Thesis no: BCS-2016-14

Procedural Terrain Generation Using Ray Marching

Performance Analysis Of Procedural Terrain Generation Using The Fragment Shader

Oscar Roosvall

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Bachelor in Computer Science. The thesis is equivalent to 10 weeks of full time studies.

Contact Information:
Author(s):
Oscar Roosvall
E-mail: osro13@student.bth.se

University advisor:
Prashant Goswami
Department of Creative Technologies

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Context. Ray marching is a ray casting algorithm used to generate image data. In this case this algorithm have been used to generate procedural terrain in realtime.
Objectives. The goal was to understand what parts of procedural terrain generation that are performance heavy.
Methods. A C++/GLSL implementation was made to read and execute the shader in order to allow for measurements to find out what components in the shader that where performance heavy.
Results. After performing multiple tests with changing parameters in the shader code, the performance heavy executions are happening in the far distance where it is hard to difference between terrain and sky, where a huge number of iterations per fragment is spent. The region covers about 20-30% of the screen.
Conclusions. The further a ray has to traverse in a ray marching environment the more iterations it will have to perform before it is complete, this will increase the amount of time spent. Finding a way to reduce the iteration count will greatly improve the performance.

Keywords: Terrain generation, Procedural methods, Terrain rendering, Ray marching.

Contents

Chapter 1

Introduction

When creating large terrains there are a number of ways to do so. You can either do it by hand and spend a long time perfecting your landscape or you can use a procedural method to do this for you. The methods can even be combined. Either by letting an artist create the base terrain that is then modified by procedural methods to enhance the details, or by first generating a base procedurally and then let an artist work on the generated terrain. These methods have their benefits and their shortcomings. As an example, hand made terrain can be build specifically for a game or a scene that a developer or artist is working on. They will have the tools to make it look just perfect for the project they are working on. However this is probably going to take some time. A procedural method however can generate a lot of terrain in just a couple of seconds, even less with the use of graphics cards. But for a procedural method to generate your specific terrain you will need parameters to adjust certain features in the terrain. Depending on what you need, the end result may need a lot of parameters.

Terrain rendering have a lot of applications. It can be used in simulations where the terrain plays a role in how the simulation turns out. Terrain rendering also has a part in how to visualize the real word, being able to render the area around you and plan for structures or other projects for an area. These are some of the areas for terrain rendering and this comes with a couple of difficulties, for example massive terrains. When working with large terrains you need to optimize the rendering of the terrain because it is most likely not possible to render everything at once.

Having one method to render terrain is probably going to depend on the data. For example, you have a massive map over a city, this data is large and needs to be rendered and in order to do so the data is split it up into sections and then optimized. Optimizing in this case could be removing triangles that would not give any details. The data gets smaller and easier to handle and render.

Procedural methods for terrain rendering involve some way to generate the render data in real time. Now some challenges will appear, first data needs to be

generated. This can be done using noise functions for example. One very popular noise function is Perlin noise[1]. By combining different noise levels and using parameters to tweak the settings the terrain will emerge. The next steps will involve some method for transferring the data so that it can be rendered and then the data needs to be managed so that it can render with reasonable performance.

There are pros and cons by using procedural methods for generating terrain. Using methods like these allows you to create large terrains and scenes with relative ease and you will not have to worry about disk space for the data required. On a downside the generation of terrain is going to take some time and it will not be free and easy to do. This is one of the challenges in procedural terrain generation, how to create and render a large terrain in real time.

Chapter 2
Background and Related Work

This chapter cover the background work in the field of procedural terrain generation and different ways to render terrain. The algorithms and render techniques described here are to give a insight in how terrain rendering may be done and are not specifically the one being implemented.

2.1 Rendering Techniques

Terrain rendering can be done in a number of ways. The common way to render anything in a 3D world are meshes consisting of triangles and vertices. This method works fine for small terrains and regular objects, but as the scene grows and become larger the terrain will be too large and the performance drops. In order to be able to keep rendering, the data needs to be culled and reduced on long distances. Kang, et al.[2] accomplished this by dividing the terrain into blocks, performing frustum culling against these blocks and improving the performance on their large scale terrain.

A different rendering technique when dealing with large scale terrains is presented by Gobbetti, et al. [3]. In this case the authors had large scale terrains with very high detail. Their method C-BDAM allows for compression on the binary data being submitted to the GPU, thus reducing the time it takes to commit the data and the commands.

The methods mentioned above uses a rasterization technique and it is possible to render terrain in a different way using ray tracing[4]. In this paper Qu, et al. presents a hybrid between rasterization and ray tracing to render terrain. Their method uses a voxel traversal algorithm to compute coordinates of sampling points. Which would then be used to calculate terrain data. The mentioned benefits of a ray tracing method is that it allows to directly work on the image/Z-buffer and also allowing for rendering of special effects.

A different method to ray tracing is ray marching. It works in a similar way where ray tracing sends out a ray from the camera with a direction and checking

if that ray intersects with any object placed in the scene. If the ray intersected with the object, the color of that object is displayed on the screen. This technique is often used in rendering animations at a great detailed level, including shadows and lights. Ray marching also uses a ray, but this ray is instead traversed. Inigo Quilez website describes how the ray traversing is being performed[5]. As mentioned the ray traversing happens in steps and for each step we check how far the ray have gone and if the ray are inside an object. If it hit, the objects color is returned and displayed on the screen.

2.2 Procedural Methods

Procedural methods is a way to generate data. The data can be generated for almost anything, some procedural methods are terrain generation, cloud generation and fluid generation. Procedural methods are even possible to implement in games, the term then used is often procedural content generation. This section will focus on procedural methods for terrain generation.

2.2.1 Volumes

Geiss explains a method using marching cubes [6]. The data is generated on the GPU using the geometry shader. Inside the shader the marching cubes contains a volume, this volume is then used to produce triangles and represent the scene. The polygons/triangles generated can be controlled with a level of detail using the distance from the camera. The volume is generated based on noise that is fed using textures. These textures can be made by hand or created procedurally using noise functions. A further benefit of this method is that textures can be used to color the terrain generated. If a fully physical world where to be made a downside would be that the objects collision and physics would need to be handled on the GPU as well. Depending on the number of objects this could potentially affect the performance. Figure 2.1 shows an example using this method to generate a terrain. Marching cubes uses as mentioned volumes to represent the terrain.

Another way that uses volumes to generate terrain is a voxel based solution[7]. The method Santamaría-Ibirika, et al generates terrain in voxel volumes that have a large range of parameters to describe the terrain. This could be features such as cave systems, layers that a given material is presented in and so on. The parameters can be changed in real time, allowing for visual results and easier tweaking. They mention that the generation is not simulating the world but is based on it. The benefits mentioned is that they can run in a real time scenario but some of the parts such as vein generation is affecting the performance.

