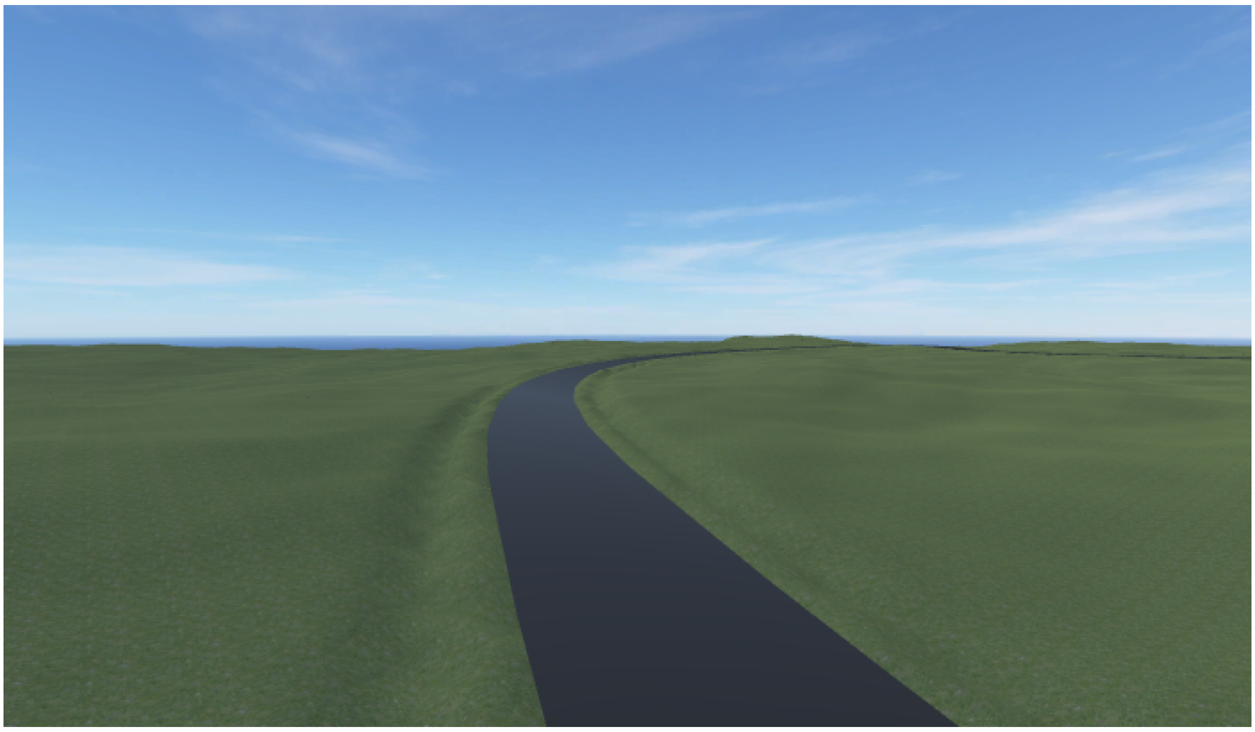
CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Procedural Generation of a 3D Terrain Model Based on a Predefined Road Mesh

Bachelor of Science Thesis in Applied Information Technology



Matilda Andersson
Kim Berger
Fredrik Burhöi Bengtsson
Bjarne Gelotte
Jonas Graul Sagdahl
Sebastian Kvarnström

Department of Applied Information Technology
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

Bachelor of Science Thesis

Procedural Generation of a 3D Terrain Model Based on a Predefined Road Mesh

Matilda Andersson
Kim Berger
Fredrik Burhöi Bengtsson
Bjarne Gelotte
Jonas Graul Sagdahl
Sebastian Kvarnström

Department of Applied Information Technology
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

Procedural Generation of a 3D Terrain Model Based on a Predefined Road Mesh

Matilda Andersson
Kim Berger
Fredrik Burhöi Bengtsson
Bjarne Gelotte
Jonas Graul Sagdahl
Sebastian Kvarnström

Supervisor: Marco Fratarcangeli, Department of Applied Information Tecnhology
Examiner: Daniel Sjölie, Department of Applied Information Tecnhology

Department of Applied Information Technology
Chalmers University of Technology
University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Images and figures are captured or created by the authors, if nothing else is stated.

Cover: An example of a procedurally generated terrain model based on a road mesh.

Typeset in LaTeX
Department of Applied Information Technology
Gothenburg, Sweden 2017

Acknowledgements

We would like to thank our supervisor, Marco Fratarcangeli, for all the support he provided during the project. We would also like to thank *Fackspråk* (*FKI*) for assisting with the writing of the report, Kristoffer Helander for arranging the cooperation with Volvo Cars, for providing ideas, and support during the course of the project.

Procedural Generation of 3D Environments Based on Predefined Road Meshes

Department of Computer Science and Engineering,
Chalmers University of Technology
University of Gothenburg

Abstract

This thesis studies procedural generation as a method for generating a realistic terrain model, that will be used as a part of a virtual environment for testing the cameras of autonomous vehicles. The project was done in collaboration with *Volvo Cars*, and resulted in software that was able to generate a terrain model with only a mesh representing a road as input according to the customer's requirements.

Specifically, the purpose of the thesis was to develop software that was able to fit the generated terrain model to a road, place ditches alongside it, make the terrain model surrounding the road resemble a hilly landscape, and decide what type of terrain each mesh should represent. The functionality to base the look of the terrain model on real world height data was also implemented.

Radial basis function interpolation was used to fit the terrain model to the road, and Perlin noise, Simplex noise, and Worley noise were used to modify the topography of the terrain model. The ditches were created by projecting parts of the terrain model to fit a cylindrical shape, and the type of terrain a mesh should represent is based on the distance between the mesh and the road.

The end result shows that it is possible to use procedural generation to create realistic terrain models, and the generated models can be used as a part of virtual testing environments. However, the software is more a proof of concept than a finished solution, and there are several areas in need of improvement such as implementing support for more terrain types, being able to use more than one noise map, and improving how the ditches interact with noise.

For future research *Volvo Cars* has expressed interest in expanding the scope of the product to allow for generation of suburban areas, as well as mountains and include support for tunnels.

Keywords: Procedural generation, Unity, Virtual Testing Environments, Radial Basis Function Interpolation, Noise Functions

Sammandrag

Denna kandidatuppsats studerar procedurell generering som en metod för att generera en realistisk terrängmodell, som kommer att användas som en del av en virtuell miljö för att testa kamerorna till självkörande fordon. Projektet genomfördes i samarbete med *Volvo Cars* och resulterade i programvara som kunde generera en terrängmodell med endast en mesh som representerar en väg som input, i enlighet med kundens krav.

Specifikt var syftet med kandidatarbetet att utveckla programvara som kunde passa den genererade terrängmodellen till en väg, placera diken bredvid vägen, få terrängmodellen att likna ett landskap med kullar, och att bestämma vilken typ av terräng varje mesh borde representera. Funktionaliteten att basera terrängmodellens utseende på riktig höjddata genomfördes också.

Radial Basis Function-interpolering användes för att passa ihop terrängmodellen till vägen, och Perlin noise, Simplex noise och Worley noise användes för att modifiera terrängmodellens topografi. Diken skapades genom att projicera delar av terrängmodellen för att passa en cylindrisk form, och vilken typ av terräng som en mesh skall representera är baserat på avståndet mellan meshen och vägen.

Slutresultatet visar att det är möjligt att använda procedurell generering för att skapa realistiska terrängmodeller, och att de genererade modellerna kan användas som en del av virtuella testmiljöer. Programmet är dock mer ett bevis på konceptet än en färdig lösning, och det finns flera områden som behöver förbättras, till exempel genom att implementera stöd för fler terrängtyper, stöd för mer än en noise map och genom att förbättra hur dikena interagerar med noise-funktioner.

För framtida forskning har *Volvo Cars* uttryckt intresse för att den utvecklade lösning skall kunna stödja procedurell generering av landsbygdsområden, samt berg och stöd för tunnlar.

Nyckelord: Processuell generering, Unity, Virtuella Testmiljöer, Radial Basis Function-interpolering, Noise-funktioner

Contents

1 Introduction

In recent years, more and more efforts have gone into the research of autonomous cars[1]. Cars with some degree of autonomy are already seeing use, and major research is being conducted in the area of entirely self-driving cars. There are several benefits associated with self-driving cars, such as decreased CO_2 emissions and fewer traffic related accidents[1]. However, there are also some unique problems which need to be solved in order for entirely self-driving cars to become reality. In order to minimise the risk of accidents, one such problem involves being able to test all possible scenarios an autonomous car might encounter, before the car is put into production[1]. Preferably, these tests should be as efficient as possible.

1.1 Background

This project contributes to the testing of the cameras that are to be used by self-driving cars. These cameras will provide the car with visual information about its surroundings, so that the car can react accordingly. Manually creating physical testing environments in real life to test these cameras is expensive, and they may be cumbersome to set up. For that reason, the creation of virtual testing environments is highly advantageous[2]. Perfect control of all the variables is provided, and the need to search for a road that perfectly matches all the requirements is eliminated. Thus, the generation of virtual environments seems to provide major savings in costs, risk, and time.

Manually creating varied virtual environments that look realistic can require a lot of work, which is expensive[3]. The problem is exacerbated when this process must be repeated for each combination of road and terrain that is used in the testing process. Instead of manually fitting the terrain to the road by hand procedural generation can be used. Procedural generation is defined as a method for step-by-step generation using algorithms[4]. Procedural generation is used in the creation of many different kinds of content such as games, movies and simulations[5].

1.2 Purpose

The project is carried out on behalf of Volvo Cars (referred to as the customer in the future), and the aim is to develop software that will procedurally generate a 3-dimensional terrain model. The models are to be used together with the customer's existing solutions to create a realistic environment for testing the cameras for their self-driving cars. The environment is needed since the cameras has to perceive things that can happen beside the road as well. It is also required that the terrain models are generated based on road meshes provided by the customer. It is important that the environment is as realistic as possible because otherwise important test cases could be missed.

While procedural generation is a well studied problem, the additional criterion that the terrain model must also adhere to a given roadway makes this a unique problem. The topics discussed here might also be of interest to other manufacturers of autonomous cars.

1.3 Limitations and Scope

The customer was interested in the development of several features, a few of which were selected and focused on in this project. The selected features were:

- The ability to closely fit a terrain mesh to a road.
- Generation of terrain meshes with bumpy surfaces.
- Generation of road ditches, along the sides of the road.

- Generation of several types of terrain meshes, allowing the customer to identify each type so that they could be populated with the use of the customer's own solutions.
- The ability to use real height data when generating the terrain meshes.

The focus of this project was exclusively to generate a terrain model. It is outside the scope of this project to generate terrain models in real-time. Generation of other models, such as trees, rocks, or other vegetation, was not the focus of this project. The customer instead uses pre-existing models and textures. This makes it possible for the models to be used in different virtual environments, which can have different requirements on the final result. The software was developed using *Unity*, and the final terrain models were exported as Alembic files, which was a requirement from the customer. Alembic is a file format which can contain computed results of complex procedural geometric constructions[6].

1.4 Related Work

Using virtual environments to test vehicles is not a new concept. For example, Huizinga et al. describe the benefits of using virtual environments for testing vehicles early in the development process, and how virtual environments can be used to calculate the fatigue life of the vehicle's different parts[7]. Aeberhard et al. also mention the benefits of virtual environments during the development of automotive vehicles[8]. The customer has also previously used virtual environments, although these were handcrafted models that were completely flat and as such, were unable to fit the terrain with roads of varying height.

Procedural generation of roads is a fairly well-studied problem[9, 10, 11, 12]. Although the roads are generally generated in order to be connected to certain objects, or generated to fit with the already generated terrain, and not the other way around, which the customer is interested in.

Procedural content generation based on input from the user is a concept that has grown in popularity over the previous years, since it is often more intuitive for the users to use[13]. For example, Parberry writes about generating terrain based on real-world elevation data[14], and McCrae and Singh describe a method for generating roads based on user sketched curves[15]. This is very similar to what the customer is requiring, and the method could be used to generate intersections, tunnels, and bridges. However, it doesn't modify the terrain to fit height differences of the roads.

Hnaidi et al. write about the use of diffusion equations to let the user be able to control the shape of the generated terrain by drawing 3D-curves[16]. Finally, Smelik et al. describe *procedural sketching* and their framework *SketchaWorld*, where the user can either colour a grid for landscape features, or place terrain objects manually, and these are then automatically fitted with its environment[17]. Here, the placement of roads that follow the terrain is possible. However, it is not possible to create roads that differ in height from the terrain, and then have the terrain adapt to the roads. This differs from what the customer is interested in since they only want to define a road, but avoid the work that comes with having to create a terrain that fits the road.

1.5 Overview

Section 2 consists of a detailed theoretical background, describing some of the mathematics and technical details behind the methods used. Section 3 describes the methods used in the project, step by step, describing the purpose of each method, and how the method works. Section 4 presents the results of the methods, describes how it relates to the purpose of the project, and discusses topics such as the suitability of the methods, the validity and usefulness of the results, as well as any unexpected findings. Finally, section 5 contains a short conclusion of the project, and discusses future research.

2 Theoretical Background

This section contains more in-depth information about procedural generation, noise and radial basis functions, which are all used in the creation of the terrain.

2.1 Procedural Generation

The most general definition of procedural generation is a system which uses algorithms to produce content based on a set of rules[4]. This means that, once the rules and algorithms for creation have been established, very little labour is required to create a virtually limitless amount of content.

A common example of content generated by procedurally generating systems is the structures of the levels or the worlds in video games, such as in the video game Minecraft as shown in figure 1 [18]. Minecraft uses procedural generation to generate the entire game world. The assets, which are mostly square blocks with different textures, are placed by algorithms following a set of rules. The result might not always be very realistic, but it is clear that the world that is generated follows certain rules.

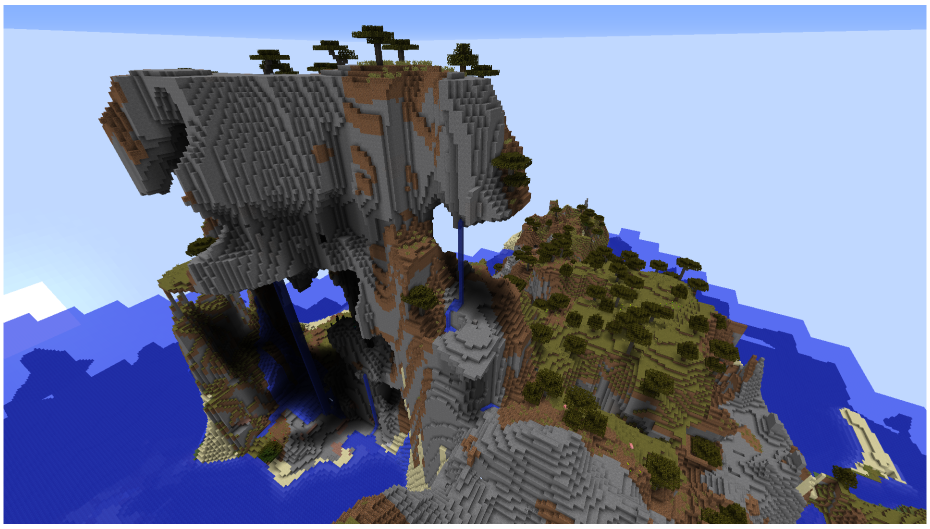


Figure 1: A screenshot of the video game Minecraft, in which the game world has been procedurally generated.

2.2 Noise Functions

Noise functions are pseudorandom sequence generators that are used to generate natural sequences. There are many types of noise. This section describes Perlin, Simplex and Worley noise.

2.2.1 Perlin Noise

Perlin noise is commonly used in procedural content generation for games and other visual media. The algorithm is used for many types of uneven materials and textures that can be used for generation of terrain, fire effects, water, or clouds[19]. The algorithm is mostly represented in two or three dimensions, but since the main feature of Perlin's algorithm is to make each point in the noise dependent on nearby points, it is possible to extend to any dimension.

When using the Perlin Noise algorithm from an implementation standpoint, the input to the function is x, y, and z, coordinates for a point in 3D, and the output will be a value between 0.0 and 1.0. The algorithm is called for every vertex in a mesh and each of these output values can be represented with colours, where values closer to 0.0 are represented with a darker colour, and values closer to 1.0 a brighter colour. With such a representation, the noise rendered as an image will resemble figure 2.

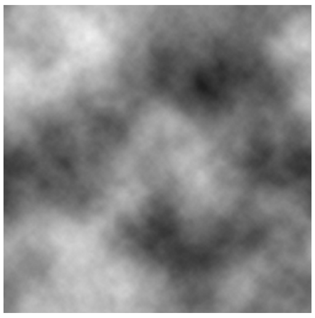


Figure 2: Two-dimensional Perlin noise function. The highest values are represented as white dots, and the lowest as the black ones (of course, various shades of gray represent all other values respectively). [20]

With each of the inputs we take modulus 1.0 in order to find a unit cube, which can be seen in figure 3. The cube is represented in 3D, in 2D a square is found and in 4D a tesseract, and together those are included in the family of hypercubes.

