without any knowledge of what sort of terrain is present. Since the scripts technically know ahead of time whether it is smooth flat grassland, lakes or even a mountain in the mesh, the decision of which to include should not be straightforward. For instance, if a direct path has to cross a large lake in the current implementation, a road might not be possible. With an improved method of determining which mesh to include, then maybe several additional chunks are included in the search, and as such a feasible path is found.

Alterations to the search could also be done based on the type of road. While small country roads, or alleyways, have to take even the smallest height differences into account, a highway does not. A highway from one city to another would realistically continue straight for large sections, regardless of smaller height alterations. This feature should ideally also be visible in a procedurally generated road. There are several methods that could be used to achieve this.

One method is to have different slope penalties on the different types of road. This will ensure some roads follow the terrain closer than others. Skipping tiles when creating large highways is another method. The latter approach would both ensure larger straight stretches and reduce the complexity of the search. A greater step-length would additionally allow for even more discrete direction of movement as a bonus. The challenge would be to determine the correct number as not to overshoot obstacles or take an unnatural path because of the lack of detail. Nonetheless, the approach of differentiating the search based on both the components in the landscape and the type of road-generated is probably advantageous.

5.2.3 Replication of Roads

A* is ensured to find the optimal path, and as a result, correct replication of all roads are ensured. Based on this premise, a study on recreating the road when a player left the area, as opposed to saving it, were performed.

Smaller roads were fast to create and as a result, ideal for recreating instead of saving. The entire process took, on average 0.05 seconds and over half the time was spent on generating the path. If only the larges files were deleted, namely the Bézier path and mesh, creating the road required approximately 0.02 seconds.

At first glance, this sounds like an excellent opportunity for optimization. However, it is not that simple. Modern games push the frame rate up to 60 Hz or more. Between each frame, there are several computations that need to be performed and recreating a road can not take priority. Previously limited memory was a driving force for procedural generation; this is not the case anymore [3]. As such, replication of a road more times than necessary will most likely do more harm than good.

A better optimization strategy would most likely be keeping the objects in memory, but not visualizing them on screen. Such an approach would allow the GPU to focus solely on essential elements to visualize. Further, the creation of roads is ensured not to interfere with the more critical tasks on the CPU.

Roads could still be deleted when the player is sufficiently far away from them that it will probably be quite some time until they are within reach again. Such scenarios, roads, and most other elements should be deleted to reduce memory usage. The appropriate analogy to non-procedural generated games is when those games drop them from memory and instead fetched them from a hard drive if needed again.

5.2.4 Improved Visual Appearance

While the visual appearance of the roads was far from an essential aspect of the study, some fundamental discoveries were made. The most important was that post-processing of roads would have to be performed for anything close to a natural appearance. Despite 16 directions of movement, the road looked quite bulky without any post-processing.

In this experiment, Bézier curves were utilized to great effect, but for a natural appearance, this might not be enough. In the current implementation, curves were generated between all of the points, and as such, ensuring the road passed through everyone. While this assured the road did not cross any obstacles and actually followed the optimal path, the corners were incredibly sharp. An improvement would be to increase the turn radius of the bends in the roads. If the curves were calculated based on several points on the path instead, the turn would be more gradual and as such, more natural. For obvious reasons, the process would still need to assure the road did not cross any illegal boundaries, but if that is done correctly, the visual result would be greatly improved.

Similarly, one of several path smoothing techniques could be used on the original path as

well. Even a simple algorithm that only took the edge off some corners would help to accomplish a more natural flow of the road. Additionally, such a technique would make the road appear to have taken a significantly higher number of discrete direction as the ensuing path is not directly in one of a set of predefined directions.

A radial basis function augmented the landscape to fit the road better. Based on the empirical results gathered from this experiment, augmentation of the terrain is essential for a natural appearance. Everywhere in the real world where either humans or animals frequently walk the terrain is affected. Especially when roads are constructed, like in this experiment, the ground has to be shaped in accordance with the road. Radial basis function, as used with this thesis, worked great within limits set elsewhere in the project. Due to the low resolution of the terrain, few vertices could be affected by the road. For better-looking results, as well as better insight into the advantages and disadvantages of such a feature, the resolution has to be drastically increased. With additional vertices for the function augment, smoother transitions between the road base and the unaffected surroundings would be possible. Such an implementation would also mitigate, or at least significantly reduce; the problem encountered when augmenting terrain beneath a winding road going uphill.