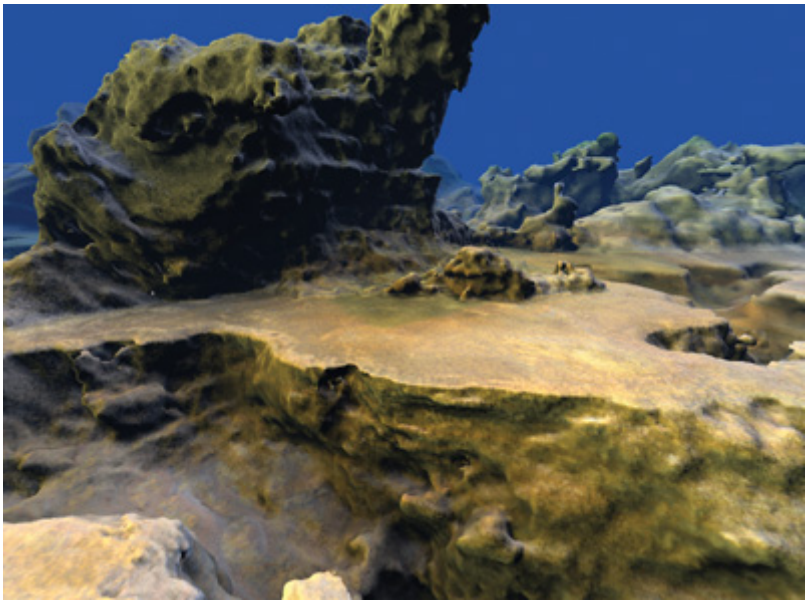


Figure 2.1: Marching cubes example. Image taken from GPU Gems 3, chapter 1 [6].

2.2.2 Height Maps

Helsing and Elster's tool allows for generating height maps in real time[8], where the user can change and tweak the settings and visually see the result. The benefits of a method like this is that as mentioned the user can tweak their terrain as they see fit. Another benefit is that by using height maps the terrain can be exported and used in a different renderer, a game as an example. Using height maps the data does not have to be stored as a mesh but can be used directly on the GPU, where it will generate the terrain when needed. This can be done using tessellation for example. On a down side the height map can only represent a given resolution. This will limit how large terrain a given height map will represent, this could lead to a lot of textures or one very large texture if the terrain is very large.

2.2.3 Tile Based

Using a tile based approach [9], the aim here is to be able to create an endless terrain that is created and rendered in real time. The method is based on "Ridged Multifractal Terrain Model" [10] for generating the height maps. Terrain details are then added using software agents that traverse the terrain. The agents are based on this paper[11]. Grelsson's objective is to find out if terrain can be generated in real time using his method [9]. He reports that the results showed that it was possible to generate terrain in real time but the performance was to low for usage in video games.

2.2.4 Vectors

By using a vector solution[12] the terrain can be adjusted. Hnaidi. et al mention that these vectors are allowed to control features that varies from ridge lines to riverbeds or even cliffs. These vectors can be controlled by the user to create the terrain they want. The benefits with a solution like this is that the user can control the features in a controlled manner, as the vectors are connected to the terrain features they want to modify. Example of terrain features described are roads and cliff sides.

2.2.5 Hydrology

Another different method for terrain generation is hydrological erosion[13]. The terrain is generated by using a simple sketch and parameters to control the generation. The sketch contains features for terrain outline and river parts. Génevaux, et al generation algorithm then takes the input data and generates a river network. The network is then analyzed and used to create watersheds and river trajectories. This data is then used in generation of the terrain. RiverLand is a similar method using river networks for terrain generation[14]. The terrains are realistic looking and provide a wide variety of terrain features such as hills, river valleys and more. The tool allows the user to directly paint onto a sketch canvas that is used to generate the features.

2.2.6 Perlin Noise

A very popular noise function is Perlin Noise[1]. This algorithm generates a pseudo random number for a given input. Pseudo random means that if we give the function the same input twice we will receive the same output. In the article Perlin describes that the usage for this noise function is to create natural looking textures for objects. By using this function you can also create terrain, by repeating the noise and scaling it a number of times. Hart presents a GPU implementation of Perlin Noise[15]. It can be used in the pixel shader for procedural texturing as an example.

2.3 The Selected Method

The rendering method used during this project is ray marching. The implementation will be done in a fragment shader and based of this fragment shader source from `www.shadertoy.com`[16]. The code is provided with the *Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License [17]* in the beginning of the document. The shader contains all the ray marching code and the functions to generate the terrain. All the features of the terrain such as land, trees and water is generated using noise functions based on Perlin Noise[1]. The parameters of the noise functions are the world position that is calculated from the ray marching steps. Depending on the component the parameters are modified differently. As an example land have more complex modification to the noise parameters where as the forest have a simpler modification to the parameters. The selected method of ray marching allows us to generate terrain on the fly in realtime, this all without the use of meshes. The way ray marching works is by sending out a ray from a center of origin, the camera, and moves it in a direction, the view direction. The algorithm using a marching technique to move the ray. Figure 2.2 shows a visual representation on how the ray is marched.

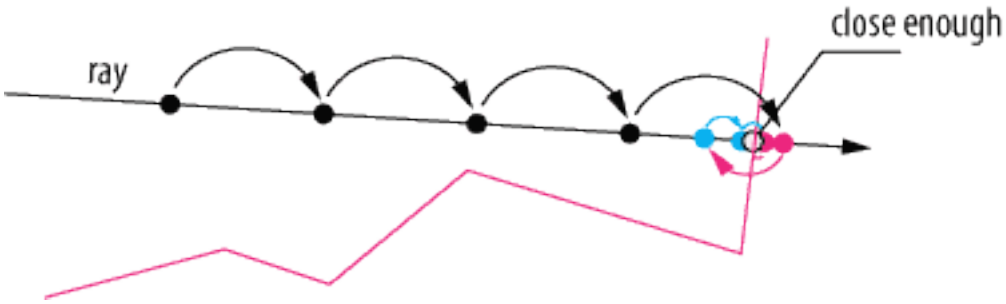


Figure 2.2: Figure visualizing ray marching, taken from `http://computergraphics.stackexchange.com/a/163`

Chapter 3

Aim and Objectives

The aim of this thesis was to understand how each of the components worked and how their parameters affected visuals and performance. The components are responsible for how the scene is generated. Example of components are land and water bodies.

To allow for the measurements a C++/GLSL implementation had to be built from the ground up. Were the C++ implementation was entirely written by me and the GLSL code where taken from `www.shadertoy.com`. The reason for building the application is that there are several types of information that needed to be gathered. While also needing to be able to change and modify the source shader in realtime and observing the changes. The fragment shader[16] was chosen as the source shader as the time of the project was limited. The shader only required minimal modification in order to work with the implemented application.

3.1 Research Questions

RQ1: What are the important parameters in each component that affect the visual result?

RQ2: How does changing different parameters of each component in the ray-marching fragment shader[16] affect the realtime performance?

3.2 Implementation Goals

The features needed were a simple text editor that could modify the shader code and apply the changes in realtime. Information such as frame rate and GPU timings was displayed per frame. Controls for changing parameters in the fragment shader was implemented to allow for easier testing.

Time was one of the default implemented parameters in the fragment shader and a special control to modify this parameter was implemented. Additional pa-

rameters could be controlled by adding a few lines of code, the parameters for the components where added using a simple parser in the C++ implementation.