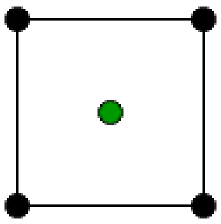


Figure 3: Unit cube with a point that represents the input.

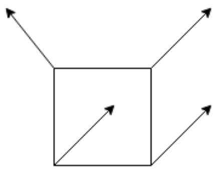


Figure 4: Unit cube with a gradient vector in each corner.

For each of the corners on the unit cube, a pseudorandom gradient vector is generated, as shown in figure 4. Each of these vectors define both a positive, and negative direction. Next we take the dot product between the gradient vector and the distance vector.

$$grad.x \cdot dist.x + grad.y \cdot dist.y + grad.z \cdot dist.z$$

In order to get a smoother transition the result from the corners in the cube is then blended. For a more natural looking interpolation, a fade function is used instead of a linear one. This fade function can look something like the following code and it is this fade function that Ken Perlin uses in his improved Perlin noise[21].

```
static double fade(double t)
{
    return t * t * t * (t * (t * 6 - 15) + 10);
}
```

2.2.2 Simplex Noise

Simplex noise was developed by Ken Perlin, which was designed to solve the issues Perlin had with his older noise algorithm, Perlin noise. This improved algorithm creates a similar noise pattern, but with multiple improvements, including less artefacts and a higher performance, with a time complexity of $O(n^2)$ instead of $O(n2^n)$[22].

This is mainly due to the fact that instead of using hypercubes like in Perlin noise to create a grid, it uses simplices, which represent a specific shape like hypercubes, but instead represent triangles in 2D, tetrahedrons (pyramids) in 3D, etc. As an example in 2D, Simplex noise uses equilateral triangles, side by side, to create a grid, as seen in figures 5 and 6.

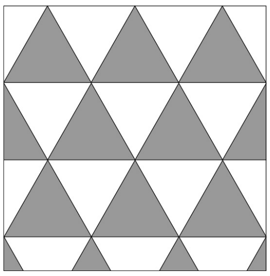


Figure 5: A representation of a simplectic honeycomb

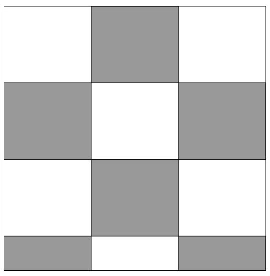


Figure 6: A representation of a hypercubic honeycomb

The reason simplices are used instead of hypercubes, is that they use less points to determine a coordinate inside its shape. In other words, when working in 2D, Perlin noise will interpolate with all four corners of a square to determine the output value for each point inside it. To determine the output of a coordinate inside a triangle, it is only necessary to work with the triangle's three corners. This might seem like a small difference, but compounds substantially for each higher dimension. For example, a 3-dimensional cube has 8 vertices, and a 4-dimensional tesseract has 16. This is in comparison to simplices, that only increase with one vertex for each higher dimension. The interpolations are calculated for each point inside each simplex. So if a higher resolution is desired for a better quality of the final product, it would lead to a large number of calculations using hypercubes, which is one reason why Simplex noise performs better than Perlin noise.

Compared to the first implementation of Perlin noise, which uses linear interpolation, Simplex noise determines the contribution of each corner by their distance to the point, and is then multiplied with the gradient of every simplex-corner affecting the current point. However, to be able to get the points from the hypercubic grid into the simplectic honeycomb grid, one has to manipulate the position of the points by skewing them, as shown in figure 7.

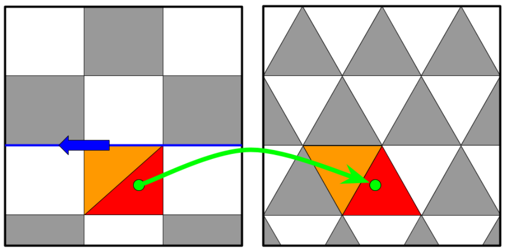


Figure 7: Skewing a hypercubic honeycomb along blue line, into a simplectic honeycomb (highlighting two triangles to show their new forms)

While Perlin noise uses a look up table to determine the index of each pseudorandom gradient vector, Simplex noise instead uses a bit manipulation system that only needs to use a small number of hardware computations to determine which way the unit gradient vector is pointing. Then the same is done as with the Perlin noise approach, using modulus with the xy-coordinates to determine which simplex the current point is in. The last step is then to compare the x and y value of the point to see which is larger and to determine which triangle the point is in this can be seen in figure 8. This works for any dimension, like in 3D as shown in figure 9.

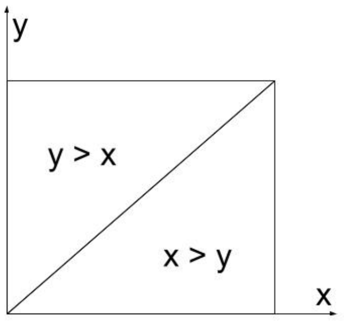


Figure 8: Determine which triangle the point is in, by comparing x and y.

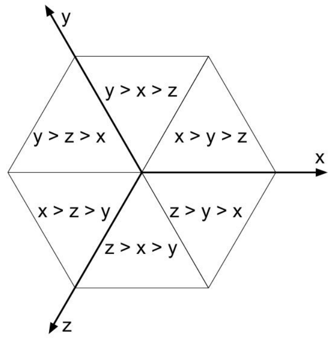


Figure 9: Determine which triangle the point is in, by comparing x, y and z.

2.2.3 Worley Noise

Worley noise, also known as Cellular noise, is an algorithm that also can be used to generate different types of patterns. The algorithm is computationally more expensive than Perlin, or Simplex noise, since the algorithm involves the calculation of the distance between all points. However, while other noise algorithms, like Perlin noise, often needs several iterations to generate interesting patterns, Worley noise can be equally interesting with just one. This algorithm was developed mainly for procedural texturing since this has proved itself valuable in image synthesis.[23]

The basic algorithm takes as input a random point in 2D or 3D, and for this point calculates the distance to the nth nearest points. This creates a pattern like the one seen in figure 10.[24]

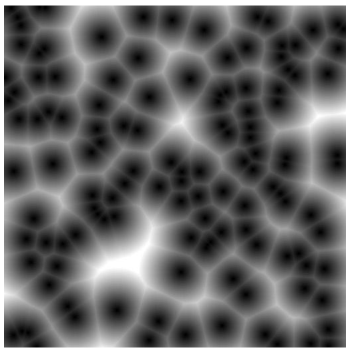


Figure 10: Worley noise with n = 128 points.

Before the function that calculates the distance is evaluated, a definition for the spread of the points in space needs to be obtained. To avoid artefacts we want an isotropic distribution and the simplest one is Poisson distribution[23]. This will divide space into a grid of uniformly spaced cubes that is shown in figure 11. The distance is then calculated to the points in the neighbour cubes.

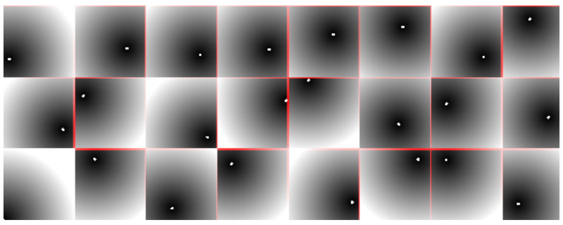


Figure 11: Space divided into a grid.
[24]

2.2.4 Fractal Noise

Fractal noise is obtained by taking several generated noises, either of the same type of noise or using different noises for every octave, and adding them together. For each consecutive octave, we modify the noise in two ways, successively decreasing the amplitude and increasing the frequency. The change in amplitude is called gain, and the change in frequency is called lacunarity[25]. For the amplitude to decrease, the gain should be somewhere in the range (0, 1). To have an increasing frequency, the lacunarity should be greater than 1. A higher number of octaves will result in a more detailed noise. In figure 12 below, we can see what a terrain based on fractal noise looks like.

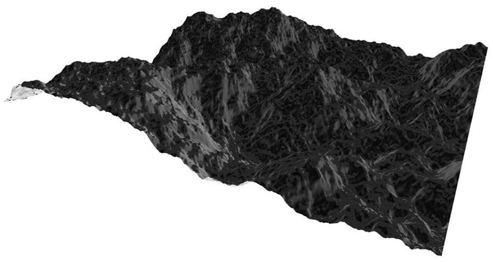


Figure 12: Fractal noise obtained using several iterations of Perlin noise.
[26]

2.3 Radial Basis Functions

A radial basis function (RBF) is a general function that takes in two real-valued points and returns an output purely based on the distance between them[27]. There are several different implementations of RBF functions; the ones covered here are two that suit this project well, namely the Gaussian, and inverse multiquadratic RBF.

The Gaussian, and inverse multiquadratic RBF are defined as:

$$\phi_{Gaussian}(p_1, p_2) = e^{-(\epsilon||p_2 - p_1||)^2}$$
$$\phi_{InverseMulti-Quadratic}(p_1, p_2) = \frac{1}{\sqrt{1 + (\epsilon||p_2 - p_1||)^2}}$$

where $p_1, p_2 \in \mathbb{R}^N$ are two points, $||p_2 - p_1||$ is the Euclidean distance between p_1 and p_2, and $\epsilon > 0$ is the shape parameter, which determines how quickly the RBF function will fall off. A large ϵ will result in a curve with a steep falloff and a small ϵ (approaching zero) will result in a gentler curve[28]. Plots of the two functions can be seen in figure 13 and 14.

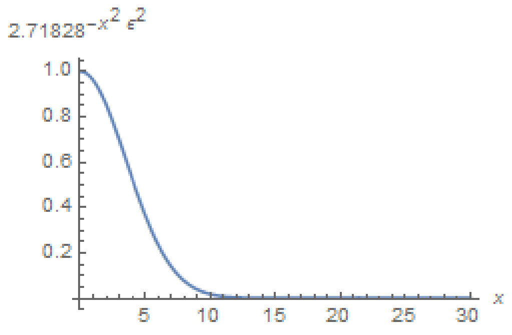


Figure 13: The Gaussian RBF with $\epsilon = 0.2$, where x is the Euclidean distance between two points p_1 and p_2.

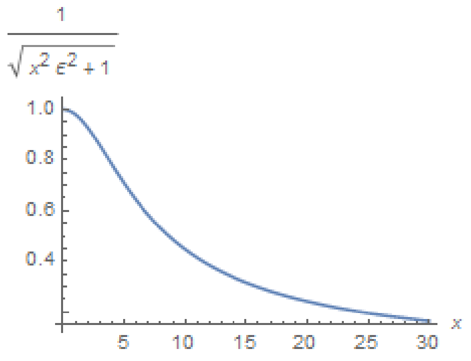
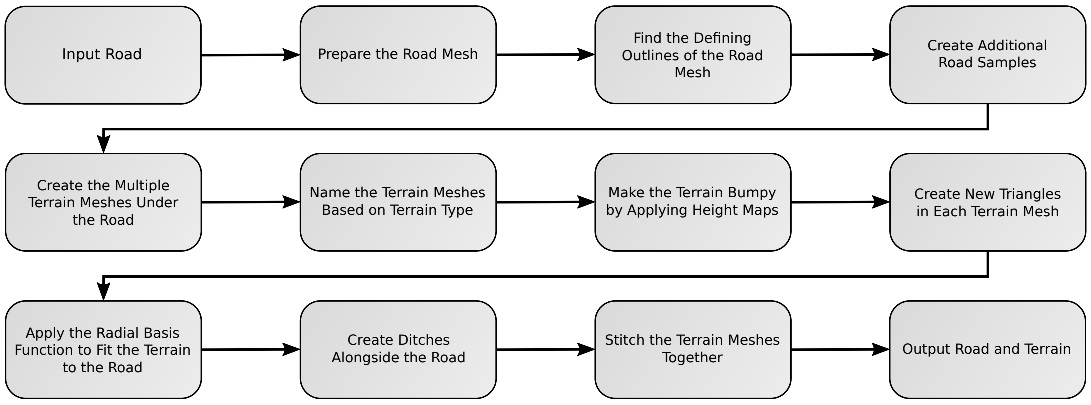


Figure 14: The Inverse Multi-Quadratic RBF with $\epsilon = 0.2$, where x is the Euclidean distance between two points p_1 and p_2.

3 Method

This section contains an exact description of how the terrain model was created, how it was fitted to the roadways, how it obtained a realistic look, and how the ditches were created. It also includes information on which tools were used, how the result was tested, and how it was verified to be correct.

3.1 Procedural Generation Sequence



Input Road
Prepare the Road Mesh
Find the Defining Outlines of the Road Mesh
Create Additional Road Samples
Create the Multiple Terrain Meshes Under the Road
Name the Terrain Meshes Based on Terrain Type
Make the Terrain Bumpy by Applying Height Maps
Create New Triangles in Each Terrain Mesh
Apply the Radial Basis Function to Fit the Terrain to the Road
Create Ditches Alongside the Road
Stitch the Terrain Meshes Together
Output Road and Terrain

Figure 15: A diagram visualising in what sequence different procedures were applied to create, shape and fit the terrain to the road.

The terrain generation was performed procedurally in several steps. In figure 15, these steps are shown, and below, each step is given a short description.

- **Road processes**
  - **Input Road** – The input road is a road mesh in the Alembic file format, provided by the customer.
  - **Prepare the Road Mesh** – The road mesh is cleaned up, and exported as a Blender model (.blend), a file format that is directly supported by Unity. In Unity, duplicate vertices are removed.
  - **Find the Defining Outlines of the Road Mesh** – The vertices in the road mesh that define the shape of road are found. This step is performed to speed up subsequent steps that use the road vertices and to make the road sampling more consistent.
  - **Create Additional Road Samples** – Additional samples along the road outlines are created. This is necessary for creating a good fit between the road and the terrain.
- **Terrain generation processes**
  - **Create the Multiple Terrain Meshes Under the Road** – Meshes are created to cover the terrain as needed, with the possibility of creating additional meshes to cover the surrounding areas.
  - **Name the Terrain Meshes Based on Terrain Type** – The terrain type and name of each mesh is assigned based on a predefined world type, and on the meshes' distance from the road.
  - **Make the Terrain Bumpy by Applying Height Maps** – Noise is applied to each terrain mesh, morphing the terrain to create an uneven and natural-looking environment.
  - **Create New Triangles in the Terrain Mesh** – Triangles are created in the terrain mesh to improve the fitting of the radial basis function.

- **Terrain deformation processes**
  - **Apply the Radial Basis Function to Fit the Terrain to the Road** – By using the radial basis function, the terrain is morphed so that the road fits with the terrain, giving a natural transition.
  - **Create Ditches Alongside the Road** – Ditches are added along both sides of the road by using the defining road outlines. The resolution of the terrain mesh where the ditches will be placed is then increased. Finally, the vertices in the terrain mesh are projected to form ditches.
  - **Stitch the Terrain Meshes Together** – The sides of the terrain meshes are stitched together, so as to avoid any gaps between the terrain meshes that occur due to the nature of the previous algorithms.
- **Output Road and Terrain** – Once the terrain has been generated, the cleaned road mesh is removed and the original, untouched, road mesh is imported using a third-party plugin called AlembicImporter[29]. The terrain is manually moved so that it aligns with the road, before everything can be exported into a single Alembic file, using the same plugin.

3.2 Preparing the Road Mesh

The road meshes were provided by the customer, but needed to be cleaned somewhat in order to work as input for the terrain generation. This section explains the details of that process.

3.2.1 Changing File Formats and Cleaning the Input Data

Each Alembic file provided by the customer contained a single road, constructed by many smaller meshes. These sub-meshes form different road segments, including the road markings. In many cases, there were several meshes on top of each other, making some of them redundant. Since the only vertices that were needed for the terrain morphing algorithms later on were the ones that made up the outlines of the road, the road markings were also redundant, and would probably have made the morphing erroneous. In order to facilitate working with the road data later on, all redundant sub-meshes were removed. The mesh names contained unique numbers, which were used to determine if they should be kept or be deleted. The remaining meshes were merged into a single mesh. All vertices were then rotated so that the mesh would get the correct rotation in Unity. This process of preparing the meshes was automated in Blender, using a Python script. Some of the roads, however, did not quite work with the Python script due to the mesh names not being consistent, and were thus cleaned manually. The modified road meshes were saved as .blend files, which were easily imported into Unity.

3.2.2 Removing Duplicate Vertices

Most of the vertices in the original road mesh were duplicated. This, by itself, was not a problem, but some triangles in the mesh used one copy of a vertex whereas other triangles would use a duplicate of that same vertex. This lead to problems when trying to find neighbours of a vertex, since the neighbours were found by checking which triangles that vertex was part of. Removing the duplicates was not difficult, but the triangles in the mesh also needed to be updated so that all the triangles used the same copy of a duplicated vertex. In algorithm 1 the process of removing duplicate vertices is described.