With the increased effort put into augmenting terrain and realistic visualization of the roads, intersections could also be explored. In the current implementation, no effort has been put into realistic modeling of crossroads. Albeit if the terrain is flat, the two road meshes will intersect at the same height, and as such not appear too out of place. However, with a difference in elevation or other nonideal situations, the result will be far from realistic. With the increased effort put on augmenting the terrain surrounding each road, the additional effort should be put on augmenting both the roads and terrain to more realistically portray crossroads and other phenomena along the road.

5.3 The Entire Experiment

5.3.1 Limited Number of Components

The landscape presented in this thesis was constructed solely based on mountains and lakes shaped as smooth bell-curves. While the results are impressive with this limiting factor, the landscape is still somewhat empty. Even though the mountains and lakes were able to create fundamental groundwork, additional components are still necessary.

The mountains only generate additional height to the landscape, but a similar process could be used to subtract from it. Instead of a mountain range, canyons could be excavated. This would break the smoothness seen elsewhere in the generated landscape and become exciting areas to explore. Additionally, a canyon on one side of an existing mountain would break the symmetry.

The basic principle found in the current implementation could also be taken a step further. A large island could, in theory, be based on a single large mountain with ocean surrounding it. In similar fashion rivers from various lakes could be generated through a similar process

as roads. Since the overall shape of the island is based on the mountain component, the overall curvature is known. This allows the river-algorithm to know which general direction to flow. Through A* or a similar algorithm, a natural path based on several parameters could be found in the terrain. With some post-processing of the path and augmentation of the terrain, a realistic river could flow across the landscape.

If the river approach were taken a step further, the simulation of hydraulic erosion could be incorporated. Either to augment the mountain and overall terrain or to improve the path of a river, hydraulic erosion could significantly increase the realism. The process is quite expensive computationally and as such might not be ideal for augmentation of terrain while a player is supposed to explore the world. Instead, it might have to be done during a more prolonged initialization phase of the game. Such a setup process would also allow for a large extent of other improvements to be performed. On the other hand, ongoing research on hydraulic erosion through GPU is showing great promise, and it might be done in real time in a not to distant future [37].

An increase in the number of components, in general, would also significantly improve the outcome. Forests are principal components that allow for a vast amount of opportunities within the domain of procedural generation. Each tree could be created differently. Parameters as height, number, and size of branches could all be altered and as such, generate an incredibly varied forest. A simpler alternative is to base it around a number of prefabricated trees, but attempt to spread them in exciting and natural patterns. Attempts have been made to great effect with blue-noise sampling for such scenarios. The trees could be spread based on the type of tree as done in [38]. The algorithms presented in that thesis allows for both uniform and adaptive sampling and as such, could create different types of forests. Alternatively, as done in [39], the trees could be spaced both on the type and other properties such as the size of the tree.

5.3.2 Procedural Placement of Components

This entire experiment was centered around careful positioning of a few principal components ahead of time, and procedurally generated a landscape based on those. Based on the research done in this thesis, there is nothing to suggest that such a spread of components cannot be done algorithmically.

There are several methods for deciding where to put each component. A simple blue-noise approach could work through similar methods as explained in the previous section. Such an approach would have to ensure both the distances apart and the ratio between different components. With such hyperparameters adequately set up, natural spread, and as such terrain, could be achieved.

Further along those lines, the world could be split into different terrestrial areas. With each area, the *rules* for procedural placement of different objects changes to better fit the purpose of the area. An example of such a feature would be to increase the number of lakes in an area and significantly reduce the mountains for the creation of a particular landscape. Similarly, some components could be bound to others, like farmland is only able to be created within a certain vicinity of a city. Rules for where to place what types, or class,

of objects, could be a highly exciting field of research. Especially with component-based procedural generation, such algorithms would open a lot of new and exciting possibilities for the creation of game worlds.

More advanced methods could also be utilized for creating a natural landscape. Studies on how the earth was created, and how humans populated it, could be the foundation for where to located different types of terrain. The visualization of different proposals could benefit from component-based procedural generation. The relative ease of visualizing several terrain proposals would greatly benefit such research and could benefit all sort of world design processes.