Chapter 4

The Shader

This chapter will explain the important parts of the fragment shader. The components and ray marching algorithm. The fragment shader used in this project have been taken directly from `www.shadertoy.com` [16].

4.1 Shader Overview

The shader generates procedural terrain using noise functions to generate pseudo random values. These random generated values are dependent on the position in the world where the ray is going. This means that every part of the terrain is known but only visible as the camera gets close to it.

4.2 The Shader Code

This section will cover how the shaders works and what exactly happens in it. The vertex shader is a simple pass through shader enabling the fragment processing. The fragment shader begins execution and will calculate the camera position and direction for each fragment. Once the position and direction is calculated the scene will be generated using this information. Performing ray marching on the camera position and using the direction, stepping is performed. When performing stepping the ray marching algorithm will reconstruct a point using an offset and then calculate the height of the terrain. If the terrain is within a threshold of the reconstructed point, the functions returns with a hit, and the distance to the terrain. In order to gain a bit more detail out of the terrain the algorithm performs a subdivision step on the distance. This is done to increase the quality of the terrain. If the function never entered the terrain threshold or went to far away from the camera origin the returned result is a miss. The hit or miss is then used to render either the terrain or the cloudy sky above it.

The fragment shader can either render the terrain or the sky but in order to do so it needs to know how the terrain looks. The terrain is calculated in the ray marching loop and uses a variation of parameters to change how it looks. The

parameters ranges from height to how rough the mountains and the plain fields should be and how much trees should be generated. The terrain is generated using an iteration of noise levels that adds on top of each other. Each iteration the noise is different so that the result looks more natural.

Once a hit or miss have been determined it is time to output the final color. The color for the sky is a mix between blue and light yellow depending on the camera direction towards the sun. The terrain is a bit more complicated and uses a second iteration phase on the terrain. This phase is executed to gather more data about the terrain intersection point. The data gathered is used to calculate if the point is in a slope or if it is flat on the ground. The height of the terrain is used to calculate beaches and snow on mountain tops. Figure 4.1 shows how the shader execution is performed.

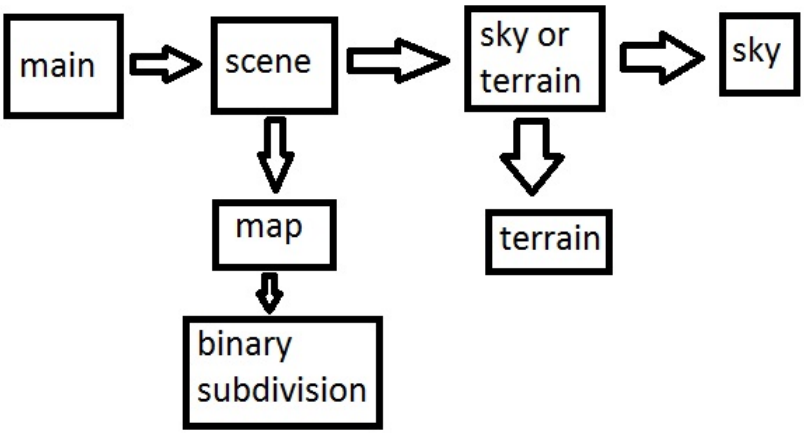


Figure 4.1: Shader flow chart created using the shader source.

4.2.1 Main

The main function of the program is called *mainImage*. In this part the camera is created using a position and direction. Functions that are called from here are, the *Terrain* function, the *CameraPath*, The *Scene* function, the *Terrain*2 and the *TerrainColour*. It could also call the *GetSky* function and *GetCould* function if the ray missed. Once this has been completed some post processing color correction is performed. This is the setup function for the steps to be performed next. The camera path is computed and used to move the camera around the landscape so that all the features can be shown respectively.

4.2.2 Scene

The scene function is what makes the ray marching and this is where it performs the point reconstruction. When the point is constructed it gets sent into the Map function. The programs ray marching loop performs a maximum of 150 iterations. The last two intersection points from the Map function are saved until the loop have finished. Once it is done a call is made to the $BinarySubdivision$ function. When this function returns it will tell if it hit or if it missed. This helps determine if the sky is to be rendered or if it is the terrain that should be rendered. The scene function is the one where most of the performance is spent as it contains a large loop.

4.2.3 Map

Map function calls the $Terrain$ function and then creates the tree lines. The tree line is what decides if a tree should be present or not on the terrain. The tree line value generated is used in a tree function that generates a cluster of trees. This gives a forest like look to the trees on the terrain.

4.2.4 BinarySubdivision

$BinarySubdivision$ function takes a vec2 and that contains two distance values. The Figure 4.2 shows how it takes two points and calculates a new point in between. The new point is checked against the terrain, the algorithm then uses this point with the old point that was closest to the terrain. This helps preventing small tips and details from disappearing due to the stepping. The more iterations you use the more detail is preserved but performance will go down. This is something that needs balancing between performance and quality.

4.2.5 Terrain

$TerrainColour$ function takes the position, normal and height of the terrain and calculates the final color. In this stage water is also added if the terrain is below the height 0. A beach effect is also added depending on the slope of the terrain. The normal data can be used to determine the angle of the terrain position. This helps to generate rocky surfaces or flat green fields. Using the height we can check if we are high enough to apply a snow like effect to the mountains that have formed. Finally we call a function called $ApplyFog$, it mixes the sky color with the terrain color to blend it for a smooth transition on the distant objects. This gives a better feel to the scene, but also helps reduce visual artifacts on distant mountains that are not visible due to the number of steps the ray marching algorithm does. Once all the colors are set the final step is a gamma correction stage that adjusts colors, makes the terrain look more crisp and correctly colored.

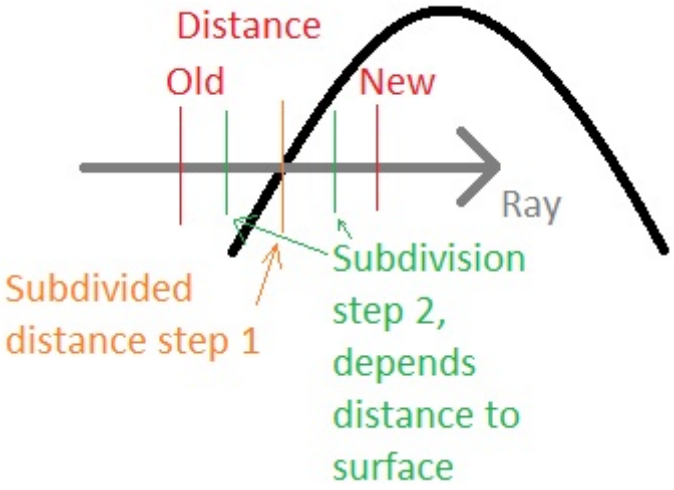


Figure 4.2: Binary subdivision.

4.2.6 Sky

As mentioned depending on the return value of the *Scene* function the sky or terrain is to be rendered. The *GetSky* function returns a gradient from light blue to light yellow depending on the camera direction towards the sun to give a simple sun flare effect. This is immediately followed by the *GetClouds* function. This one performs a fractal noise and mixes white color with the sky color to give a cloud effect.