Algorithm 1 Removal of duplicate vertices

```
1:  add all vertices in the road mesh to a set V
2:  create a new index array A of the same size as the set V
3:  //Initialise A
4:  for each integer i from 0 to size of A - 1 do
5:      A[i] ← i
6:  end for
7:  create a map M and map all duplicates of a vertex v to the vertex v
8:  create a new triangle index array P
9:  for each integer i from 0 to size of the old triangle index array T - 1 do
10:     get old index d ← T[i]
11:     P[i] ← M[d] //Replace the old index with the new one
12: end for
13: set the new index array of the road mesh to A and the new triangle array to P
```

3.3 Finding the Defining Outlines of the Road Mesh

To be able to correctly place ditches, the road outline was needed. The ditch creation and RBF interpolation processes would also be more efficient if fewer road vertices were used. There needed to be a balance between performance and visual quality however. Visual quality was more important since the product was not be a real-time application and it was important that the terrain looked realistic. Luckily there should be no decrease in visual quality if the vertices in the middle of the road mesh are ignored, since this part of the terrain should never be seen anyway. The goal then, was to find the outermost vertices of the road and minimise the number of vertices used in the outline.

3.3.1 Finding the Outline Vertices

The outline of the mesh was found by sorting the vertices of the road mesh into an adjacency list and counting the neighbours of each vertex. In figure 16 the structure of the vertices in the mesh is illustrated. An alternative to the adjacency list is the adjacency matrix, but since each vertex only had up to six neighbours, the matrix would be very sparse and take up an unnecessarily large amount of memory compared to the list[30]. It is also easier and quicker to count the number of neighbours a vertex has when using an adjacency list.

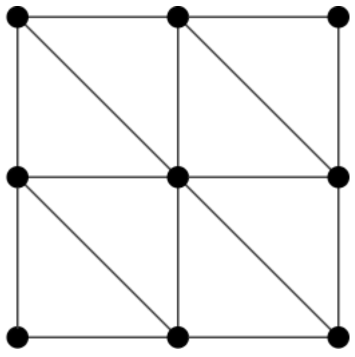


Figure 16: A depiction of the triangle mesh structure. Note that the vertex on the edges of the mesh have fewer neighbours than the vertex in the middle of the mesh.

Elements were added to the adjacency list by finding the vertices that made up each triangle, since each of those vertices had to be neighbours. A set structure was used, in which each element must be unique, instead of a normal list since each neighbour was only to be stored once. Once an adjacency set (same as adjacency list but using a set structure) was obtained, it was iterated through and a map was created. This map mapped the outline index to a list of its neighbours. Once this map was created, all that was left to do was to remove the vertices in the map that were not part of the outline. This was done by simply checking how many neighbours each vertex had, removing the vertices that had six neighbours. For the vertices with less than six neighbours an additional check was performed on each neighbouring vertex, where each neighbour who had six neighbours themselves was removed from the neighbour list.

3.3.2 Sorting the Outline Vertices

The road mesh that was given as input was not necessarily optimised. For the mesh to be optimised it would need to have as few vertices and triangles as possible while still retaining its shape. If this outline was used, the outline would have more vertices than necessary to retain the shape of the road. An example of an unoptimised outline and its optimised counterpart can be seen in figure 17.

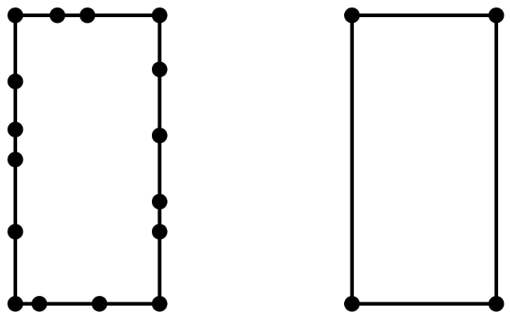


Figure 17: To the left, the unoptimised outline can be observed. To the right is the optimised version of the same outline. In this case the unoptmised outline uses four times as many vertices as the optimised outline.

The excess vertices were removed by comparing the slope of the edges between vertices. If the edge between the current pair of vertices had the same slope as the previous edge, then the vertex between the edges could be removed while still retaining the slope of both edges. Before the excess vertices could be removed, the vertices of the outline needed to be sorted so that the slopes of neighbouring edges could be compared. Then the actual comparison was performed. When sorting the vertices, the adjacency map was a very useful data structure, as the neighbours of any vertex on the outline could easily be found. Algorithm 2 describes the process of sorting the vertices.

Algorithm 2 Outline vertex sorting

```
1: while index map M is not empty do
2:     pick arbitrary start vertex s
3:     set the current vertex c ← s
4:     create next vertex n and previous vertex p
5:     while c != s && c != p do
6:         add c to the list of sorted vertices
7:         pick neighbour of c that is != p and set n to be this neighbour
8:         remove c from M
9:         p ← c
10:        c ← n
11:    end while
12:    add s to the list of sorted vertices as the last element
13: end while
```

For any outline, any vertex could be chosen as the start vertex since the outline would connect in a loop. The start vertex was added as the last element because the edge between the actual last vertex and starting vertex also needed to be considered. This algorithm divides the vertices into lists of different outlines, as some roads had outlines that were not connected by any vertices (this was true of circular roads for example).

3.3.3 Removing redundant vertices

When the vertices had been sorted, the superfluous vertices were removed. The algorithm uses a list of sorted vertices and is described in algorithm 3.

Algorithm 3 Removal of redundant outline vertices

```
1: set start index s ← 0
2: for each integer i from 0 to size of the list of sorted vertices S - 2 do
3:    calculate slope a between S[i] and S[i + 1]
4:    calculate slope b between S[i + 1] and S[s]
5:    if a != b with error margin $\epsilon$ then
6:       add S[i] to defining outline
7:       s ← i
8:    end if
9: end for
10: //Perform check to see if the first and last vertex (same vertex) is part of the defining outline
11: calculate slope c between S[1] and S[0]
12: calculate slope d between last element in S and S[1]
13: if c == d with error margin $\epsilon$ then
14:    remove S[0] from defining outline
15:    add S[1] to defining outline
16: else
17:    add last element in S to defining outline
18: end if
```

The algorithm checks if the edge between the current and the next vertex have the same slope as the edge between the last vertex where the slope is known to have changed and the next vertex. It is necessary to check whether the first vertex was actually needed or not, since the first node was chosen arbitrarily, it might be the case that the first vertex is not part of the defining outline. The margin for error had a use beyond allowing for rounding errors when normalising the vectors. It could be used to allow vertices which have very slight differences for the slope between vertices to count as being the same slope, which would reduce the number of defining vertices. Granted, the shape would not be preserved exactly, but it would allow for a further increase in performance.

3.4 Creating Additional Road Samples

If only the defining outline vertices of the road mesh were used as samples for the RBF interpolation, there was no guarantee that there would be enough samples to make the road and terrain look like they were realistically connected. The problem is caused by having sparsely placed road samples and it can be remedied by taking more and consistently distanced samples along the outline of the road mesh. The reason that new samples were created on the defining outline instead of simply adding samples to the original outline of the road is that when using the defining outline, new samples could be placed with the desired distance from each other more consistently. The sampling was performed by simply calculating the slope between each of the vertices in the list of sorted defining vertices. In algorithm 4 this process of taking more samples is outlined.

Algorithm 4 Outline sampling

```
1: get distance between samples d
2: for each integer i from 0 to size of the defining outline D - 2 do
3:    start vertex s ← D[i]
4:    current vertex c ← D[i]
5:    next vertex n ← D[i + 1]
6:    add s to the sampled outline
7:    calculate slope l between n and c
8:    //Keep adding samples until a sample would be placed on or beyond the next defining vertex
9:    while distance between n and c < m do
10:      create new sample p ← c + l · m
11:      add p to the sampled outline
12:   end while
13: end for
14: add last element of D to the sampled outline
```

The reason that the defining vertices were added to the list of samples was so that the shape of the road mesh was preserved. In order to avoid visual glitches when morphing the terrain later on, samples that were too close to another sample were ignored. Since the samples were sorted, this could be done by comparing the Euclidean distance between each sample and the previous one.

3.5 Creating the Terrain Meshes

The first step of generating the terrain model was to create meshes[31]. Meshes were used because the height of individual vertices can be altered to create something which resembles a natural looking terrain.

Due to technical limitations, there was an issue generating larger terrain. Unity supports a maximum of 65,535 vertices per mesh, which meant that it was necessary to create multiple meshes to represent the terrain. Using multiple terrain meshes introduced the need to stitch the meshes together since they act as separate objects when the various calculations are done.

To create larger terrains, an additional parameter is used as input to the terrain generation function which specifies the number of additional terrain mesh layers to add outside the terrain. The terrain meshes are created to cover the entirety of the road, so that no translation is required. By ensuring the road is directly centred above the terrain, the road will not be close to the terrain mesh edges even when there are no outside layers.

Since each terrain mesh has a specific size and resolution, some calculations had to be done to ensure that the terrain completely covered the road. By looking at the minimum and maximum vertices of the bounding box of the road mesh, a check is performed to see how many meshes can fit under the road[32]. To keep it centred, it is also offset so that at least half the terrain will be outside the road.

Based on the road mesh, two bounding box vectors b_{min} and b_{max} are extrapolated, representing two opposite corners of the terrain. The size of the terrain in the two axes x, $size_x$, and z, $size_z$ are calculated as follows. The size of each terrain mesh, $terr_{sz}$, is also used to calculate the offset.

$$offset = \frac{terr_{sz}}{2}$$

$$size_x = (b_{max}.x + offset) - (b_{min}.x - offset)$$

$$size_z = (b_{max}.z + offset) - (b_{min}.z - offset)$$

The size of the terrain is used to calculate the number of terrain meshes that fit under the terrain, so that it is easy to iterate up to that number when creating the meshes. Note that the offset is used to centre the terrain, as without it, the terrain will start immediately where the road begins. The number of terrain meshes is calculated as follows, where oc is the number of outside layers to be added.

$$count_x = \lceil \frac{size_x}{terr_{sz}} \rceil + (2 \cdot oc)$$

$$count_z = \lceil \frac{size_z}{terr_{sz}} \rceil + (2 \cdot oc)$$

To get an integer value, the value is rounded up. The reason for this is that it is better to fit too many terrain meshes under the terrain, than too few. If there are too few, and the number of outside layers is too low, there will be a part of the road that is not above the terrain.

By iterating through a two-dimensional array of size $count_x * count_z$ with indices i and j, several terrain meshes were created with values based on the previously calculated values and using i and j as indices. Each new terrain mesh got its position, where the position is the upper left corner of the new mesh, as follows.

$$\begin{bmatrix} b_{min}.x + ((i - oc) \cdot terr_{sz} \\ 0 \\ b_{min}.z + ((j - oc) \cdot terr_{sz} \end{bmatrix}$$

Looking at the bounding box to find the position of the road, the terrain mesh is offset further. Using its index, the position of the terrain mesh is multiplied by its size, so that each terrain mesh fits directly next to one another. Additionally, it is possible to check if the terrain mesh is under the road or not, by looking at its i and j index. A terrain mesh will be under the road if the following four Boolean statements are true.

$$i > (oc - 1)$$

$$i < (count_x - oc)$$

$$j > (oc - 1)$$

$$j < (count_z - oc)$$

This can be useful for heavy operations on the terrain that are only really useful where the terrain is under the road, such as for the radial basis function.

3.6 Defining Terrain Types

In order for the customer to be able to populate the virtual testing environment efficiently, the meshes needed to be named according to the terrain type the meshes should correspond to. What type of terrain a single mesh should represent was determined by one of four preset world types, and the distance between the bounds of the mesh to the closest point on the road.

In order to achieve realistic transitions between different types of terrain, a terrain type called "transition" was added in between the different terrain types. If, for example, the desired world consists of fields and forests, a majority of the created meshes would be of one of these types. These can be populated automatically, but where different terrain types meet, the customer might want to populate them manually.

The algorithm for determining the name of the meshes works as follows. For every terrain mesh, starting with the terrain type farthest from the road, a check is performed to see if the closest squared distance between the bounds of the mesh and the closest vertex on the road was less than a preset value associated with that terrain type. If it was, that mesh is assigned the name of the terrain type. Otherwise there was at least one vertex in the mesh that was too close to the road, which means that is should be part of another type.

The same process is then repeated for each different terrain type except for the one closest to the road. Between every layer of different kinds of terrain, a transition type layer is added. In order to avoid any errors caused by meshes not being given a type, if a mesh is not named when every terrain type except the one closest to the road has been checked, it is automatically named after the type closest to the road.

In order to achieve a realistic result, the algorithm requires a fairly small mesh size if more than one kind of terrain is desired. The reason already existing meshes are used instead of generating new ones is because the used method is simpler, and the achieved result was sufficient for the intended purpose.

3.7 Making the Terrain Model Bumpy

In order to make the terrain model resemble a realistic terrain, it needs to be changed from its current flat state. The method which was used was to either generate a noise map which can be used as a height map, or to create a height map from real height data. The height map would then be used to offset each vertex in the mesh with the corresponding value in the height map which will make the terrain model bumpy.

- **Generating Fractal Noise** The software allows various input that will influence the generated noise map, as well as the ability to choose between Perlin noise, Simplex noise or Worley noise. In order to get more realistic terrain the noise is applied several times using fractal noise[25].
- **Generating Worley Noise** For the Worley noise a library called LibNoise is used to generate the noise map with help of the creation of a generator called a Voronoi. The Worley noise was initially planned to be used as a basis for generating mountains, but that functionality was not implemented fully due to time contraints[33].
- **Applying Real World Height Data** The customer requested functionality for using real height data, instead of using artificial height data generated by a noise function. This implementation was specifically geared to use data provided by Lantmäteriet[34], as was suggested by the customer. The real world height data application method takes as input a text file with all the height data. This file has to have at least the same resolution as the mesh, in both dimensions.

3.8 Creating New Triangles in the Terrain Mesh

In order to get a believable transition between the road and the terrain, the terrain had to connect properly to the road's edges. Since there were no guarantee that there were any terrain vertices at the same (x, z)-position as a given road sample, the terrain might not adhere properly to the road. The goal was to solve this problem by adding new terrain vertices along the road's edges, creating new triangles in the terrain mesh. The triangulation functionality was implemented but removed later on, due to underwhelming results. Regardless, the process is explained below.

The first objective of the algorithm was to find the triangle in the terrain mesh that lied directly under or above each road vertex. In order to find the correct triangle quickly, all squares in the terrain mesh were stored in a list when the mesh was created. The squares were defined by a global (x, z)-position and a list of contained triangles. Since the squares were sorted both by x-value and z-value, comparing the given point's x- and z-values to the squares' x- and z-values, a binary search algorithm could find the correct square with time complexity $O(log(\sqrt{n}))$, where n was the number of squares in the terrain mesh.

When the correct square had been found, each of the square's contained triangles had to be checked to see if they contained the given point. This was effectively done by converting the point into barycentric coordinates with regard to the triangle. The barycentric coordinate system is a way of describing the location of a point inside of a simplex, in this case a triangle, by how close the point lies to each of the simplex' corners. By definition, if and only if all of the barycentric coordinate values are positive, the point lies inside that triangle[35].

When the correct triangle was found, the task was to split it into three new triangles. This was done by updating the mesh's list of vertices and its list of triangles. To update the lists for each road vertex would have been computationally costly. In order to optimise the time, all new triangles were stored in the mesh square data structure mentioned earlier. The mesh's triangle list could then be updated after all triangulation was complete, by iterating through all squares and saving all of the squares' triangles as mesh triangles. The vertex list, however, had to be updated for each road vertex since the squares didn't contain any information about the vertices.

3.9 Fitting the Terrain to the Road

In order to get a believable and natural transition between the road's edges and the terrain model, the terrain model had to be morphed in some way, by means of moving its upwards or downwards. The height of points on the terrain model, located directly under or above a sample vertex from the road outline, should be offset so that they end up at the exact same height as the corresponding road sample, removing the gap between the road and the terrain. Terrain model vertices that lie just a little bit from the road should be moved almost as much, while vertices further from the road should not be displaced very much, if at all. This gave incentive to use a Radial Basis Function (RBF) to calculate the target height offset, since the RBF gives an output value purely based on the distance between two given points.

3.9.1 Morphing the Terrain with RBF Interpolation

In order to get a smooth and natural terrain curvature, each terrain vertex would have to be influenced by several road sample vertices. For simplicity, let's assume that each terrain vertex is influenced by all road samples, based on the distance between the terrain vertex and each sample. Thus, the target height value for each terrain vertex is an RBF interpolation of all samples' heights (global coordinates are used in all calculations below):

$$\boldsymbol{h}_{terrain,i}^{(target)} = \boldsymbol{h}_{terrain,i} + \sum_{j=1}^{M} \phi(\boldsymbol{t}_i, \boldsymbol{r}_j)\boldsymbol{w}_j \tag{1}$$

where $\boldsymbol{t}$ is the vector of terrain vertices, $\boldsymbol{r}$ is the vector of road sample vertices, $\phi(\boldsymbol{t}_i, \boldsymbol{r}_j)$ is the two-dimensional RBF for point $\boldsymbol{t}_i$ and point $\boldsymbol{r}_j$, $\boldsymbol{w}$ is the vector of unknown weights for each road sample, and M is the number of road sample vertices.