Chapter 6

Discussion

6.1 Design of the Specification

With the problem described in the very start of this thesis, several different directions for a proof of concept was possible. The direction chosen focused on the exploration of different approaches and whether they were feasible or not. As a result, each general direction of exploration had to be attempted with a lot of different parameters. This made each approach quite time consuming as multiple implementations and experiment had to be carried out in order to be reasonably confident as to the feasibility of each method. The results presented in chapter 4 were representative for the general approach, but still far from the only result generated during the experiment.

While this gave great insights into different prospects of procedural generation of both terrain and roads, it is far from the only possible specification. A different approach could flip the order of events. The first component constructed could be an interesting road from point A to point B with interesting bends and elevation. Then secondly mountains and lakes could be located to adhere to the preexisting road. While the order of events is unnatural compared to the real world, it is not uncommon in the design of fictional worlds.

Since all components will be in place before any person playing a game is able to observe the area, what parts were created first is impossible to determine. The aforementioned order could better ensure a more exciting road with additional scenery. This is an interesting concept with great potential; however, it was opted out of in the experiment. Primary because of the desire to create an interesting landscape, with roads as an additional element for realism. As opposed to interesting roads to travel along, with terrain elements located beside them for an enhanced experience.

The specification implemented was centered towards two primary areas of research, moun-

tains, and roads. While the road was generated based on terrain created through components, the creation was affected minimally by this fact. Lake-components were a clear advantage for the road. Each instance was saved to memory as a combination of location and size and thus made it easy to incorporate in the search. Likewise, the cities were located at both ends of a road for natural stop points. Mountain-components, on the other hand, did not affect the search. A* utilized the heightmaps for the terrain, however, it made no difference how the heightmap was created.

Another proof of concept could attempt to create an even closer connection between the mountain-components and the road. Such research could, in theory, explore even further the possibilities components introduce, and how they could improve the prospect procedurally generated roads.

6.2 Dedicated Time to Support-Components

Within the intended specification, there was also room for decisions concerning time and effort dedicated to each part. Especially the visual appearance of several components could have been dedicated significantly more or less time.

Cities and lakes served only the secondary purpose of being either start locations or obstacles for the road as such a significant amount of time could be reduced by swapping the implemented solution with a simple *placeholder*. Such objects could be the likes of geometrical shapes native to Unity, such as boxes or cylinders. Objects such as these would break all realism, but enable a more substantial portion of time dedicated to the two primary aspects of research.

The polar opposite approach could also be attempted, an increased focus of realism on the supporting components. Cities that would transition from the heart of the city towards outskirts could drastically improve their appearance. Cities could consist of several different types of buildings and greatly help increase the realism of the world. While certainly increasing the visual appearance of the result, the aforementioned ideas would reduce the time available to the actual creation of roads.

As a whole, the time and resources dedicated to the supporting components were adequate. The objects were created as simple as possible without significant loss to their realism. This ensured a good looking demonstration, as well as functioned as further evidence that component-based generation works.

The playable character is also subject to a similar line of thought as the secondary components. The implementation includes a controllable humanoid knight that can walk, or run, in the generated landscape. Like the two components mentioned above, the knight was also based on external assets. Despite this saved a large amount of time compared to creating a custom model, the implementation still took some time. The combination of multiple external assets and integrating them with the custom features took away resources from the core area of research. Simultaneously the inclusion of a controllable humanoid character kept some focus on the primary long term goal for research in the

field, video games. A playable, realistic, model also vastly increased the possibility for realistic demonstrations on how a game based on this type of procedural generation could be created. With these aspects in mind, the time spent on controllability of the character and integration of external assets were justified.

6.3 Retrospective on the Experiment

The experiment gave additional insights into the problems faced with component-based procedural generation and the creation of roads. Based on experience gathered, some parts of the experiments carried out could have been even better. Based on the insights and evaluations presented up to this point, some changes could have benefited the experiment.

The most significant change would be a closer connection between components and the roads. At several earlier instances, ideas on how to link the components to the roads are presented. In retrospect, the components could easily be an even larger part of the search. In theory, A* could create a custom graph without the heightmap. Weight of each point in the graph could be computed directly proportional to the distance from surrounding mountains. Similarly, the lakes could be passable, but at an exponentially increasing cost closer its center. While the ensuing road probably takes a sub-optimal path, the advantages of components become clearer.

The primary focus when generating new mountains were a study on what approaches were feasible. During the experiment of each approach, a lot of different experiments were carried out. For instance, with the creation of smaller mountains based on white noise, there was a significant different attempt to generate the best possible results. Different heights, radius, area of effect and number were attempted to study if white noise gave acceptable results. During this experimentation, a second study on the ideal ratio between primary mountain and new mountains could also be performed.