Chapter 5

Method

This chapter will cover about the method used to implement and perform test on the ray marching shader. The first part will cover the different components in the scene and how the parameters where defined and implemented. The second part of the chapter will cover the experiment bit and how that was performed.

5.1 Ray Marching

Ray marching is a algorithm that renders an image by sending out rays from a given point in space (camera) with a direction (view direction). The ray will then traverse a distance and it will check if the height of a given distance is larger than the height of the ray. If this is the case then the terrain has been hit, otherwise it have missed and another step is taken increasing the traversed distance. This process is repeated until it have either collided with the terrain or performed a set number of iterations [5]. This can easily be achieved by using the fragment shader in realtime.

5.2 Terrain Components

The scene can be seen as a tree of different components that each have there own set of parameters. The tree can be visualized something like this, where land and water are part of the terrain, and the sky color and clouds are part of the sky.

- Terrain
  - Land
  - Forests & Trees
  - Water
- Sky
  - Sky Color
  - Clouds

These are the components in the scene and they have there own set of parameters. The land component will have parameters that will affect the height, offset and scaling frequency. It also have parameters for some more fine detail generation. The forest component have parameters that will affect the height of the trees and how the trees are positioned on the terrain such as clustering and density parameters. Water component have parameters to affect the water level height defining how much water the scene is having, and controls that affect how the waves look including offsets and turbulence. And finally the clouds component have parameters that affect the density of the clouds, a height offset that moves them higher up or down in the sky.

5.3 Defining The Parameters

There are where a number of ways to define the parameters, we could have found them in the file and modify them each separately and noting the default values for each parameter, or we could add a list of variables to the beginning of the shader shader file and use these for modifying the parameters. With the latter approach it would be much easier to see all the parameters. This method would also allow to be able to define more parameters if that was needed.

The following code listing shows how the parameters where defined in the shader code. Each parameter needed a variable name that would be referenced in the code, it would also need a default value that it was set to at the beginning of execution. And after that the min and maximum value for the parameter. The minimum and maximum range where added to limit the amount of values available. These values where chosen so that if the value where set larger than the value it would no longer affect the visuals or it would be hard to notice. Example of such parameters are the forest density and cloud density. Some parameters such as terrain height and forest height could have almost an infinite range so they where limited where changing them to the maximum or minimum value would start to show large visual artifacts.

```
//parameters
//    name, default, min, max

// terrain parameters
#param float terrainHeight = 66.0, -150.0, 150.0
#param float terrainOffset = 0.05, -0.15, 0.15
#param float terrainScaleFreq = 0.251, -0.5, 0.5
#param float terrainParam3 = 0.751, -1.0, 1.0
#param float terrainParam4 = 0.15, -1.0, 1.0
#param float terrainDetails = 0.4, -1.0, 1.0

// forest parameters
#param float forestHeight = 1.0, -100.0, 100.0
#param float forestDensity = 0.0, -5.0, 10.0
#param float forestParam1 = 0.3, -4.0, 5.0
```

```
#param float forestClusterFreq = 3.3, -5.0, 5.0
#param float forestRegionSize = 0.5, -5.0, 5.0

// water parameters
#param float waterLevel = 0.0, -50.0, 50.0
#param float waterParam1 = 4.5, -10.0, 10.0
#param float waterParam2 = 4.7, 0.0, 50.0
#param float waterParam3 = 1.3, 0.0, 50.0
#param float waterParam4 = 4.69, 0.0, 50.0
#param float waterParam5 = 35.0, 0.0, 50.0

// cloud parameters
#param float cloudDensity = 0.0, -5.0, 5.0
#param float cloudHeight = 0.0, -100.0, 100.0
#param float cloudScaling = 0.01, 0.0, 1.0
#param float cloudParam2 = 0.55, 0.0, 5.0
#param float cloudThickness = 5.0, 0.0, 10.0
```

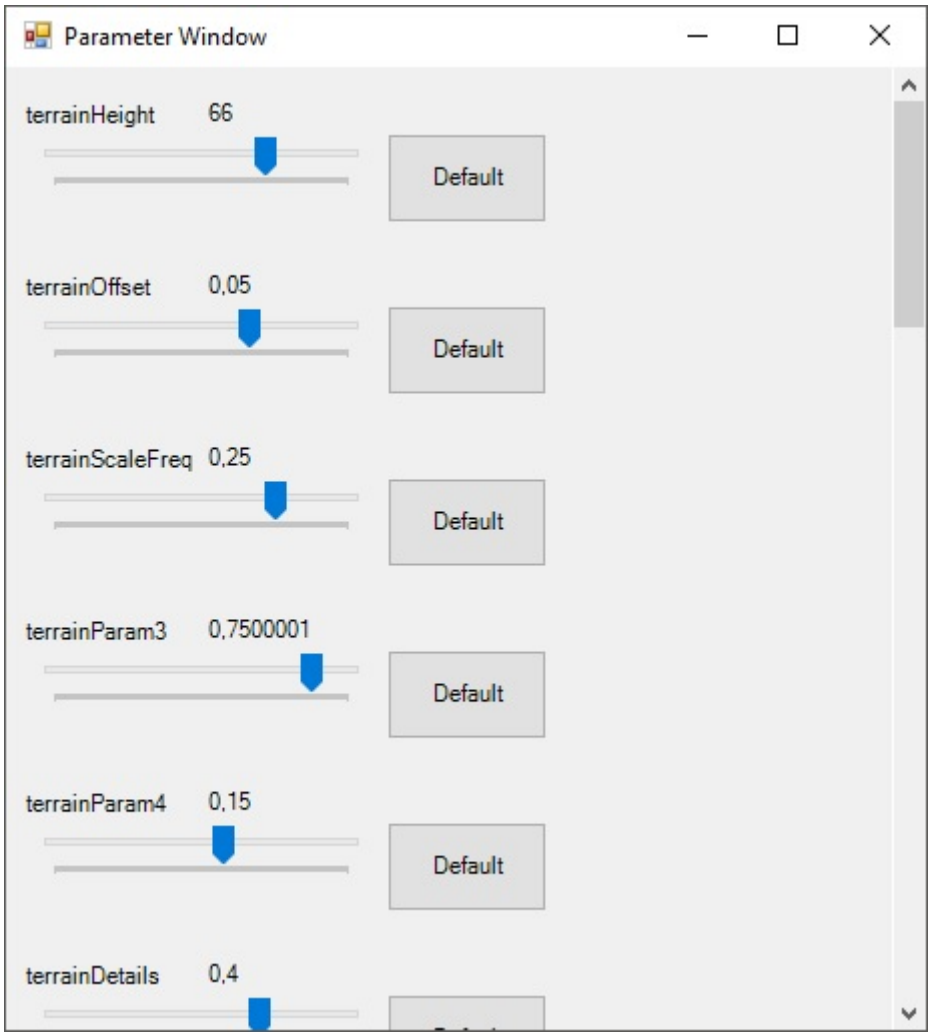


Figure 5.1: The parameter window for real time editing of the parameters.