There are many RBFs to choose from. Two feasible options for the RBF in this context are the Gaussian RBF and the inverse multiquadratic RBF (IMQRBF)[28]. Both worked well but with slightly different outputs. In the end, the IMQRBF was chosen for the final result, since it gave a more smooth and natural slope. Both of the mentioned RBFs have a real, positive shape parameter ϵ which determines the slope of the terrain towards the road. A large ϵ will give a sharp slope towards the road and a small ϵ will give a more smooth slope[28]. Through empirical testing, $\epsilon = 0.2 \pm 0.1$ was chosen since, in combination with the IMQRBF, it gave a smooth and natural transition towards the road.

In order to get the weight corresponding to each road sample, the height difference between the terrain and each road sample were used. For the points on the terrain model which lie directly under a road sample, when morphing, the target height must be the same as the corresponding road sample's height.

$$\begin{aligned} \boldsymbol{h}_{\boldsymbol{p}_i}^{(target)} &= \boldsymbol{h}_{\boldsymbol{p}_i} + \sum_{j=1}^{M} \phi(\boldsymbol{p}_i, \boldsymbol{r}_j)\boldsymbol{w}_j \\ &= \boldsymbol{h}_{\boldsymbol{r}_i} \end{aligned}$$

where $\boldsymbol{p}$ are the points on the terrain mesh which lie directly under or above the road samples $\boldsymbol{r}$.

Let $\boldsymbol{A}$ be the $M \times M$-matrix with $\boldsymbol{A}_{i,j} = \phi(\boldsymbol{r}_i, \boldsymbol{r}_j), i, j \in (1, \ldots, M)$, $\boldsymbol{h}_{terrain}$ the vector of all $\boldsymbol{p}$'s heights, and $\boldsymbol{h}_{road}$ the vector of all road samples' heights. Using all road samples, a linear equation system is extracted, with M equations and M unknowns, the weights:

$$\boldsymbol{h}_{\boldsymbol{r}} - \boldsymbol{h}_{\boldsymbol{p}} = \boldsymbol{A}\boldsymbol{w}$$

Since $\boldsymbol{A}$ is invertible, the weights can be solved for:

$$\boldsymbol{w} = \boldsymbol{A}^{-1}(\boldsymbol{h_r} - \boldsymbol{h_p})$$

In the implementation, $\boldsymbol{w}$ was calculated using the *Solve* method in the C# library Mathnet.Numerics[36].

Once $\boldsymbol{w}$ had been calculated, all terrain vertices could be updated according to (1).

3.9.2 Avoiding Terrain Clipping Through the Road

Due to the previously applied noise, the terrain model would in some places clip through the middle of the road, since there were no road samples there to make the terrain fit to the road. In order to avoid this, terrain vertices that were positioned above the road were assigned a height that was a little bit lower than the road height instead of the RBF interpolation value.

To determine whether a terrain vertex lied inside the road, a ray casting algorithm, known as the Crossings test, was used[37]. The test was performed in two dimensions, along the x- and z-axis. The crossings test is the fastest point-in-polygon test without processing the data (the vertices) in any way [37]. It was used in conjunction with an effective line-line intersection test[38], which was modified slightly. The modification simply changed the allowed range of one of the lines from [0, 1] to [0, ∞), which makes the test into a ray-line intersection test.

3.9.3 Optimising the Implementation

Due to the way RBF interpolation was initially implemented, an increasing number of terrain meshes would significantly degrade the performance of running the RBF interpolation. While performance was not the main goal of the project, the optimisations were easy enough to implement, and it greatly increased the efficiency of rendering and testing the terrain, speeding up the development process.

The original implementation of the RBF interpolation used every road sample with each of the terrain meshes. Given the potential distance between road samples and terrain mesh vertices, it was possible that such a calculation would not even alter the terrain, as the distance between the two vertices would be too large.

The first optimisation was to calculate which terrain meshes were counted as being outside the road area. This was touched upon at the end of section 3.5, where the possibility of checking if the terrain mesh was under the road or not by immediately looking at its index during generation was discussed, by checking the following Boolean statements.

$$i - p > (oc - 1)$$
$$i + p < (count_x - oc)$$
$$j - p > (oc - 1)$$
$$j + p < (count_z - oc)$$

A property is added to each terrain mesh for which these statements hold true, specifying that it is, indeed, under the road. An additional input parameter, p is also added, specifying how many outside layers will be counted as being under the road. This is used when the terrain meshes are too small, as it is necessary to extend the RBF interpolation to meshes further away.

The second optimisation is to look at which terrain mesh each road sample is above, and map each terrain mesh to a list of road samples, which will be used as the input to the RBF for that terrain mesh, instead of the list of all road samples. This can be done by checking a simple box collision, ignoring the y value of each sample, and focusing only on the x- and z-values.

Assume a vector $\boldsymbol{r}$ which represents a single road sample in the iteration of all road samples, and two vectors $\boldsymbol{b}_{min}$ and $\boldsymbol{b}_{max}$, representing the two opposite corners of the bounding box of the terrain mesh. The vector $\boldsymbol{r}$

will be above the terrain mesh if the following Boolean statements are true. An additional input parameter, q, is added, which specifies the additional distance to be checked in the box collision.

$$\boldsymbol{r}_x + q > \boldsymbol{b}_{min_x}$$

$$\boldsymbol{r}_x - q < \boldsymbol{b}_{max_x}$$

$$\boldsymbol{r}_z + q > \boldsymbol{b}_{min_z}$$

$$\boldsymbol{r}_z - q < \boldsymbol{b}_{max_z}$$

The reason for using the additional variable q is because road samples may lie too close to the border of two terrain meshes, and it would therefore only morph one of the terrain meshes with RBF interpolation.

3.10 Creating the Road Ditches

The creation of road ditches was completed in several steps. First, using minimal lists of road vertices, the correct sides of the road, where the ditches should be placed, were found. Following that, the ditch was placed alongside the road. When the placement of the ditch was completed, vertices in the terrain mesh were projected to fit a cylindrical shape, which resembled a ditch. But before the projection, the resolution of the terrain where the ditch was to be created was increased. This was done because the resolution of the terrain was too low to support a realistic looking ditch. Once the resolution was increased, the vertices in the terrain mesh could be projected. Here the methods detailing these processes will are explained in further detail.

3.10.1 Finding the correct of side of the road

The first step in placing the the ditches was to find which side of the outline the ditch was to be placed on. The ditch was to be placed parallel to the road with some offset. The offset needed to be perpendicular to the the outline for the placement to be parallel, and so needed be rotated, either clockwise or counter-clockwise, to the outline. At this point, the road could be thought of as only existing in in two dimensions, ignoring the axis pertaining to the height, in this case the y-axis. Depending on how the offset vector was rotated (clockwise or counter-clockwise), the ditch would either be placed inside the road, or outside of the road.

The road outline consisted of one or several lists of sorted vertices, depending on how many outlines the road had. A road that did not connect with itself in any way would have a single outline, whereas a circular road would have two outlines. It was only necessary to find the direction that the offset was to be rotated in for each outline, since the direction for the rotation for all the vertices of the outline would be the same. Since one of the two directions yields the correct rotation, it was enough to check one of the cases, the counter-clockwise case was used in this implementation, but checking the clockwise case would is equally valid.

Which direction the vertex should be rotated towards was decided by doing a crossings test [37]. But before the crossings test was performed, a road vertex was offset $\frac{\pi}{2}$ radians around the y-axis in a counter-clockwise direction. Then the crossings test was performed to check whether the vertex was inside the road or not.

Two neighbouring vertices in the list of road outlines were chosen, the first and second vertex for example. Any pair would do but it was important that they were ordered correctly since the direction of the desired offset would be reversed otherwise. The y-values for the vertices were ignored. The slope that is counter-clockwise perpendicular to the slope between the two vertices could then be found with the rotation matrix:

$$\begin{bmatrix} cos\,\theta & 0 & sin\,\theta \\ 0 & 1 & 0 \\ -sin\,\theta & 0 & cos\,\theta \end{bmatrix} \qquad [39]$$

Since $\theta = \frac{3\pi}{2}$, $cos(\theta) = 0$ and $sin(\theta) = -1$, the resulting matrix multiplied with the slope resulted in the following slope.

$$\begin{bmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -z \\ y \\ x \end{bmatrix}$$

The perpendicular slope was used to offset the first vertex very slightly, first the slope was normalised and with the following expression the new position for the vertex was found:

$$\boldsymbol{newPosition} = \boldsymbol{vertexPosition} + \boldsymbol{perpendicularSlope} \cdot t$$

where t is a small number, t = 0.01 for example. This was to ensure that the vertex was not offset so much that the new position actually ended up outside the road outline when a very small offset would have positioned the vertex inside the outline. If the point was found to be inside the road then it was offset in the direction clockwise perpendicular to the outline. In the case that the vertex was found to not be inside the road, the vertex was offset counter-clockwise perpendicular to the outline.

3.10.2 Road Ditch Placement

Each ditch segment was to consist of a start and end point, and the edge between them was supposed to be parallel to the edge between a segment in the defining road outline. By taking vertex i and i + 1 from the list of defining road vertices, where $i = 0, 1, ..., n$ and where n is the last element in the list, and offsetting these vertices and increasing i by one for each pass in the loop, the result was a start and end vertex for each ditch segment that are offset along the same direction. This process is depicted in figure 18.

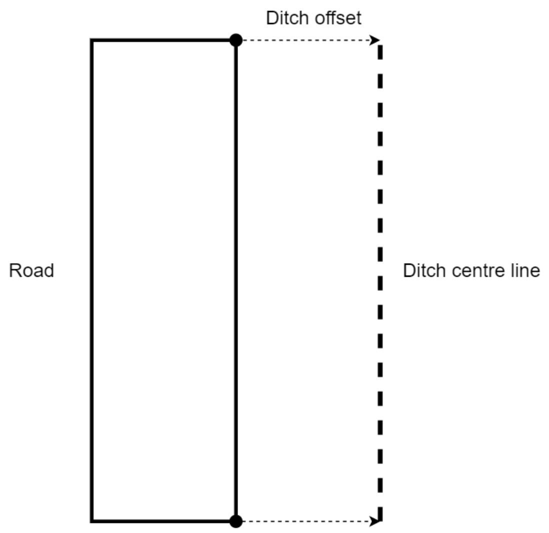


Figure 18: A figure visualising how ditch segments are placed by offsetting the line segment between road vertices.

This also meant that neighbouring segments that had an angle smaller than π radians from the side closest to the road would not have any empty space between them, they would in fact intersect. But for neighbouring segments that had an angle greater than π radians there would be some empty space between the ditch segments, which was a problem that needed to be rectified.

Once the direction of the offset was known, the rotation matrix for rotating around the y-axis was used to position the ditch segments, rotating the corresponding vertices of the defining road outline either $\frac{\pi}{2}$ or $\frac{3\pi}{2}$ radians, depending on whether the segment should be rotated clockwise or counter-clockwise respectively. The offset vector was normalised, and the position for the the start and end vertex for the ditch segment was calculated as in the expression:

$$\boldsymbol{d} = \boldsymbol{v} + \boldsymbol{o} \cdot t$$

where d is the ditch vertex, v is the road vertex, o is the offset vector and t is the offset distance.

As previously mentioned, the pairs of segments with angles greater than π radians (measured from the side away from the road) would not intersect. This would lead to strange looking, spotty ditches that would seemingly start and end arbitrarily. To remedy this an additional pass over the ditch vertices was required. A line segment-line segment intersection test was performed to check if the segments intersected. If there was an intersection nothing was done, since the ditch would be continuous. If there was no intersection, then it could be concluded that the angle between the segments was greater than π radians. Knowing this, it could also be concluded that if the segments were longer, they would intersect at some point. A depiction of what the original ditch placement looked like can be found in figure 19.

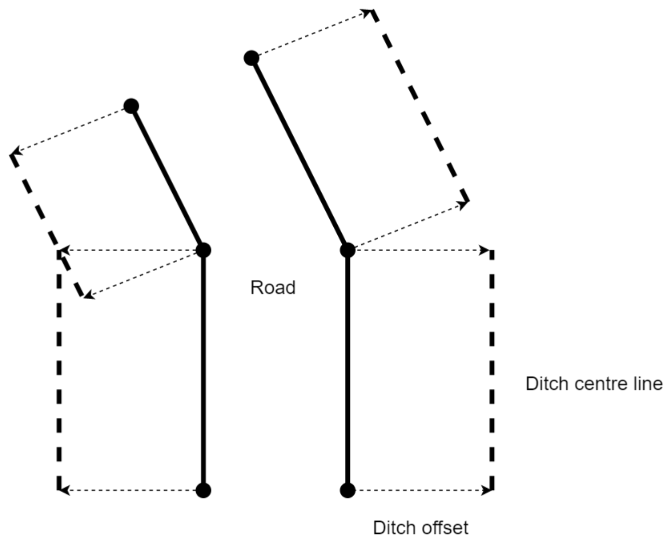


Figure 19: A figure depicting a curved road. The ditch alongside the inside of the curve intersects with itself. The ditch alongside the outside of the curve will not be continuous, the segments need to be elongated.

To find at which point the line segments intersected, rays were shot along the slope of both lines, from the start points of both lines and along the positive direction for the first segment and negative direction for the second segment. The intersection point was found almost identically to how the intersection point of two lines is found[38], only the allowed ranges for the line factors changed. This intersection point replaced the end point of the first ditch and start point of the second ditch. Before the points were replaced, a small offset was added to the end point of the first segment along the slope of the line going from the end point to the point of intersection. For the start point of the second segment a similar offset was added, but instead along the slope of the line going from the start point of the second segment to the point of intersection. The purpose of these offsets were to ensure that the segments did in fact overlap. Without offsets, vertices in the terrain mesh were are positioned exactly between the ditch segments might have been found to not intersect with either ditch segment.

3.10.3 Ruling Out Remote Terrain Meshes

Before increasing the resolution and creating a ditch in the terrain mesh, an intersection test was performed to see if it was even possible that the mesh would have a ditch on it or not. By doing an axis aligned bounding box (AABB)-AABB intersection test between each terrain mesh and the ditches, meshes that were not intersecting were precluded from further tests [32].

3.10.4 Increasing the Resolution of the Road Ditch

For the ditches to look round, the terrain mesh needed to have fairly high resolution. To ensure that the ditches looked good even in cases where the resolution of the mesh was quite low, an algorithm that increases the resolution of the terrain mesh where the ditch is positioned was used.

The resolution was increased by splitting existing triangles into four new, smaller triangles, all of which are equal in size and with the same angle ratios as the larger triangle. This allowed for an even distribution of vertices and it integrated nicely with the existing mesh as neighbouring triangles would not be affected by a triangle having an increased resolution. To increase the resolution of the ditch, the triangles that were part of the ditch needed to be found. Since the ditch segments had a cylindrical shape it was reasonable to perform a cylinder-triangle intersection test to find which triangles were part of the ditch.

The way in which the resolution was increased was by iterating through a loop where a check was performed to see which triangles in the terrain mesh were intersecting with any ditch segment, and then increasing the resolution of those triangles. This process was repeated for each triangle, saving the list of intersected triangles from the last iteration, and repeating the loop. For the first iteration, all the triangles in the terrain mesh were used when checking if an intersection occurs with a ditch segment. For each ditch segment a cylinder-triangle intersection test was performed to see if the triangle was part of the ditch.

To summarise the process of the cylinder-triangle intersection test, first a test was performed to see if a triangle and cylinder (ditch segment) intersected on the z-axis, and if they did intersect, another test was performed to check whether they intersected on the x- and y-axis. To simplify the coordinates, a change of basis was performed so that the point (0, 0, 0) was positioned in the middle of the centre line for the ditch segment [40].

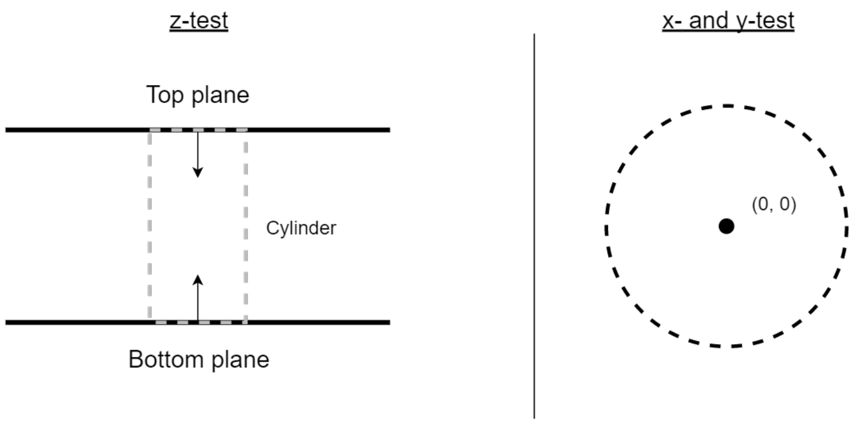


Figure 20: A depiction of the cylinder-triangle intersection tests.

First, the new base for the coordinate system was calculated. For this new coordinate system, the y-axis went along the slope of cylinder centre-line. The x-axis was perpendicular to the the y-axis. The z-axis was then found by taking the dot product of the y- and x-axis. Then the new coordinates for each triangle vertex was calculated as:

$$\begin{aligned} x_q &= U \cdot (x - C) \\ y_q &= V \cdot (y - C) \\ z_q &= D \cdot (z - C) \end{aligned}$$

where the C is the centre point of the cylinder and U, V and D correspond to the x-, y- and z-axes in the new coordinate system. With the new coordinate system, the z-axis can be used to test against the height of the cylinder, whereas the x- and y-axis can be used to test against the radius of the cylinder. Once the position of the triangle vertices in the new coordinate system had been calculated, they were sorted after their z-value in ascending order. The z-values were then tested to make sure that there was at least one vertex positioned between or on the planes defined by the end or start point of the cylinder, and by a normal that was directed toward the cylinder centre.