While such an additional study would take some resources away from the primary goal of testing feasibility, the potential reward would probably be worth it. A study on ideal aspect ratios between different components would be beneficial for future research.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, a *proof of concept* for procedural generation has been specified and implemented. The implementation focused on two primary areas, namely how to best use mountains as primary components for creating the landscape and procedural generation of roads in the aforementioned landscape. Primary mountains were located ahead of time, and new mountains were located based on either linear interpolation or different variants of noise in order to create a realistic landscape. Supporting components as cities and lakes were also manually distributed for increased realism when constructing the road. The path of the road was found through an A* graph search based on the height in the terrain.

The study showed that mountains as components were quite suitable, both for creation of mountain ranges and mimicking the natural unevenness of real terrain. Noise-based approaches gave more exciting results than a stricter method; however, it increases the risk of unnatural phenomena occurring. Choice of method is dependent upon the application and becomes a trade-off between risk and reward the developer has to make.

Generation of roads through A* search gave realistic results. The use of lakes and cities as components meant that the road had a natural start and stop environments and was able to avoid crossing bodies of water. Slope penalty when computing the path drastically affects the path and different weighting depending on both application and type of road is probably necessary for optimal results.

The experiment as a whole shows excellent promise for component-based procedural generation. This approach gives better control and significantly more predictable results than traditional pure noise based approaches. As a consequence, the methods presented could both be used directly as the in-game map or serve as inspiration for manual designs. Regardless, procedural generation is an excellent tool for overcoming the bottleneck of con-

tent creation.

7.2 Future Work

The work presented in this thesis could be used as a foundation for several new studies in the future. Many different aspects of the work described could be taken further, improved, or otherwise altered for new experiments on the topic. Nonetheless, based on the results of the aforementioned experiments, there are a few general directions of exploration that is especially encouraged.

An area of research is an increasing number of components. Several new aspects of the landscape included, but not limited to, the aforementioned rivers, forests, and canyons. Additionally, the road could treat all type of components as traversable, but at a different cost. This would mimic the way roads are constructed in the real world with bridges and tunnels as options, but at a higher cost per distance than a regular road.

Another direction for future research is the procedural placement of components. Different methods of distribution could be attempted and valuable knowledge on how to ideally distribute aspects of the world. A system with the ability to create a large number of different terrains at random could serve both to create the game world directly or as the foundation for a designer to improve.

Bibliography

[1] Entertainment Software Association. *Essential facts: About the computer and video game industry*. 2018.

[2] Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. Procedural content generation for games: A survey. *ACM Trans. Multimedia Comput. Commun. Appl.*, 9(1):1:1–1:22, February 2013.

[3] Alba Amato. *Procedural Content Generation in the Game Industry*, pages 15–25. 03 2017.

[4] Jakob Schaal. *Procedural Terrain Generation. A Case Study from the Game Industry*, pages 133–150. 03 2017.

[5] Michael Blatz and Oliver Korn. *A Very Short History of Dynamic and Procedural Content Generation*, pages 1–13. 04 2017.

[6] William Raffe, Fabio Zambetta, and Xiaodong Li. A survey of procedural terrain generation techniques using evolutionary algorithms. pages 1–8, 06 2012.

[7] Lawrence Johnson, Georgios N. Yannakakis, and Julian Togelius. Cellular automata for real-time generation of infinite cave levels. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10, pages 10:1–10:4, New York, NY, USA, 2010. ACM.

[8] Christopher Beckham and Christopher Pal. A step towards procedural terrain generation with gans. 07 2017.

[9] Citygen - procedural city generation, 2010. [Online; accessed 11-May-2019] Available at `http://www.citygen.net`.

[10] George Kelly. H.: Citygen: An interactive system for procedural city generation. In *In Proceedings of GDTW 2007: The 5th Annual International Conference in Computer Game Design and Technology*, pages 8–16, 2007.

[11] Yoav I. H. Parish and Pascal Müller. Procedural modeling of cities. In *Proceedings*

of the 28th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '01, pages 301–308, New York, NY, USA, 2001. ACM.

[12] Kevin R. Glass, Chantelle Morkel, and Shaun D. Bangay. Duplicating road patterns in south african informal settlements using procedural techniques. In *Proceedings of the 4th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*, AFRIGRAPH '06, pages 161–169, New York, NY, USA, 2006. ACM.