A parameter is defined using the #param keyword and is being parsed by the C++ implementation. The parameter is then converted to a uniform that will have the default value of what is specified. The min and max value exist in order to add a range where the user can specify their input. The parameter was then stored as a user interface element so that the user could modify the parameter value in real time. The Figure 5.1 shows how the parameter window looks like.

5.4 Experiment

Once the implementation and defining parameters where done an experiment was performed. In the experiment I changed the values of the parameters. To start off I changed the parameters by hand to some random values to see which of the parameters would affect the performance. Later when more data was needed for generating performance graphs an built in test where used to iterate through all the parameters and change them individually. It started with the first parameter and once it was done the parameter was reset to it's default value. Then the next parameter was changed. The parameter was changed 25 times and the chosen parameter value was interpolated between the min and max range for the specific parameter. The frame was then let to stabilize for two seconds and then the result was saved to a file. Each parameter was saved separately to make it easier to generate graphs and look at the data. The result saved was the parameter value and the FPS that the value gave. The graphs shows a precision of $\pm$ 1 FPS.

Chapter 6

Results

This chapter will list all of the results from performing changes to the parameters. In order to be able to compare the visuals and performance a reference were needed to compare values against. The parameters would then operate on the same frame. The Figure 6.1 shows how the scene looks without any changes. The tables in this chapter will show the hand picked values used to see how a specific parameter where affecting the visuals and the performance. The graphs will show how the entire range of values for a specific parameter would affect the performance.

Figure 6.1: The scene without any changes to the parameters.

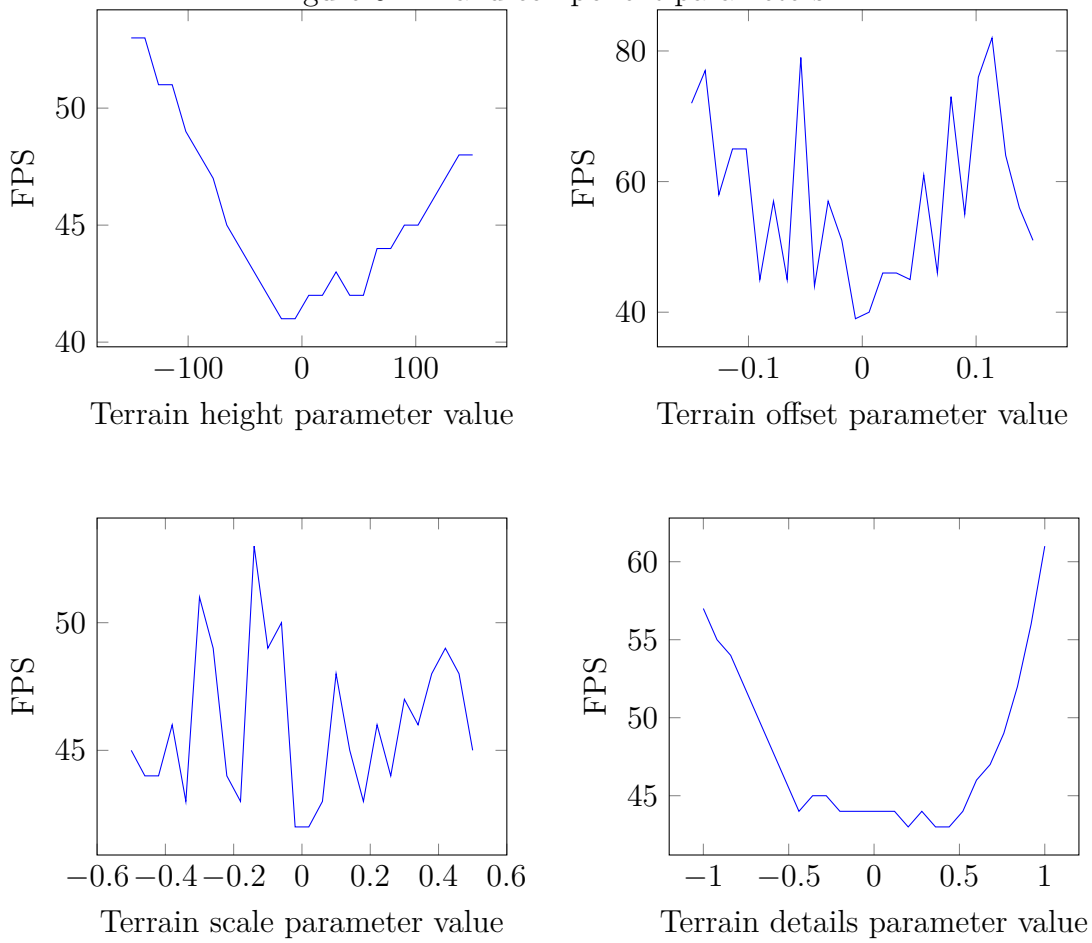
6.1 Land Parameters

When changing the parameters for the land component, height and details affected the performance the most. The offset and scale frequency changed the look very much and this could cause this parameter to have a high performance increase for a specific value. The Table 6.1 lists the parameters and some of their hand picked values to see if the parameters affected performance in any big manner. The Graph Figure 6.2 shows how the performance changed when changing the parameters.

Table 6.1: Land Parameters

Test Number	Terrain height	Terrain offset	Terrain scalefrequency	Terrain param3	Terrain param4	Terrain details	FPS
0	66.0	0.05	0.25	0.75	0.15	0.4	43 fps
1	150.0	0.05	0.25	0.75	0.15	0.4	49 fps
2	66.0	0.0	0.25	0.75	0.15	0.4	42 fps
3	66.0	0.05	0.5	0.75	0.15	0.4	46 fps
4	66.0	0.05	0.25	1.0	0.15	0.4	46 fps
5	66.0	0.05	0.25	0.75	0.1	0.4	43 fps
6	66.0	0.05	0.25	0.75	0.15	0.3	43 fps
7	50.0	0.045	0.29	0.6	0.235	0.42	45 fps
8	-45.0	-0.05	-0.109	0.643	0.328	0.35	52 fps
9	66.0	0.05	0.25	0.75	0.4	0.4	49 fps
10	66.0	0.05	0.25	0.75	0.15	1.0	60 fps
11	14.384	0.05	0.25	0.75	0.15	0.4	42 fps
12	66.0	-0.045	0.25	0.75	0.15	0.4	46 fps
13	66.0	-0.045	-0.5	0.75	0.15	0.4	50 fps
14	66.0	0.05	0.362	0.75	0.15	0.4	46 fps
15	66.0	0.05	0.362	-1.0	0.15	0.4	44 fps

Figure 6.2: Land component parameters



The first graph shows how the height with different values goes from a high fps down to low and back up again. This is because the higher the terrain height value is the more bumpy and rough it will look. This then causes the ray marching loop to exit earlier than normal. When the parameter value is low the terrain is rather flat and it takes a bit more time to complete the loop.