Different tests were performed for the x- and y-axis depending on how the triangle vertices were placed. If no points were placed between the planes, but one point was positioned on one of the planes, the distance to the centre point was simply calculated. If two point were placed on a plane, the distance between the closest point and the centre was calculated. The factor for finding the closest point on a line to a point can be found according to:

$$t = -\frac{(x_1 - x_0)(x_2 - x_1)}{|x_2 - x_1|^2} \qquad [41].$$

The value t was then clamped to the nearest value in the range [0, 1] as this is a line segment.

If at least one point was between the planes, it was necessary to find the polygon existing between the planes. This polygon was found by finding the point where the edges between vertices intersected with the planes. The factor for the point along the line that intersects the plane can be found according to:

$$t = \frac{(P_0 - P_1) \cdot n}{(P_2 - P1) \cdot n} \qquad [42].$$

Once the polygon between the planes had been found, a crossings test was performed to check if the centre point was inside the polygon. If this test failed then distance between each edge and the centre was calculated. If the shortest distance was no greater than the radius, there was an intersection. Otherwise, there was no intersection [40].

Once a triangle had been found to be intersecting with any ditch segment, its vertices were added to a list of intersected triangles and then the next triangle was tested. Before creating new triangles, a new list of triangles was created by going through the array of triangles in the terrain mesh and adding the triangles to the list but excluding any triangle that were also in the list of intersecting triangles. This removed the existing, intersected triangles, inside each of which four new triangles would be created. This was necessary, since otherwise the larger triangle would envelop the four new triangles that were to be created, possibly covering or intersecting them, diminishing the point of creating new triangles in the first place. Once all the intersecting triangles had been removed from the new list of mesh triangles, new triangles could be created.

When creating triangles the first step was to create new indices for them, these would be the current length of the index array plus 1, 2 and 3 for the three indices. Then the three new vertices needed to be placed. Each vertex was placed in the centre of each edge in the intersected triangle. The correct indices needed to be chosen for each triangle and added to the list of new triangles. The indices of the triangles were added in counter clockwise order, it did not matter which index was chosen first. The figure 21 shows which vertices were part of each triangle.

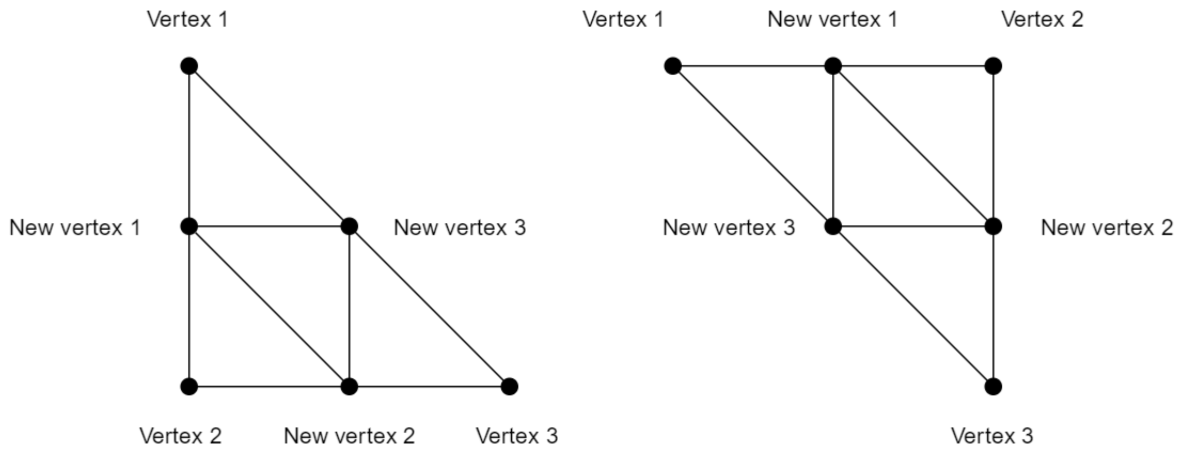


Figure 21: The figure visualizes the creation of new triangles and vertices, showing which vertices are used in each of the four new triangles.

The process of creating new triangles was repeated for each intersected triangle. Once this process was completed, the new list of triangles, where the intersected triangles had been excluded and the new triangles have been added, was converted to an array and set as the triangle array for the terrain mesh. This process was then repeated a number of times equal to the resolution modifier.

The reason that cylinder-triangle intersection tests were performed again once the new triangles had been created was that it was likely that only some of those new triangles would be intersecting with the ditch, since the new triangles were not as large as the old ones. That meant that not as many new triangles would need to be created if intersection tests are performed again. This may have reduced performance for the generation of the ditches, but the resulting terrain mesh was more optimised, since it had fewer triangles.

3.10.5 Projecting Terrain Mesh Vertices

Once the terrain mesh had a satisfactory resolution, the vertices could be projected to create the ditches. This was a two step process, first a verification to confirm that the vertex was indeed intersecting with the cylinder representing the ditch. Then, the actual projection of the vertex. If the resolution of the ditch had been increased, then only the vertices of the triangles that were previously found to be intersecting with a ditch were tested and projected. Otherwise, every vertex in the mesh was tested and projected.

The test was a cylinder-point intersection test. By calculating the dot product between the start vertex of the cylinder and the point, a cap test was performed to see that the point was actually between the start and end vertex of the cylinder. Once this had been confirmed, the distance between the vertex and the centre line was calculated. If this distance was no greater than the radius, then there was an intersection [43].

Once the vertex was verified to be intersecting with the ditch, it could be projected. To project the vertex, the point on the centre line closest to the terrain vertex needed to be found so that the projection would be perpendicular to the centre. This was done to create a more uniform ditch. The height of the terrain vertex was compared to the point on the centre line, and if the terrain vertex's value was greater, its height was set to be equal to the point on the line. This was done to avoid projecting the point upwards, creating a small hill instead of a ditch. The slope between the terrain vertex and point on the centre line was calculated, but first a small height-offset was also applied to the point on the centre line of the cylinder. This was done to avoid projecting the terrain vertex exclusively in a lateral direction, in the case when the point on the line and terrain vertex have equal, or almost equal height. The offset was removed after calculating the slope so that the vertex was projected the correct distance. Finally, the terrain vertex was projected from the point on the centre line along the calculated slope with a distance equal to the radius of the cylinder. Figure 22 visualises this process.

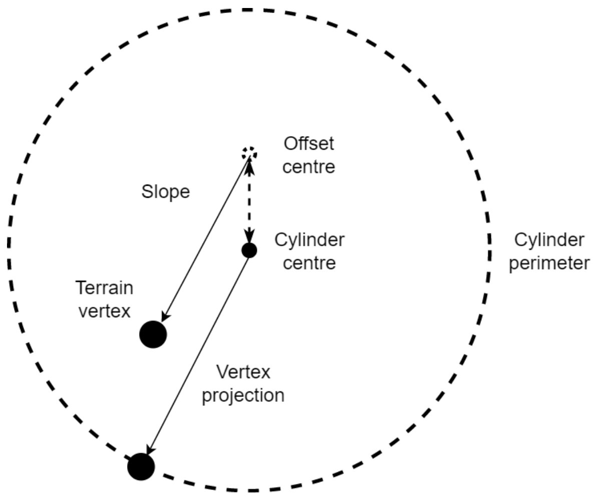


Figure 22: The figure visualizing the process of offsetting the cylinder centre, calculating the slope to the terrain vertex, removing the offset and projecting the vertex.

3.11 Stitching the Terrain Meshes Together

When the noise function had been applied to the terrain meshes, it inevitably caused gaps between them. Fortunately, they shared the same resolution, so stitching them together was not a mathematically complex task. The way stitching was accomplished was by going from the top left terrain mesh to the bottom right mesh. If a terrain mesh had a right neighbour, or one below, it got stitched together with that. In this way, each side between two terrain meshes was checked only once.

For the algorithm, the terrain meshes are contained in what is essentially an array, but Unity makes this slightly more complicated. In the array, the meshes are indexed starting at 0, and each mesh has a right neighbour if $((index + 1) \bmod \#\text{sides}) \neq 0$, and has a neighbour below if $index < (\#\text{meshes} - \#\text{sides})$.

By looping through the outermost vertices stored in a vertex array, they are compared to the outermost vertices of the other mesh. For example, the right-most vertices of the left mesh are compared with (and subsequently stitched together with) the left-most vertices of the right mesh. Similarly, the bottom-most vertices are stitched together with the top-most vertices of the mesh below.

Calculation is then done based on the height difference of two vertices, represented by $\overrightarrow{v}$ and $\overrightarrow{u}$ in this instance.

$$\boldsymbol{v} = \begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} \qquad \boldsymbol{u} = \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix}$$

$$\Delta y = y_1 - y_2$$

$$\boldsymbol{v} = \begin{bmatrix} x_1 \\ y_1 - \frac{\Delta y}{2} \\ z_1 \end{bmatrix} \qquad \boldsymbol{u} = \begin{bmatrix} x_2 \\ y_2 + \frac{\Delta y}{2} \\ z_2 \end{bmatrix}$$

One might also simply set $y_1 = y_2$. This is a computationally less expensive approach, as no arithmetic is needed, and requires only writing to one of the vertex arrays belonging to the terrain meshes. However, the results are not as visually appealing, as it causes sharp edges along the seams between terrain meshes.

At this point in the process, there is an issue that needs to be solved regarding the bottom right corner vertex of each terrain mesh. When it gets updated by the first pass of the algorithm, it does not get its final value, as the vertex it gets matched to will later have to change, as it is on the very right side. Figure 23 illustrates the problem.

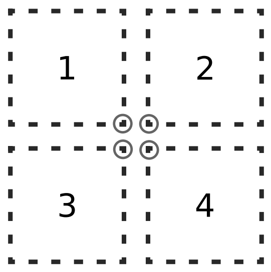


Figure 23: A simplified illustration of how the corner stitching problem occurs. Vertex #1 gets updated with the positions of vertices #2 and #3. Later on in the algorithm, vertices #2 and #3 will get updated with the position of vertex #4. Then, assuming vertex #4 has a different y value, vertex #1 will not close the gap

Running the algorithm one more time does not actually solve this issue, unless, in the second pass, the $y_1 = y_2$ solution is used. Not to mention, running the same algorithm twice is not efficient. Instead, to solve the problem, the second pass uses the $y_1 = y_2$ solution, but only sets the bottom right corner vertex to the value of the matching vertex of the mesh beneath it.

3.12 Tools and Verification

To meet the goals for this project, some third party tools have been used. These tools consist of the chosen programming language and a graphical engine. There is also some tools that has been used before the main work. One of these was the free and open source 3D creation tool, Blender. This tool have been helpful in order to get the most important information from the alembic files that the customer provides this project.

The main tools used in the project include:

- Unity, a game engine first developed to make games for the OS X operating system. It has since been expanded to have support for many more platforms. At the time of writing, 27 platforms are supported by the Unity Engine. It has been developed for ease of use, and has a strong underlying framework, which makes it the perfect tool for prototyping projects involving graphics and games.[44]
- C sharp (C#), an object oriented programming language developed by Microsoft, designed as a modern general-purpose language. It is based on the C++ programming language, but it largely dissimilar, and is instead very similar to Java.[45]

In accordance with the requirements from the customer, the terrain needs to look realistic and it needs to have a good fit with the road. This meant that the major part of the verification was done by visually inspecting the generated models. This was not only done because it was the most accessible option that was still able to produce satisfactory results, but also because of the ease with which the models can be examined during run-time, as seen in figure 24. Here it can be seen how every individual triangle is clearly distinguishable, which makes it easy to spot any problems with the model. In this particular example it is possible to identify a problem with clipping between the road and the terrain, shown by the green triangles that intersect with the white road.

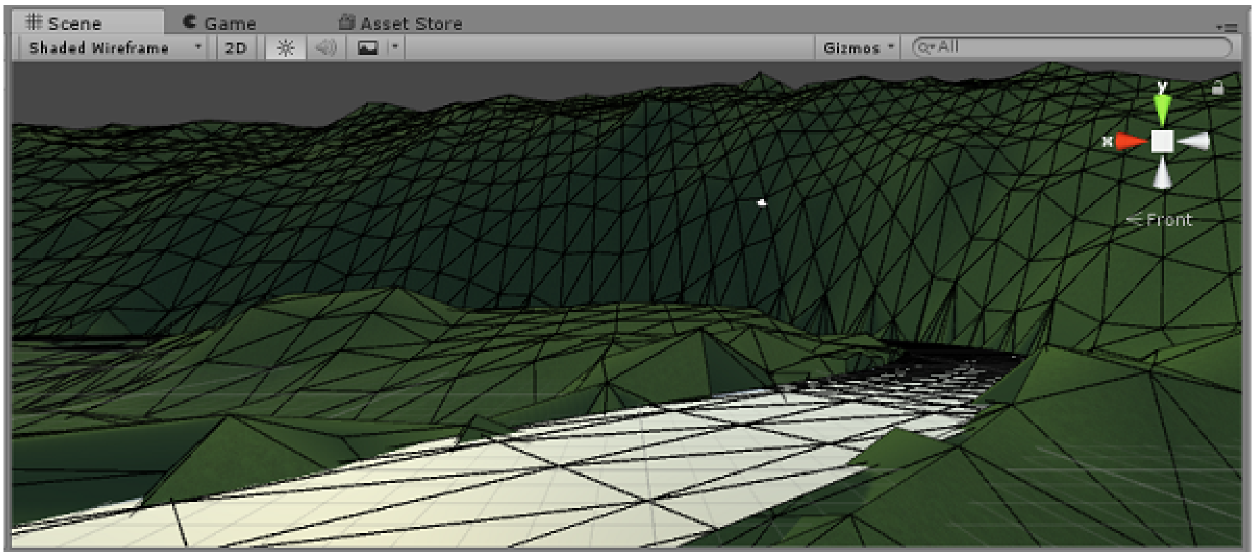
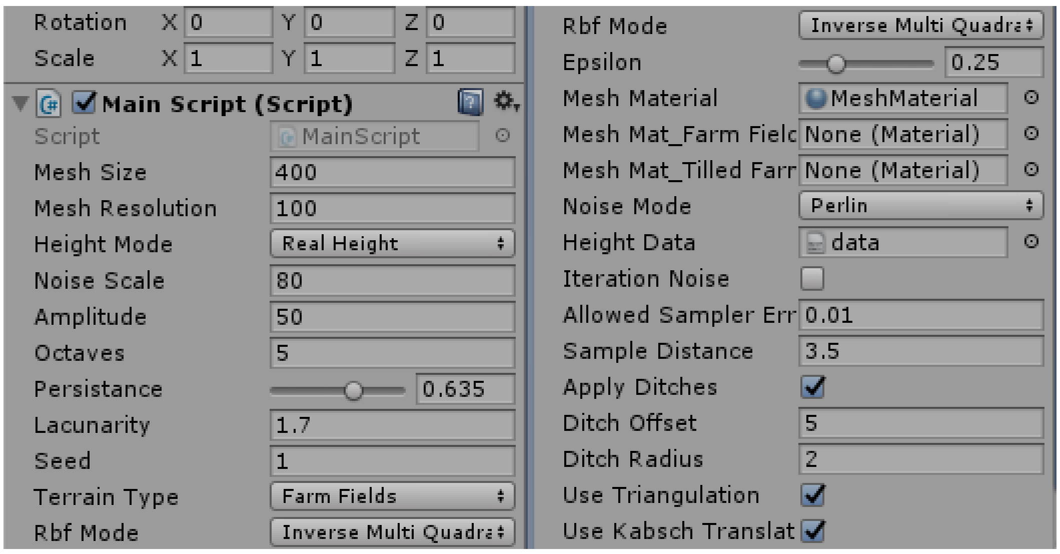


Figure 24: Part of the terrain model and a road model as shown in *Unity* used to verify that the generated terrain looks realistic.

It is also possible to easily create editors to change the different parameters as seen in figure 25. Since there is a lot of fine-tuning parameters involved in creating realistic terrains, it is very beneficial to be able to change these without making changes in the source code.



Rotation	X	0	Y	0	Z	0
Scale	X	1	Y	1	Z	1

Main Script (Script)

Script	MainScript
Mesh Size	400
Mesh Resolution	100
Height Mode	Real Height
Noise Scale	80
Amplitude	50
Octaves	5
Persistance	0.635
Lacunarity	1.7
Seed	1
Terrain Type	Farm Fields
Rbf Mode	Inverse Multi Quadra
Rbf Mode	Inverse Multi Quadra
Epsilon	0.25
Mesh Material	MeshMaterial
Mesh Mat_Farm Fielc	None (Material)
Mesh Mat_Tilled Farr	None (Material)
Noise Mode	Perlin
Height Data	data
Iteration Noise	☐
Allowed Sampler Err	0.01
Sample Distance	3.5
Apply Ditches	✓
Ditch Offset	5
Ditch Radius	2
Use Triangulation	✓
Use Kabsch Translat	✓

Figure 25: The editor created for changing various input parameters for the terrain generation.