[13] Eric Galin, Adrien Peytavie, Nicolas Maréchal, and Eric Guérin. Procedural generation of roads. *Comput. Graph. Forum*, 29:429–438, 2010.

[14] Øystein Brox. Specialization project - procedural terrain generation. 2018.

[15] Gabro Media. Barrles, 2016. [Online; accessed 12-December-2018] Available at `https://assetstore.unity.com/packages/3d/props/barrels-32975`.

[16] Ben Golus. Normal mapping for a triplanar shader, 2017. [Online; accessed 28-May-2019] Available at `https://medium.com/@bgolus/normal-mapping-for-a-triplanar-shader-10bf39dca05a`.

[17] Tommy Gravdahl Olav Egeland. *Modeling and Simulation for Automatic Control*. Marine Cybernetics AS, 2 edition, 6 2002.

[18] CartouCHe Cartography for Swiss Higher Education. Computer graphics - bézier curve, 2019. [Online; accessed 11-May-2019] Available at `http://www.e-cartouche.ch`.

[19] Martin D. Buhmann. *Radial Basis Functions: Theory and Implementations*. Cambridge University Press, Cambridge, 2003.

[20] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[21] Stuart Russell. *Artificial Intelligence: A Modern Approach*. Pearson International, 8 2013.

[22] Adi Botea, Martin Müller, and Jonathan Schaeffer. Near optimal hierarchical pathfinding. 2004.

[23] M. Nosrati, Ronak Karimi, and Hojat Allah Hasanvand. Investigation of the * (star) search algorithms: Characteristics, methods and approaches - ti journals. 2012.

[24] Yuhanes Liauw, Pandu Pratama, Sang Kwon Jeong, Duy Vo Hoang, and Sang Kim. Experimental comparison of a* and d* lite path planning algorithms for differential drive automated guided vehicle. *Lecture Notes in Electrical Engineering*, 282:555–564, 01 2014.

[25] David T. Wooden. *Graph-based Path Planning for Mobile Robots*. PhD thesis, 2006.

[26] Wei-Chien Cheng, Wen-Chieh Lin, and Yi-Jheng Huang. Controllable and real-time reproducible perlin noise. pages 86–97, 08 2014.

[27] Dong-Ming Yan, Jian-Wei Guo, Bin Wang, Xiao-Peng Zhang, and Peter Wonka. A survey of blue-noise sampling and its applications. *Journal of Computer Science and Technology*, 30(3):439–452, May 2015.

[28] Robert L. Cook. Stochastic sampling in computer graphics. *ACM Trans. Graph.*, 5(1):51–72, January 1986.

[29] Robert Bridson. Fast poisson disk sampling in arbitrary dimensions. In *ACM SIGGRAPH 2007 Sketches*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[30] Unity Technologies. Public relations.

[31] Miles Macklin and Matthias Müller. Position based fluids. *ACM Transactions on Graphics (TOG)*, 32(4):104, 2013.

[32] Bernhard Fleck. Real-time rendering of water in computer graphics. 04 2019.

[33] Unity Technologies Unity Asset Store. Standard assets, 2018. [Online; accessed 14-May-2019] Available at `https://assetstore.unity.com/packages/essentials/asset-packs/standard-assets-32351`.

[34] Aquarius Max. Unity asset store - castle supply lite, 2016. [Online; accessed 21-March-2019] Available at `https://assetstore.unity.com/packages/3d/environments/fantasy/castle-supply-lite-23699`.