The second graph shows how the offset greatly affect the visual results and performance. This is happening because one small change to the offset will cause a frequent change in the visual. This is why the FPS goes up and down a lot in this graph.

The third graph shows how the scaling affects the performance and this parameter and this parameter acts in a similar manner to the offset parameter. Therefore a small change to this parameter can cause a rather big change in the visual and cause a rapid change in performance.

The last graph shows how the details parameter affects the performance. Where a value around 0 would give lower fps than a value around 1. The FPS is again dependent on how early the shader can finish processing a fragment. When the details parameter value becomes larger than 0.5 or less than -0.5 the terrain will start to gain artifacts and not look like real terrain any more.

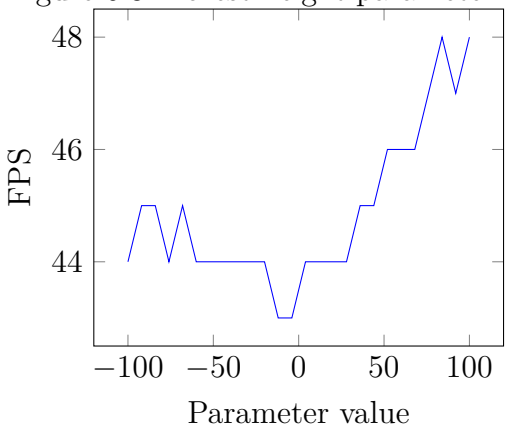
6.2 Forest Parameters

Changing the forest parameters, height where the one affecting performance the most, clustering and density only affected the visual effect the tree would have. The Table 6.2 lists the parameters and their values. The Graph Figure 6.3 shows how the tree height would affect the performance going from the minimum value to the maximum value.

Table 6.2: Forest Parameters

Test Number	Forest height	Forest density	Forest param1	Forest clusterfrequency	Forest regionsize	FPS
0	1.0	0.0	0.3	3.3	0.5	43 fps
1	100.0	0.0	0.3	3.3	0.5	49 fps
2	-100.0	0.0	0.3	3.3	0.5	46 fps
3	1.0	5.377	0.3	3.3	0.5	43 fps
4	1.0	0.0	-2.15	3.3	0.5	43 fps
5	1.0	0.0	0.3	0.0	0.5	43 fps
6	1.0	0.0	0.3	3.3	4.	43 fps
7	1.863	0.0034	-0.856	1.644	0.411	43 fps
8	1.0	-0.685	0.105	3.3	0.5	43 fps
9	1.324	1.005	0.35	1.984	0.5	43 fps

Figure 6.3: Forest height parameter



The graph shows how the forest height affects the performance, in this case a value below 0 will make holes in the ground and not look good. A higher value will also start to cause visual artifacts.

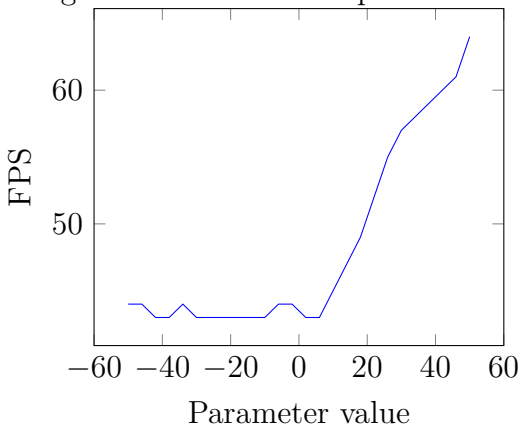
6.3 Water Parameters

When changing the water parameters a water body was needed therefore the time variable was changed to 53.0 to show an open water patch. The Figure 6.5 shows how this scene looks like. The parameters with most impact on the performance was the water level. This increased the performance as the water level got higher. Other parameters for the water affected scaling and offsets and only changed the visual look of the water. Table 6.3 lists the parameters and values for the water component. The Graph Figure 6.4 shows how the water affected performance during different heights.

Table 6.3: Water Parameters

Test Number	Water level	Water param1	Water param2	Water param3	Water param4	Water param5	FPS
0	0.0	4.5	4.7	1.3	4.69	35.0	43 fps
1	50.0	4.5	4.7	1.3	4.69	35.0	68 fps
2	-50.0	4.5	4.7	1.3	4.69	35.0	43 fps
3	0.0	-10.0	4.7	1.3	4.69	35.0	43 fps
4	0.0	4.5	28.424	1.3	4.69	35.0	43 fps
5	0.0	4.5	4.7	24.65	4.69	35.0	43 fps
6	0.0	4.5	4.7	1.3	38.67	35.0	43 fps
7	0.0	4.5	4.7	1.3	4.69	7.19	43 fps

Figure 6.4: Water level parameter



The graph shows how the water level affects the performance. The water level also affects how high in the air the camera is positioned. The height of the camera then offsets the scene and the higher up the camera is positioned the less of the scene we can see. This causes the raymarching loop to take large steps in each iteration and therefore exit much earlier than normal, therefore we gain performance.

Figure 6.5: The scene without any changes to the parameters, time set to 53.0.

6.4 Cloud Parameters

When changing the parameters for the cloud component the performance did not get affected, The parameters for clouds affected density, height, scaling and thickness. The parameters only affected the visual result. The Table 6.4 shows the parameters and their values. Because there where no performance impact there is no graph for the cloud component.

Table 6.4: Cloud Parameters

Test Number	Cloud density	Cloud height	Cloud scaling	Cloud param2	Cloud thickness	FPS
0	0.0	0.0	0.01	0.55	5.0	43 fps
1	5.0	0.0	0.01	0.55	5.0	43 fps
2	0.0	-100.0	0.01	0.55	5.0	43 fps
3	0.0	0.0	0.29	0.55	5.0	43 fps
4	0.0	0.0	0.01	0.171	5.0	43 fps
5	0.0	0.0	0.01	0.55	10.0	43 fps
6	1.575	-49.315	0.021	0.788	2.877	43 fps

6.5 Analysis and Discussion

The result shows that some of the parameters affect the performance more than others. When we take the terrain parameters in Table 6.1. We can see that the amplitude is the one that affects the performance the most. In this case changing it from 66.0 up to 150.0 would make the terrain more spiked. And this increased the performance. The other parameters affected the performance slightly. And the last two parameters, listed as $Terrainparam4$ and $Terraindetails$ when changed to a minimum amount would not affect the performance at all, however if they were changed by a large value shown in the Test 9 and 10, the performance increased. The $Terrainparam4$ would affect the terrain such as the mountains become taller and the valleys became deeper. The parameter had a large range but when reaching the limit the terrain would not look real any more. The $Terraindetails$ is what controls most of the details generated in the terrain, this means that decreasing this value will make the terrain loose a lot of the details and look pretty flat. Increasing it will make the details pop more but if the increase is to large the terrain will become a mess and most of the rendering will be artifacts. The Test 8 and 9 shows a different set of parameters that will make a different terrain. Figure 6.6 shows the different terrains generated.