4 Result and Discussion

This section looks at the final results, and discusses how well the result and the implemented functionality matches what was requested by the customer. It also discusses a few more general points about the chosen methods, and the value of the developed software. In figure 26 the final result is shown with a terrain that fits to the road, ditches, and a non-flat terrain that looks realistic.

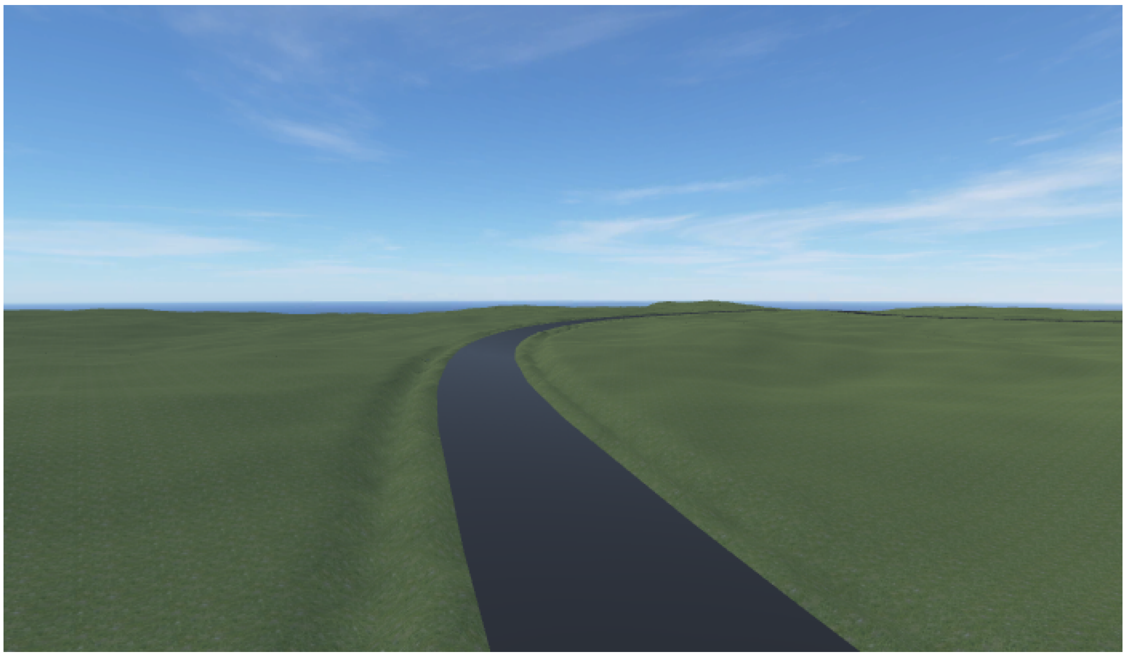


Figure 26: An image showing what a terrain model can look like using many of the implemented features.

4.1 Implemented Features

Methods that played a large role in the visual aspect of the final model are shown and discussed here. The discussion mainly focuses on problems that have not yet been solved with a feature, and ways in which the feature could be extended.

4.1.1 Realistic, Hilly Terrain

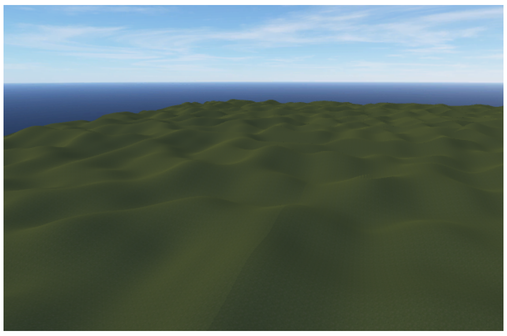


Figure 27: Perlin Noise, 1 octave

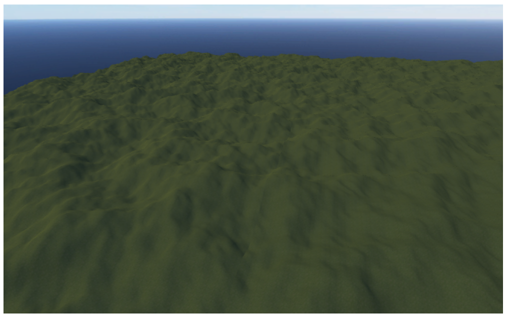


Figure 28: Perlin Noise, 3 octaves

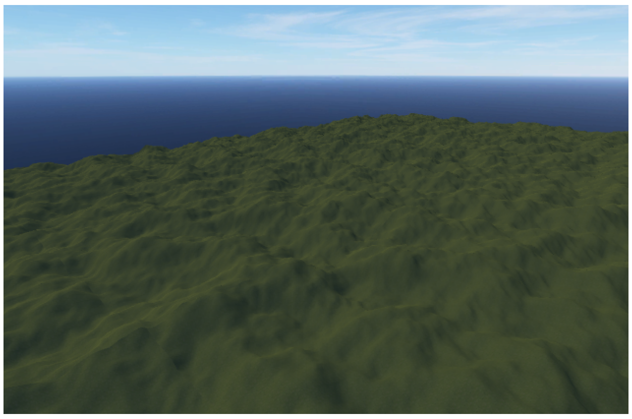


Figure 29: Perlin Noise, 5 octaves

Figure 27 shows the terrain model with Perlin noise applied, using only one octave. This creates the appearance a hilly, and smooth terrain. Increasing the number of octaves, as seen in figure 28 with three octaves and figure 29 with five octaves, increases the amount of detail in the terrain model. Increasing the number of octaves beyond this, while not shown here, does not have much effect, if viewed from far away. This is simply because the scale of the changes are too small to be visible.

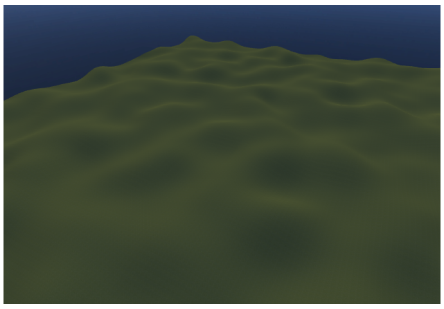


Figure 30: Simplex, 1 octave

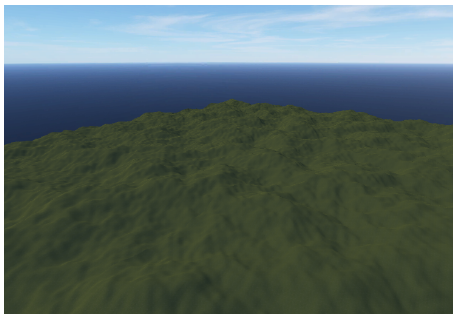


Figure 31: Simplex, 3 octaves

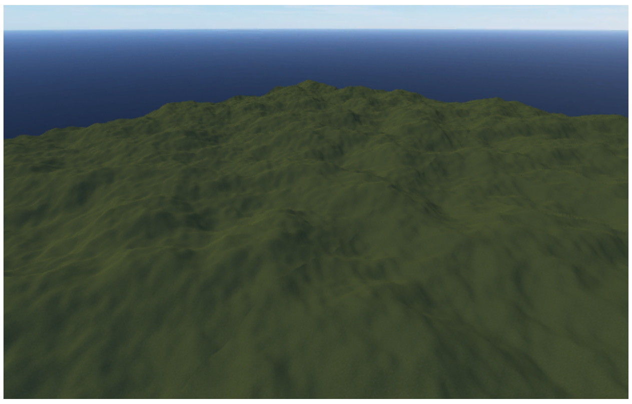


Figure 32: Simplex, 5 octaves

Figure 30 shows the Simplex noise with one octave. Simplex appears to have a less uniform look, which is good for the purpose of generating a realistic looking terrain model. Figure 31 shows the same noise with three octaves, and figure 32 with five octaves, showing the increased detail. Interestingly, the implementation used for Simplex actually performed worse than Perlin Noise. This contradicts the initial statement that Simplex noise should be faster. The reason for this could possibly be related to the noise implementations, or perhaps limitations with C#.

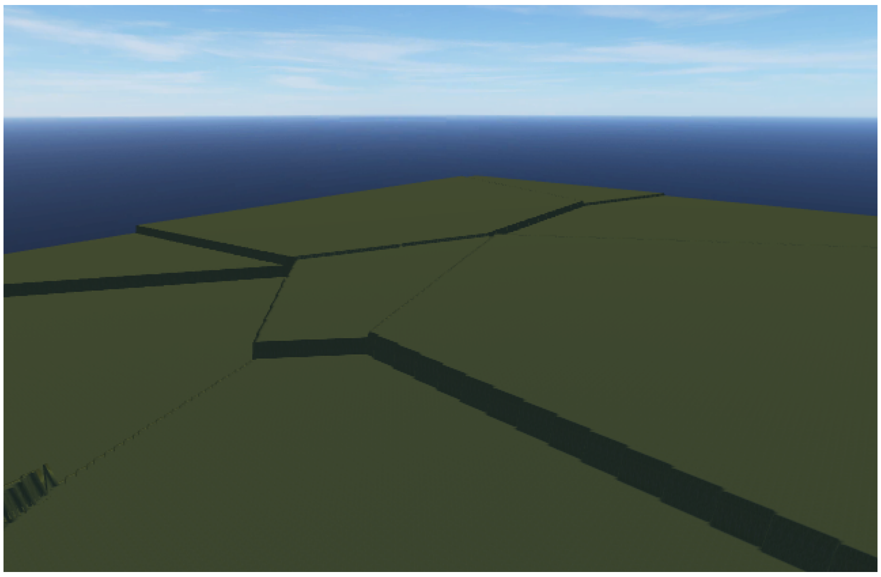


Figure 33: Worley Noise

Figure 33 shows what the terrain looks like with Worley noise. This creates more plateau-like terrain. Ideally, this is best implemented when combined with another type of noise. The Worley noise could also be used to generate terrain types which would be difficult to generate with Perlin or Simplex noise, like mesas for example.

Currently the noise applied to a mesh is not affected by what type of mesh it is or by its distance to the road. Because the customer primarily was interested in a fairly homogeneous testing environment, this was not a major problem and the same noise could be applied to all meshes. This does however create some issues with the ditches.

4.1.2 Realistic Terrain-Road Transition

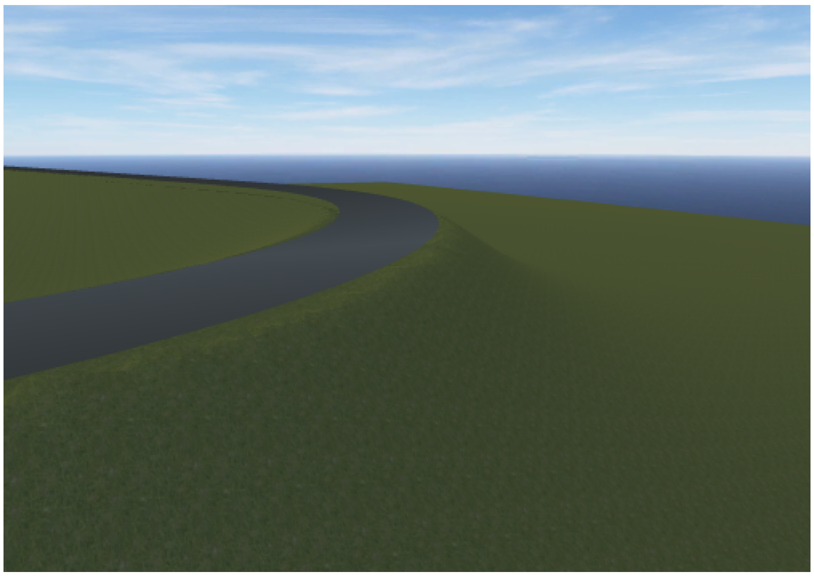


Figure 34: Inverse Multi-Quadratic RBF interpolation with $\epsilon = 0.2$.

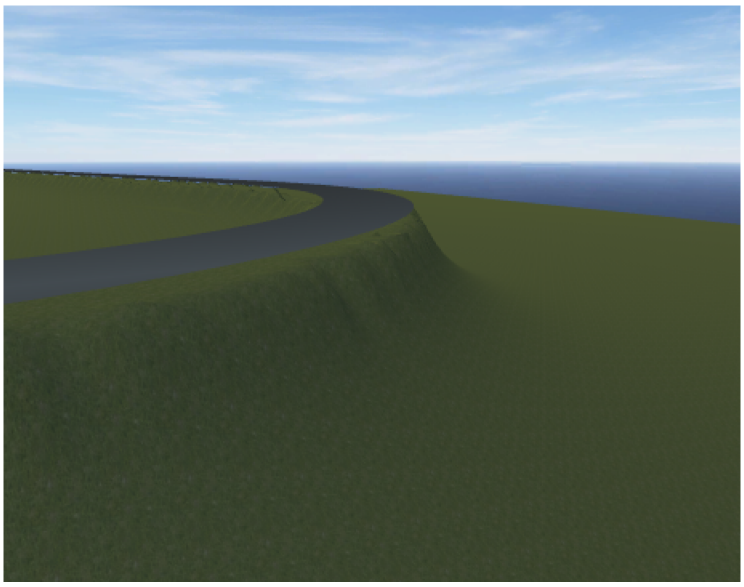


Figure 35: Inverse Multi-Quadratic RBF interpolation with $\epsilon = 1$.

The interpolation provided by the Inverse Multi-Quadratic RBF creates a very smooth curve between the road and the terrain. As shown in figure 34, a lower ϵ creates a much smoother transition. Figure 35 show a much more harsh transition, which is gained by increasing ϵ.

When compared to real life, it's easy to imagine that more smooth surfaces, such as for lower ϵ values, creates a more natural feeling. This might be because it blends in with the terrain quite easily. This is opposed to the higher ϵ values, where the transitions are harsh. This, however, is not necessarily unrealistic, as it is often the case that mountains are naturally steep. Since the RBF method works on any set of input points, it is easy to expand the morphing functionality to work with virtually any 3D objects, not just roads.

Some issues arose when using the RBF morphing with the roads. Some vertices close to the road edge would be offset way too much, which lead to spikes or holes appearing in the ground, as can be seen in figure 36. The source of the problem was later confirmed to be the distance between road samples. After assuring that no samples were too close to each other, most of the spikes were eliminated. In some cases, however, the problem remained to some extent. This was probably due to the meshes still having unwanted vertices, even after the mesh cleaning process. Having to take fewer road samples was unfortunate, since many samples would, in theory, make the transition between terrain and road more precise, thus producing a better result.

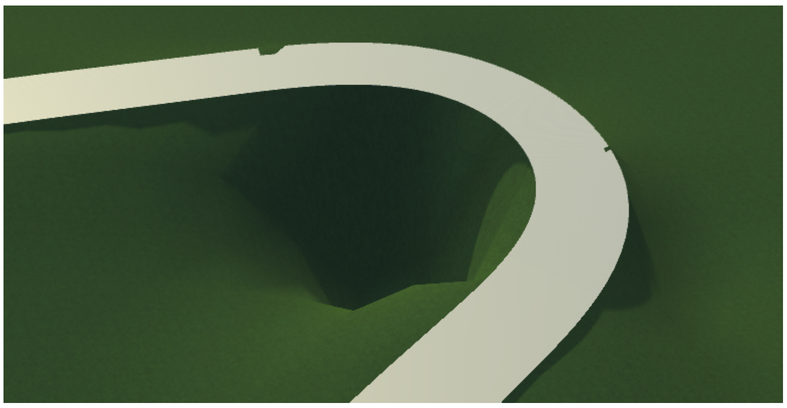


Figure 36: A glitch in the the RBF interpolation creating a downward spike in the terrain where road samples were too close to each other.

4.1.3 Additional Terrain Triangles Along the Road

The creation of additional terrain triangles along the road was implemented with the goal to adhere the terrain properly to the road.

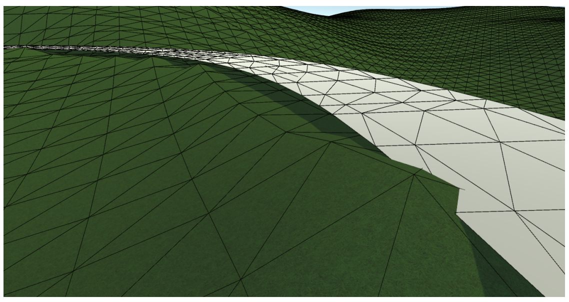


Figure 37: Triangulation deactivated.

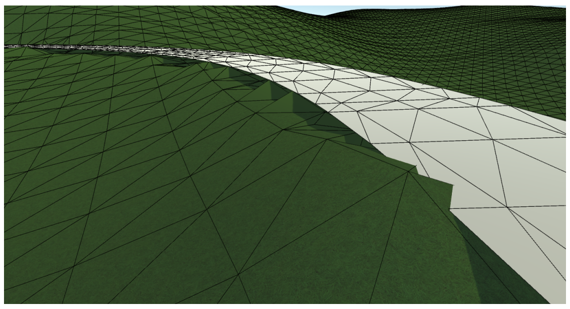


Figure 38: Triangulation activated.