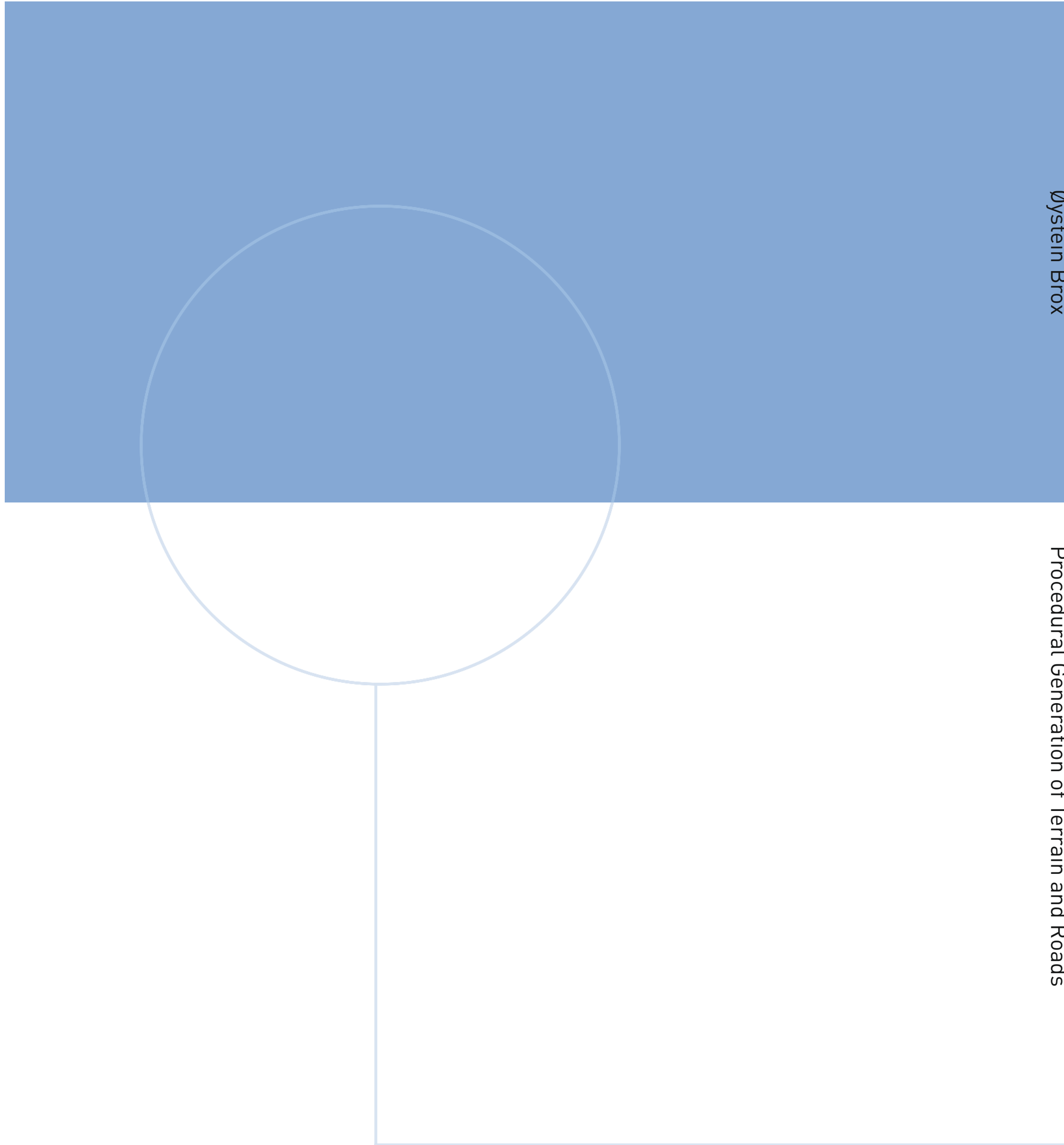
[35] 3DMaesen Unity Asset Store. Strong knight, 2018. [Online; accessed 21-March-2019] Available at `https://assetstore.unity.com/packages/3d/characters/humanoids/strong-knight-83586`.

[36] High resolution seamless textures. [Online; accessed 13-May-2019] Available at `http://seamless-pixels.blogspot.com/2012/09/free-seamless-ground-textures.html`.

[37] David Müller, Christoph Buchenau, and Michael Guthe. Partial updates to accelerate gpu-based interactive hydraulic erosion. In *Proceedings of the 32Nd Spring Conference on Computer Graphics*, SCCG '16, pages 33–40, New York, NY, USA, 2016. ACM.

[38] Li-Yi Wei. Multi-class blue noise sampling. In *ACM SIGGRAPH 2010 Papers*, SIGGRAPH '10, pages 79:1–79:8, New York, NY, USA, 2010. ACM.

[39] Jiating Chen, Xiaoyin Ge, Li-Yi Wei, Bin Wang, Yusu Wang, Huamin Wang, Yun Fei, Kang-Lai Qian, Jun-Hai Yong, and Wenping Wang. Bilateral blue noise sampling. *ACM Trans. Graph.*, 32(6):216:1–216:11, November 2013.



NTNU
Norwegian University of
Science and Technology