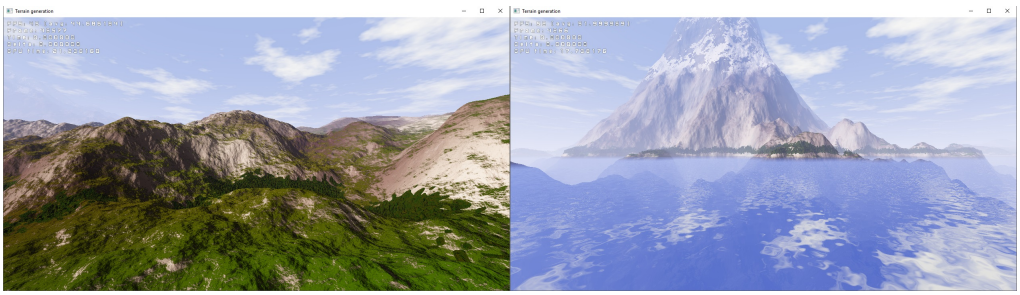


Figure 6.6: Test 8 and Test 9, showing different terrain. Test 8 is to the left and Test 9 is to the right.

The Table 6.2 shows that not much in the tree component can affect the performance of the rendering. The forest height is being the dominant one when it is increased, a slight increase in performance is happening when the height is decreased. Most of the other parameters affected features as density or clustering behavior. With the height parameter maxed out, there are however noticeable artifacts generated where the trees are stretching up in the sky. These artifacts are occurring due to how ray marching is done. The only way to remove them is by increasing the number of steps being performed and reducing the step distance. This change would the most likely affect the performance in a bad way. The different visual results is shown in Figure 6.7.

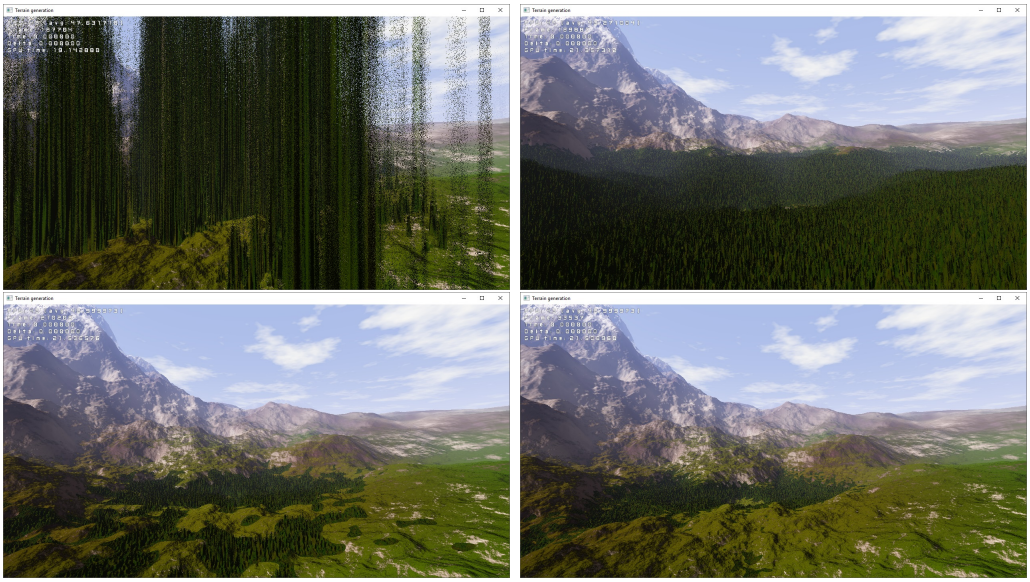


Figure 6.7: Shows forest test 1, 3, 7 and 8. And their parameter effects.

Looking at the Table 6.3 we can see that the only parameter that affects the performance is the water level. With an increased value we gain an increased frame rate. Decreasing the water level did not affect the performance at all, even tho it removed the water patch we where seeing. The other parameters simply change how the water looks and acts as offsets or scaling to the water details. The Figure 6.8 shows the water level parameter in effect, and Figure 6.9 shows how the surface noise can be changed using the parameters.

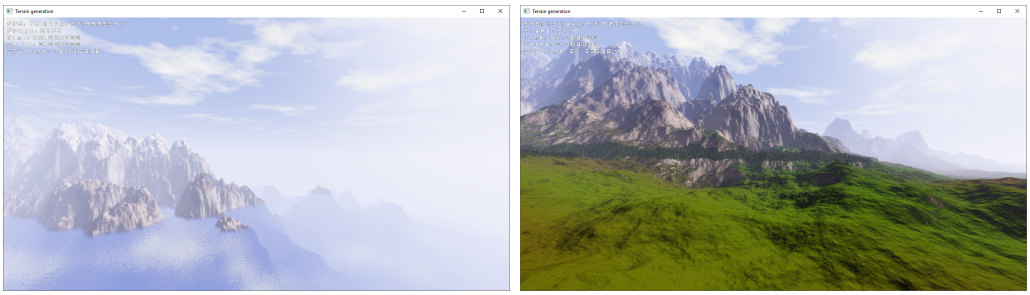


Figure 6.8: Water test 1 to the left and water test 2 to the right. Showing the different water levels.

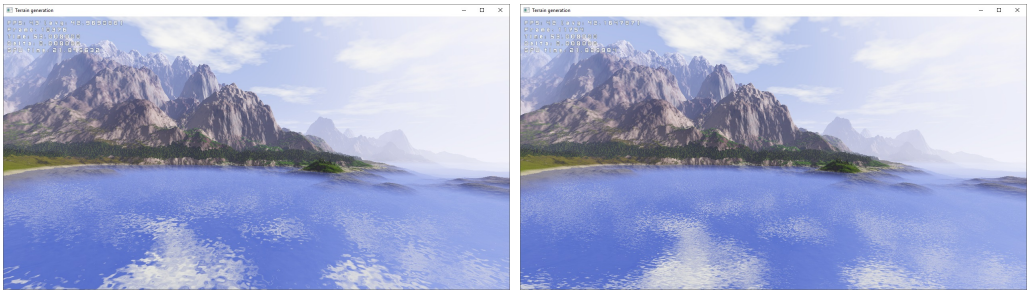


Figure 6.9: Water test 3 to the left and water test 4 to the right. The parameters affect the offset and scaling of the noise.

The Table 6.4 lists the changes done to the clouds, none of the parameters affected the performance. Height and density did not change anything other than moving the clouds higher up or down and increasing or decreasing how much of the screen where covered in clouds. The figure 6.10 shows the density and height parameter.

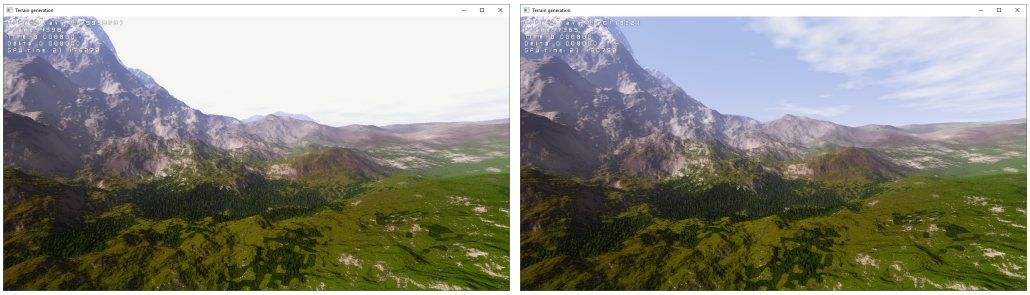


Figure 6.10: Cloud test 1(left) and 2(right),

Doing changes to all parameters can change the landscape completely. The figure 6.11 shows another two different scenes created using different parameters.