Unfortunately, as one can see by comparing figure 37 and 38, the method didn't quite produce the wanted result. While creating contact points where the terrain vertices connect properly to the road, in most places there were still gaps between the road and the terrain. This was due to the road samples being too far from each other for the method to produce a satisfying result. Taking more road samples might have given a better result in theory, but in reality it resulted in spike glitches caused by the RBF interpolation (see 4.1.2).

Since the results of the triangulation method were below expectations, and because of prioritisation of other issues, the functionality was removed in the end product.

4.1.4 Generating Different Kinds of Terrain

This solution depends on a few assumptions to work. Firstly, computation and memory is not a major issue for the customer. Secondly, it is so important that the terrain looks realistic that some manual work is accepted in order to achieve this. Finally, it is preferable to have as little *Unity* specific code as possible since the customer does not use *Unity.*

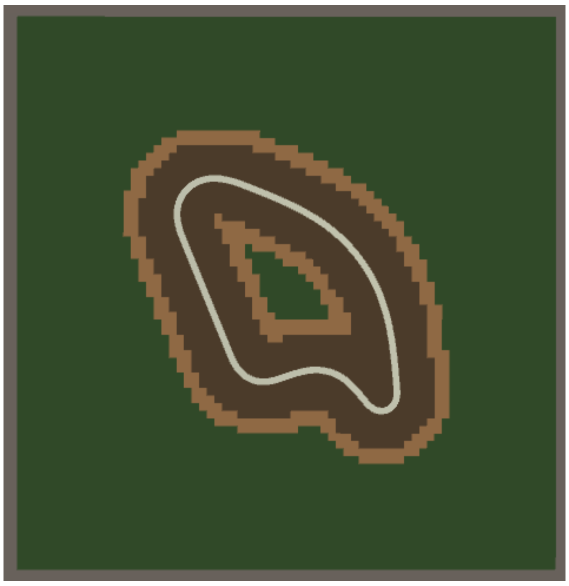


Figure 39: The world type "Forests and Fields". The brown colour represents fields, and the green represents forests. The transition between the areas are represented in light brown.

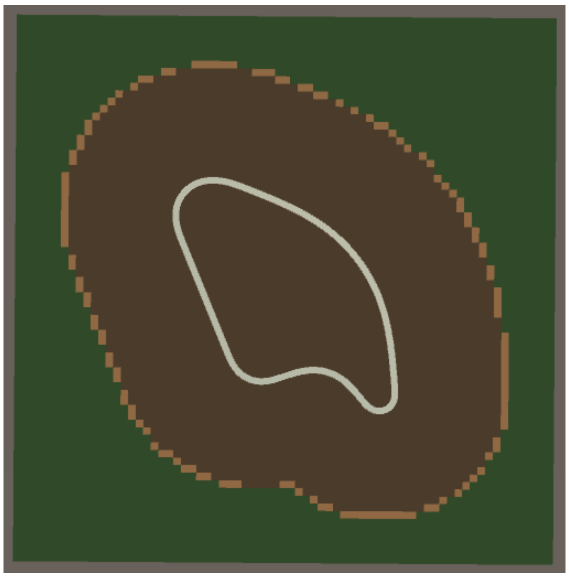


Figure 40: The world type "Mountains Far Away". The brown colour represents forests, while the green, in this case, represents mountainous areas.

In the implemented model rendered in *Unity*, the terrain is only coloured to distinguish between the terrain types in order to verify the method, meaning that it is not much to look at. However, the rendered model that is sent to the customer does not contain the colours. Instead, at the request of the customer, the name of each mesh is used to distinguish between the types. This is because the customer wants to populate the terrain with objects such as trees or crops after it has been generated, to make the entire scene and not just the terrain look realistic, and with each mesh named after a specific type of terrain it is easy to loop through all meshes and populate them accordingly.

Figure 39 shows a top-down view of the world generated with different terrain types, showing how the terrain types are distinguished based on their distance from the road. Figure 40 shows a different preset, which has been generated with different settings. This shows that it is possible to devise multiple types of terrain, depending on what the customer wants.

As seen clearly in figure 39, there are light brown meshes between the green ones and the darker brown ones. These represent the area in between two types of terrain, which is called the transition terrain type here. This terrain type exists in order for the customer to be able to make realistic transitions between different kinds of terrains. It requires some manual work, but it is possible to obtain smooth transitions and make sure that individual meshes are not distinguishable.

Currently the implemented terrain types are "field", "forest", "mountain" and "transition" with a few different combinations defined. Evidently, not a large number of different combinations of terrains are currently implemented, and the algorithm only uses the distance from a mesh to the road, and not for example the type of its neighbours which would be reasonable. This is, however, in accordance to the demands of the customer since, as previously noted, the developed software is used to prove that the concept is solid, and to be a platform to continue building upon.

4.1.5 Realistic Ditches

The customer wanted to simulate road ditches. It was thought that the ditches would need to be fairly round to appear realistic, which is why cylindrical ditches were decided upon. Furthermore, the ditches were to be placed along the road, on both sides of the road.

Overall the resulting ditches looked promising. The placement of ditches worked well, there were no problems with the ditches intersecting the road, as long as an accurate road outline was provided. The shape of the ditches seemed suitably round. The fact that realistic ditches are quite small required a fairly high mesh resolution. While there wasn't an explicit request for a specific resolution, the implemented method for increasing the resolution was able to produce decently realistic results. There were however some issues with integrating the ditches with noise in the terrain. The ditches may not fit every type of terrain, but they are a good starting point for future development.

Figure 41 and 42 show how the placement of the ditches work, they both depict the same ditch, figure 41 shows the triangles in the terrain mesh, figure 42 shows the shaded version of the same terrain and also includes the road. The ditches have a resolution modifier of 4.

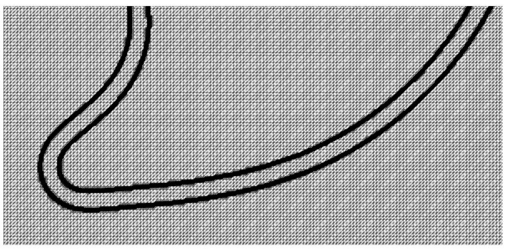


Figure 41: A wireframe of the terrain. The darker areas of the image are a result of a higher concentration of triangles.

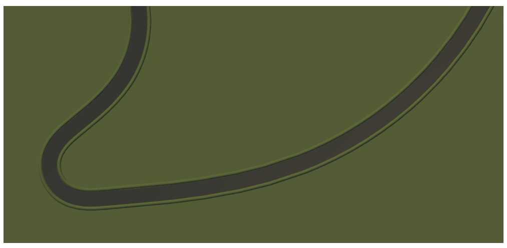


Figure 42: The same terrain as figure 41 but shaded and with the road included.

Figure 43-46 shows a more detailed view of ditches, showing the difference of an increased ditch resolution modifier. The modifier scales exponentially with base 4. A resolution modifier of 1 will turn one triangle into four smaller triangles. A modifier of 2 will turn the initial triangle into four new triangles, and then turn those four triangles into four new triangles each, resulting in a total of 16 new triangles.

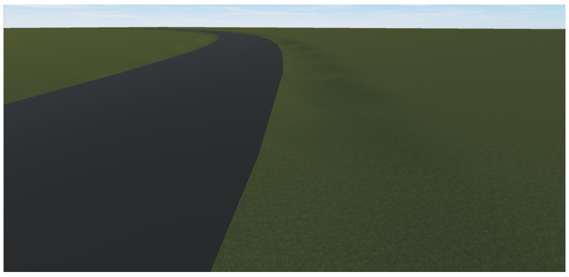


Figure 43: Ditch without an increased resolution. The ditch is barely noticeable, looking into the distance, a few vertices in the terrain can be seen to have been depressed slightly.

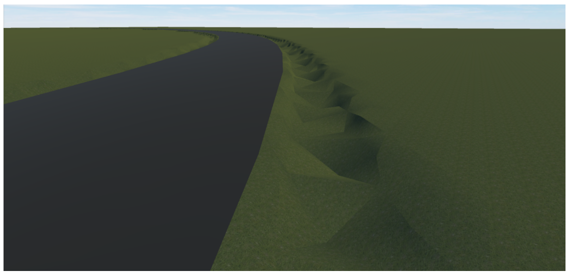


Figure 44: Ditch with a resolution modifier of 1. The ditch is noticeable, but very rough and uneven. The cylindrical shape that the ditch is supposed to follow does not come across.

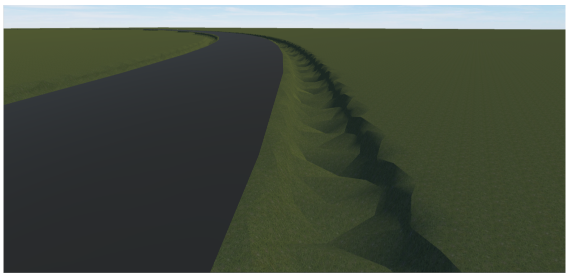


Figure 45: Ditch with a resolution modifier of 2. The ditch can now be seen to clearly follow a cylindrical shape, it is somewhat rounded but still quite rough-looking.

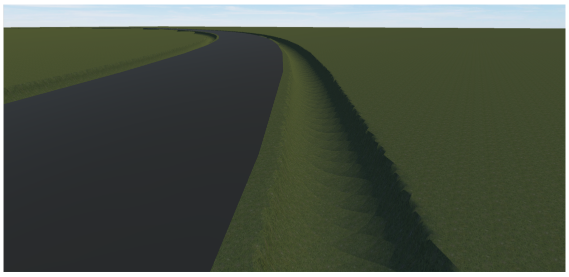


Figure 46: Ditch resolution modifier of 4. The ditch is round and looks fairly even.

The figures 47 and 48 show what the ditches can look like on both the outside and inside of a curve.

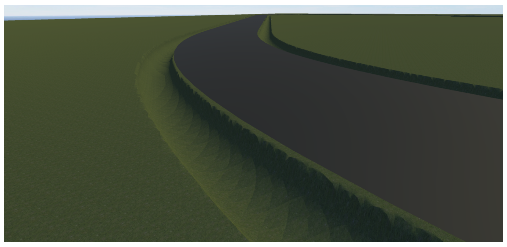


Figure 47: Outer curve with a ditch resolution modifier of 4.

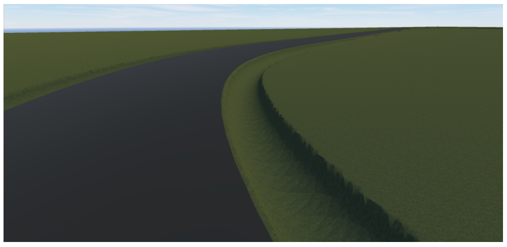


Figure 48: Inner curve with a ditch resolution modifier of 4.

There are two significant issues that still remain with the ditches. Both only seem to appear when there is a substantial amount of noise near the road. The first is that in some places where the terrain meshes meet, there is some clear tearing in the terrain near the ditches. The reason for why these tears are not stitched together is that after the ditch resolution is increased, the vertices in meshes are longer uniformly positioned, which interferes with the stitching algorithm.

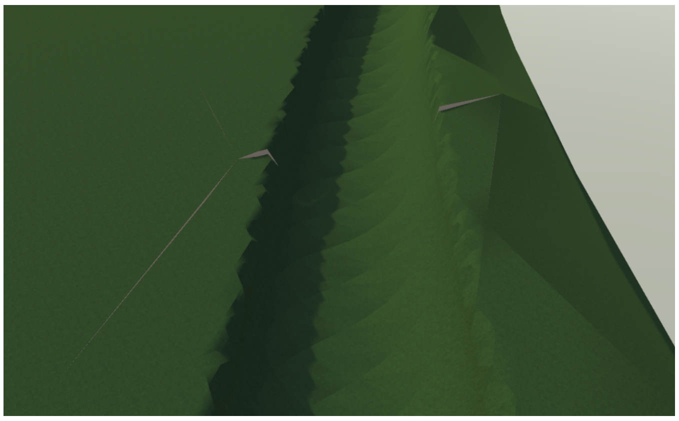


Figure 49: An image of the tearing issues that appear where meshes meet, near ditches.

The second issue is that when noise causes the height of the terrain to be significantly higher than the road, as the ditches have the same height as the road, the resulting ditches can look quite rough. Furthermore, they can become unexpectedly deep, which may look unrealistic. There does not seem to be an easy solution to this, since the problem is not so much the ditch itself as it is a problem with the noise getting too close to the road. One possible solution would be to use the RBF to not only fit the terrain to the road, but also to the ditches.

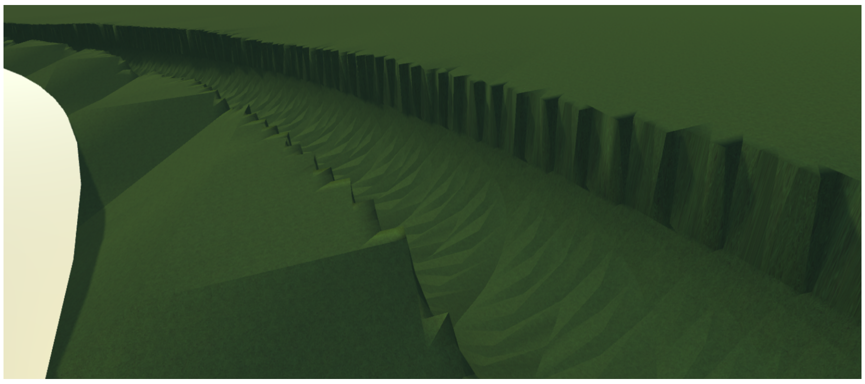


Figure 50: An image of the fairly harsh looking and unintentionally deep ditches.

The main benefit of only using one type ditch shape was that only one type of test was needed when increasing the ditch resolution and when projecting the vertices. This saved a lot of time that would otherwise need to be spent on researching intersection tests and implementing and debugging code. There are of course benefits to using several types of ditch shapes.

Semi-circular ditches can be combined with cylindrical ditches to avoid intersection between ditch segments. This could also result in smoother ditches, as semi-circles are better fit to hand sharp angles, especially 90-degree angles, which can occur when roads intersect. Semi-circular ditches were implemented and experimented with but ultimately excluded from the final product. While there is certainly potential for semi-circular ditches, using cylindrical ditches produced adequate results, and so they were excluded mainly because of the time it would have required to implement additional intersection tests. There might also be problems with harsh vertical transitions as the semi-circles are not rounded towards the centre, which may make them unsuitable to be used in combination with cylindrical ditches.

An alternative method that was initially used for intersection testing was a cylinder-point intersection test. This test was significantly faster than the cylinder-triangle intersection test, in part because the method for cylinder-point test requires less computations but also because there are only half as many vertices as there are triangles. The problem with this method was that the point-intersection could not detect an intersection with a triangle edge. This caused tearing in the terrain, which is why it was replaced by the cylinder-triangle intersection test, which detects intersections with a triangle edge.

4.1.6 Terrain That Covers the Entire Road

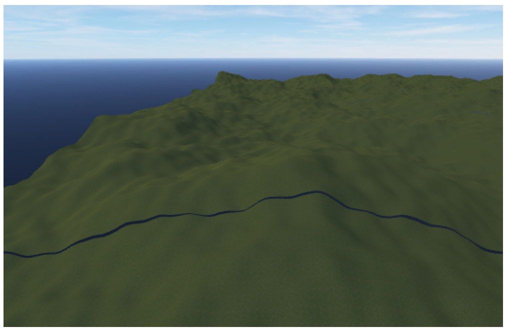


Figure 51: The multiple terrain meshes with stitching turned off, with the gaps visible.

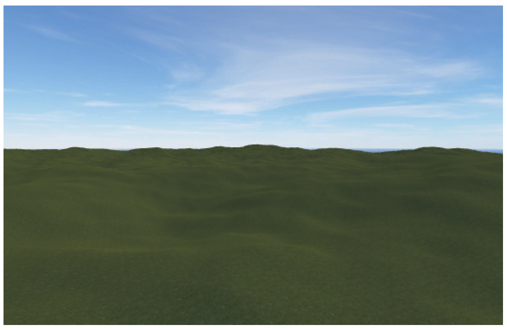


Figure 52: The multiple terrain meshes with stitching turned on, with no gaps visible.

Stitching was implemented early into the implementation of creating multiple terrain meshes, as the gaps between terrain meshes were immediately noticeable. The first version of the stitching was a simple, non-interpolated stitching, where the height of one vertex was immediately set to be of equal height to its corresponding vertex in the other mesh.

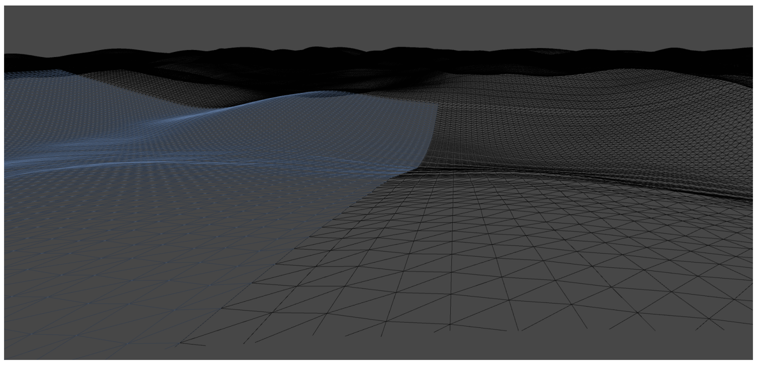


Figure 53: A wireframe view of the terrain meshes, with one mesh selected. This clearly shows where the meshes meet.

The second version of the stitching instead used the difference between the two, and dividing it by half. This meant that the two sides would simply meet in the middle causing a much smoother and more natural transition. This is shown in figure 53.

The stitching implementation is not perfect. For example, an increase in triangles caused by the implemented triangulation or ditches, will actually interfere with the stitching. More stitching techniques exist for non-uniform meshes, but they are outside the scope of this project, as they are very mathematically complex.

A smooth transition is required for a realistic terrain. One might circumvent the stitching by increasing both the resolution and size of the mesh, so that multiple terrain meshes are not needed. However, as discussed earlier, due to the vertex limit per mesh in *Unity*, doing that will decrease the maximum number of vertices for the terrain as a whole. This makes stitching an integral part of making the terrain more realistic. Of course, the stitching works best when it is not noticeable at all.

4.1.7 Using Real World Height Data

The customer expressed interest in being able to generate a terrain that would follow real world height data. As seen in figure 54, it is possible to generate terrain that exactly follows the input data. Although the input data was created only to illustrate this and not to actually look realistic, we can clearly see that the algorithm works. The functionality is a proof of concept and would have to be developed further in order to achieve a realistic relationship between the road and the terrain.

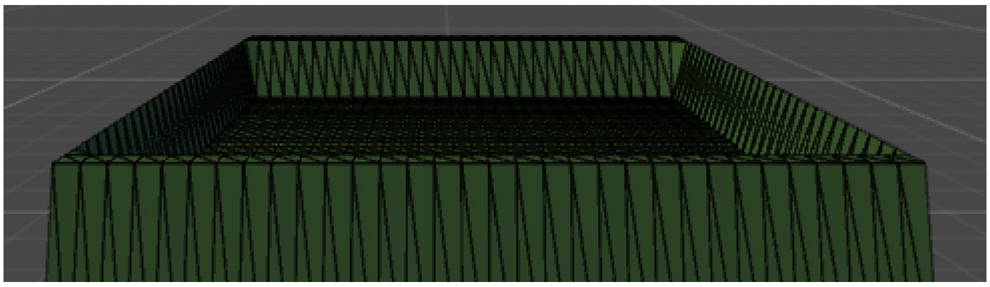


Figure 54: An illustration of how real height data was used to generate the terrain. The input data was read from text file as a matrix, with each element along the border set to the value 200, with the other entries set to 100, which is the same height as the road. The heights from the input file are also scaled with a factor of 10, in order to illustrate the differences clearer.

When actual height data is used, much more realistic results are obtained as seen in figure 55. In this case the input data is read as an 1250*1250 matrix.

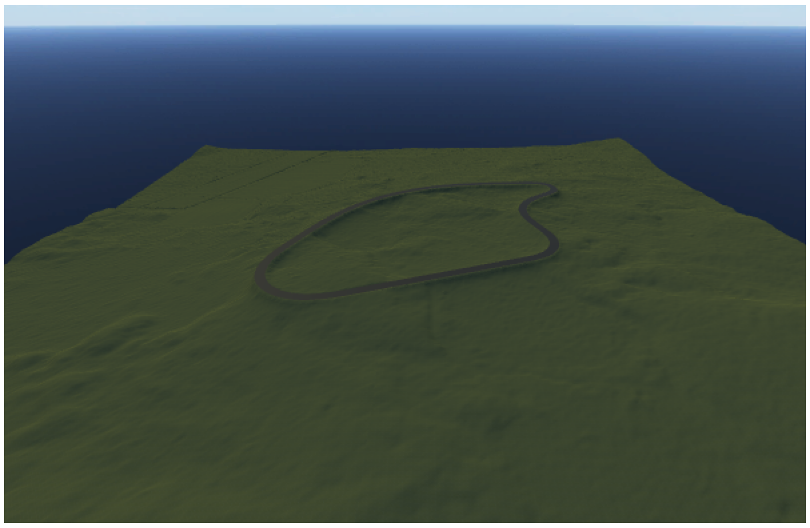


Figure 55: Terrain generated from actual Height Data

There are however some limitations to the way generation from height data has been implemented. The size of the input data is not taken into consideration at all when the terrain is generated. This means that the only way to get a result exactly matches that of the real world, is to make sure that the height data has the same resolution as the generated terrain, so that each entry in the data matches a vertex in the terrain. This means that it is currently not possible to, for example have an entry in the height data affect several vertices.

4.2 Implications for a More Sustainable Development

There are several arguments for how autonomous vehicles contribute to a more sustainable development, both regarding the economical, social, and environmental perspectives[1]. This includes lower emission, less traffic accidents, less time spent on driving and so on. There is however a significant amount of work left before this technology will be considered safe enough for public use[1], and hopefully this software will be able to help, if ever so slightly, in speeding up the process. Still the actual implications of the developed software is fairly slim since at best it will help by slightly advancing the development of another technology that is believed to have a positive effect for a more sustainable development.

4.3 Handling Input Data

The input files given by the customer that were used in the project were of Alembic format. The customer also provided XML files following the OpenDRIVE format, which were the original files that the Alembic files were based on. Instead of having road meshes, the OpenDRIVE files contained logical descriptions of the roads. It was decided that working with the Alembic files would be easier, and basing the algorithms on meshes would give a more general solution. However, since the meshes in the Alembic files contained some artefacts and irregularities, the end result might have been more consistent and with fewer visual glitches if road edge samples had been taken directly from the OpenDRIVE files.

The process of cleaning the road meshes in Blender with a Python script had some issues. The removal of redundant meshes were based on numbers in the names of the meshes. The numbers were however not completely consistent, leading to some of the redundant meshes being kept, which lead to problems later on. Furthermore, several of the meshes were structured in a seemingly arbitrarily fashion, which was not compatible with the developed methods of retrieving the road outline. This meant that several of the roads were completely unusable, making it difficult to test more complex roads, like a road with several lanes.

4.4 Drawbacks with Using *Unity*

Since the project was developed using the Unity engine, some limitations had to be taken inte account. One critical limitation in the Unity engine was the maximum amount of vertices in a mesh (65,535). This meant that the mesh couldn't have too high resolution, which in some cases could be troublesome, for example when creating high-resolution ditches. The problem was avoided by splitting the terrain into multiple smaller meshes and assuring that the ditches didn't have too high resolution. The problem was not completely solved however, and manual calibration was needed to ensure that the combination of mesh size, mesh resolution and ditch resolution modifier did not cause too many vertices to be created in a mesh.

Another limitation with the Unity engine was the support of exporting Alembic files, which the program couldn't handle natively. This was however solved by using the third-party plugin *AlembicImporter*[29].

5 Conclusion

On behalf of the customer Volvo Cars, software for generating realistic terrain models for testing the cameras of self-driving cars was developed. The software accepts input in the form of a 3D-mesh representing a road, and procedurally generates a terrain model around it. Noise is applied to to the mesh to make the terrain model mimic real world terrain, the terrain model is fitted to the edges of the road, and ditches are generated and placed next to the road. The separate meshes are also named depending of what kind of terrain they are supposed to mimic, in order to make it easy to populate the meshes with for example trees or crops, and it is possible to base the generated terrain model on height data to simulate real world areas.

Unity was used both to generate the terrain model, and to verify the correctness of the solution. The finished result fulfilled the requirements from the customer, but is more a proof of concept than a completely finished solution, and needs to be expanded upon. Functionality in need of improvement includes adjusting the real height data to fit the size of the generated terrain and adding more terrain types, and combinations of terrain types. It would also be beneficial to make the ditches look more realistic when placed on terrains with a lot of noise applied and to be able to use more than one noise map for the generation of the terrain.

For future research, it would be interesting to extend the described solution. The customer expressed interest in several additional features, like being able to generate mountains next to the road, as well as tunnels through mountains. There was also an expressed interest in being able to test scenarios in rural areas, so the generation of a realistic city surrounding a road, or a network of roads. For more rigorous testing of different scenarios, the algorithm for naming the meshes would also need to be extended to handle more than the few cases currently implemented. These are areas that the customer has also expressed interest in, but there was not enough time to research them further.

References

[1] C. Katrakazas, M. Quddus, W.-H. Chen, and L. Deka, “Real-time motion planning methods for autonomous on-road driving: State-of-the-art and future research directions,” *Transportation Research Part C: Emerging Technologies*, vol. 60, pp. 416–442, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0968090X15003447

[2] S. R. Ellis, “What are virtual environments?” *IEEE Computer Graphics and Applications*, vol. 14, no. 1, pp. 17–22, jan 1994. [Online]. Available: http://ieeexplore.ieee.org/document/250914/

[3] N. Shaker, J. Togelius, and M. J. Nelson, *"Why use procedural content generation" in Procedural Content generation in Games: A Textbook and an Overview of Current Research.* Springer, 2016.

[4] S. W. David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, “Texturing & Modeling - A Procedural Approach.” [Online]. Available: https://books.google.se/books?id=F4t-5zSsqr4C&printsec=frontcover#v=onepage&q&f=false

[5] M. Hendrikx, S. Meijer, J. V. D. Velden, and A. Iosup, “Procedural content generation for games: {A} survey,” *Tomccap*, vol. 9, no. 1, p. 1, 2013. [Online]. Available: http://doi.acm.org/10.1145/2422956.2422957

[6] Lucasfilm, “Alembic,” 2010. [Online]. Available: http://www.alembic.io/

[7] F. Huizinga, R. Van Ostaijen, and A. Van Oosten Slingeland, “A practical approach to virtual testing in automotive engineering,” *Journal of Engineering Design*, vol. 13, no. 1, pp. 33–47, 2002. [Online]. Available: http://www.tandfonline.com/action/journalInformation?journalCode=cjen20

[8] M. Aeberhard, S. Rauch, M. Bahram, G. Tanzmeister, J. Thomas, Y. Pilat, F. Homm, W. Huber, and N. Kaempchen, “Experience, results and lessons learned from automated driving on Germany’s highways,” *IEEE Intelligent Transportation Systems Magazine*, vol. 7, no. 1, pp. 42–57, 2015. [Online]. Available: http://ieeexplore.ieee.org/document/7014396/

[9] Y. I. H. Parish and P. Müller, “Procedural Modeling of Cities,” in *28th annual conference on Computer graphics and interactive techniques*, 2001, pp. 301–308.

[10] E. Galin, A. Peytavie, N. Maréchal, and E. Guérin, “Procedural generation of roads,” *Computer Graphics Forum*, vol. 29, no. 2, pp. 429–438, may 2010. [Online]. Available: http://doi.wiley.com/10.1111/j.1467-8659.2009.01612.x

[11] D. M. De Carli, F. Bevilacqua, C. T. Pozzer, and M. C. D’Ornellas, “A survey of procedural content generation techniques suitable to game development,” *Brazilian Symposium on Games and Digital Entertainment, SBGAMES*, pp. 26–35, 2011.

[12] R. M. Smelik, K. J. De Kraker, S. A. Groenewegen, T. Tutenel, and R. Bidarra, “A survey of procedural methods for terrain modelling,” *3AMIGAS - 3D Advanced Media In Gaming And Simulation*, no. June 2015, 2009.

[13] R. Smelik, T. Tutenel, K. J. de Kraker, and R. Bidarra, “Integrating procedural generation and manual editing of virtual worlds,” in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, 2010, pp. 1–8.

[14] I. Parberry, “Designer Worlds: Procedural Generation of Infinite Terrain from Real-World Elevation Data,” *Journal of Computer Graphics Techniques (JCGT)*, vol. 3, no. 1, pp. 77–78, 2014. [Online]. Available: http://jcgt.org/published/0003/01/04/

[15] J. McCrae and K. Singh, “Sketch-based path design,” *Proceedings - Graphics Interface*, pp. 95–102, 2009.

[16] H. Hnaidi, E. Guérin, S. Akkouche, A. Peytavie, and E. Galin, “Feature based terrain generation using diffusion equation,” *Computer Graphics Forum*, vol. 29, no. 7, pp. 2179–2186, sep 2010. [Online]. Available: http://doi.wiley.com/10.1111/j.1467-8659.2010.01806.x

[17] R. M. Smelik, T. Tutenel, K. J. de Kraker, and R. Bidarra, “Interactive creation of virtual worlds using procedural sketching,” *31st annual conference of the European Association for Computer Graphics*, pp. 1–4, 2010. [Online]. Available: https://graphics.tudelft.nl/Publications-new/2010/STDB10e/STDB10e.pdf

[18] Mojang, “Minecraft,” 2011.

[19] J. Doran and I. Parberry, “Controlled Procedural Terrain Generation Using Software Agents,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 2, 2010. [Online]. Available: http://ieeexplore.ieee.org/abstract/document/5454273/?section=abstract{&}part=1

[20] Stevo-88, “File:PerlinNoise2d @ commons.wikimedia.org,” 2007. [Online]. Available: https://commons.wikimedia.org/wiki/File:PerlinNoise2d.png

[21] P. Ken, “Perlin fade function,” 2002. [Online]. Available: http://mrl.nyu.edu/~perlin/noise/

[22] K. Perlin, “Noise Hardware,” *SIGGRAPH Course Notes*, 2001.

[23] Steven Woley, “A Cellular Texture Basis Function.” [Online]. Available: http://www.rhythmiccanvas.com/research/papers/worley.pdf

[24] P. Gonzalez and J. Lowe, “CellularNoise,” 2015. [Online]. Available: https://thebookofshaders.com/12/

[25] P. Gonzales and J. Lowe, “The Book of Shaders: Fractal Brownian Motion,” 2016. [Online]. Available: https://thebookofshaders.com/13/

[26] “FractalNoise.” [Online]. Available: https://commons.wikimedia.org/wiki/File:Fractal_terrain.jpg?uselang=sv

[27] M. D. Buhmann, “"Introduction - Radial Basis Functions" in Radial Basis Functions,” in *Radial Basis Functions*, P. Ciarlet, A. Iserles, R. Kohn, and M. Wright, Eds. Cambridge University Press, 2003, ch. 1.1, pp. 2,3,4,5.

[28] M. Mongillo, “Choosing basis functions and shape parameters for radial basis function methods,” *SIAM Undergraduate Research Online, . . .*, pp. 190–209, 2011. [Online]. Available: https://www.siam.org/students/siuro/vol4/S01084.pdf

[29] Unity-Technologies-Japan, “AlembicImporter.” [Online]. Available: https://github.com/unity3d-jp/AlembicImporter

[30] E. B. Koffman and P. A. Wolfgang, *"Implementing the graph ADT" in Data structures: Abstraction and design using Java*, 2nd ed. Hoboken, New Jersey: Wiley, 2010.

[31] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-time rendering*, 3rd ed. CRC Press., 2008.

[32] ——, *"Intersection Test Method" in Real-time rendering*, 3rd ed. Boca Raton, FL: CRC Press, 2008.

[33] R. J. Méndez., “worleyNoise,” 2016. [Online]. Available: https://github.com/ricardojmendez/LibNoiseTutorials

[34] “Lantmäteri.” [Online]. Available: https://www.lantmateriet.se/sv/Kartor-och-geografisk-information/Hojddata/

[35] G. Upton and I. Cook, “barycentric coordinates,” 2014.

[36] “Math.NET Numerics - Linear Equation Systems.” [Online]. Available: https://numerics.mathdotnet.com/LinearEquations.html

[37] E. Haines, *"Point in polygon strategies" in Graphics Gems IV*, 1st ed., P. S. Heckbert, Ed. Academic Press, 1994.

[38] D. Kirk, *"IV.6 Faster line segment intersection" in Graphics Gems III.* Cambridge, MA: Academic Press, 1992.

[39] G. B. Arfken and H. J. Weber, *"Determinants and matrices" in Mathematical Methods for Physicists*, 4th ed. San Diego, California: Academic Press, 1995.

[40] D. Eberly, “Intersection of a Triangle and a Cylinder,” pp. 2–4, 2005. [Online]. Available: https://www.geometrictools.com/Documentation/IntersectionTriangleCylinder.pdf

[41] E. W. Weisstein, “Point-Line distance 3-Dimensional,” 2002. [Online]. Available: http://mathworld.wolfram.com/Point-LineDistance3-Dimensional.html

[42] J. F. Hughes, A. Van Dam, M. Mcguire, D. F. Sklar, J. D. Foley, S. K. Feiner, and K. Akeley, *"Ray-plane intersection" in Computer graphics: Principles and practice*, 3rd ed. Willard, Ohio: Addison-Wesley, 2013.

[43] G. James, “Fast Point-In-Cylinder Test,” 2004. [Online]. Available: http://www.flipcode.com/archives/Fast_Point-In-Cylinder_Test.shtml

[44] Unity, “Unity facts.” [Online]. Available: https://unity3d.com/public-relations

[45] J. Liberty, *C#*. O’Reilly, 2005.