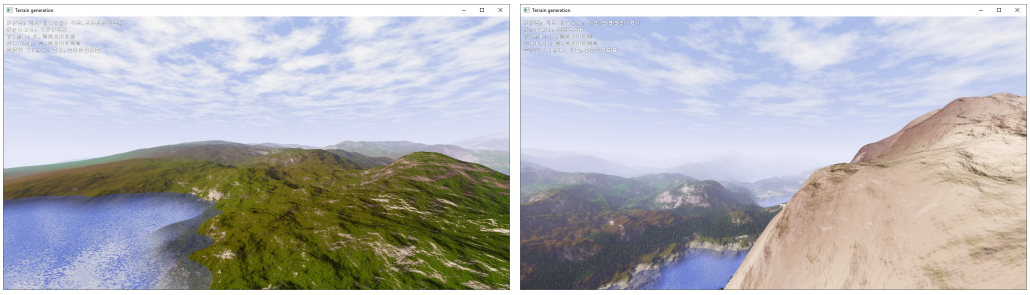


Figure 6.11: Showing a small hilly area to the right and a big mountain cliff to the right.

6.6 The Performance

At first it may be confusing that increasing the water level or the forest height could affect the performance. But when you look at how the code works it is actually not that confusing. This happens because of the way the ray marching iteration loop is written. This is how it looks like. In this case we use the unchanged version of the code.

```
//-------------------------------------------------------------------------------------
bool Scene(in vec3 rO, in vec3 rD, out float resT, in vec2 fragCoord )
{
 float t = 1.2 + Hash12(fragCoord.xy);
 float oldT = 0.0;
 float delta = 0.0;
 bool fin = false;
 bool res = false;
 vec2 distances;
 for( int j=0; j< raymarchIterCount ; j++ )
 {
  if (fin || t > 240.0) break;
  vec3 p = rO + t*rD;
  //if (t > 240.0 || p.y > 195.0) break;
  float h = Map(p); // ...Get this positions height mapping.
  // Are we inside, and close enough to fudge a hit?...
  if( h < 0.5)
  {
   fin = true;
   distances = vec2(t, oldT);
   break;
  }
  // Delta ray advance - a fudge between the height returned
  // and the distance already travelled.
  // It's a really fiddly compromise between speed and accuracy
  // Too large a step and the tops of ridges get missed.
  delta = max(0.01, 0.3*h) + (t*0.0065);
  oldT = t;
  t += delta;
 }
 if (fin) resT = BinarySubdivision(rO, rD, distances);
 return fin;
}
```

At the end of the iteration the distance variable is increased by a delta. The delta is calculated with this formula $delta = max(0.01, 0.3 * h) + (t * 0.0065)$; h is the height of the surface from the point. This means that the larger the distance to the point we are checking against the larger the next step will be. This is a performance optimization done in the code to speed things up. The increase comes when the camera is either very high up in the air. This means that the sky and cloud fragments will traverse further distances by average and completing faster. Completing faster is when the distance from the camera is far enough away that we consider it a miss. The threshold is 240.0 units away. The iteration loop will also finish if there is a hit against the terrain. Making more bumpy terrain faster to render as it is covering more of the scene close up. Flat terrain will cause most of the performance in the middle to perform a lot of iterations, resulting in a less

increase in performance and sometimes even a decrease in performance.

To answer the first research question, RQ1: *What are the important parameters in each component that affect the visual result?* The land component most important parameters that affected the visual result are the height and details. Offsets and scaling frequency are also important since they can change the look only with a very small change to the parameter. The important parameters for the forest component are height and density affecting the visuals the most. The parameters for the water component that affected the visual result the most where water level. The other parameters did change visuals but they where not easily noticeable. The important parameters for the cloud component are the density.

The second research question, RQ2: *How does changing different parameters of each component in the raymarching fragment shader[16] affect the realtime performance?* Some of the parameters affect the performance and some parameters don't affect the performance. The parameters that affect the performance also heavily affects how the final scene would look. Heavily in this case is noticeable difference in how the terrain looks when it comes to shape. If the difference in shape between two frames are very small the performance will be similar. if the visuals are very different between two frames the performance could be different. The reason that it is not a guarantee that the performance is different is because of how the shader is executed. We mentioned earlier that the performance is dependent on how early the shader can stop executing a given fragment on the screen. This also means that parameters that did not change the visuals in any major way would not affect the parameters, such as cloud density and water offsets and scaling.

Chapter 7
Conclusions and Future Work

This chapter will cover the conclusion and future work in the scope of procedural terrain generation using the fragment shader.

7.1 Conclusion

The conclusion is that the performance is depending on how fast the ray marching algorithm can finish. The algorithm can finish in two possible ways, either by letting the ray travel far enough to indicate a miss or by letting it hit the terrain and cause the loop to terminate. There is really no best way to terminate the loop, for example when you are close to the camera it will be best to exit with a terrain hit. But as the distance increases it will be best if the ray have reached its threshold limit and cause a miss, thus rendering the sky with clouds. Being close to the center of the screen where the horizon is placed there will be a hard time detecting a terrain hit vs a terrain miss, essentially putting more time into these pixels.

7.2 Future Work

Future work in the field of procedural terrain generation using the fragment shader will be to find out if the algorithm may be optimized more. It would be possible to reduce iterations by using a level of detail for example. This can be done at least for the coloring phase that requires more iterations on the terrain function. By having a limitation distance it could be greatly reduces and maybe give a slight increase in performance.

To find if there a possibility to reduce the total number of iterations in the ray marching stage is also something that needs to be experimented with further. Ways to achieve this could be by using early distance calculations or by using reduced quality of the terrain.

References

[1] K. Perlin, "An Image Synthesizer," in *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '85, (New York, NY, USA), pp. 287–296, ACM, 1985.

[2] L. Kang, J. Xu, C. Yang, B. Yang, and L. Wu, "An efficient simplification and real-time rendering algorithm for large-scale terrain," *International Journal of Computer Applications in Technology*, vol. 38, no. 1/2/3, p. 106, 2010.

[3] E. Gobbetti, F. Marton, P. Cignoni, M. Di Benedetto, and F. Ganovelli, "C-BDAM – Compressed Batched Dynamic Adaptive Meshes for Terrain Rendering," *Computer Graphics Forum*, vol. 25, pp. 333–342, Sept. 2006.

[4] H. Qu, F. Qiu, N. Zhang, A. Kaufman, and M. Wan, "Ray tracing height fields," in *Computer Graphics International, 2003. Proceedings*, pp. 202–207, July 2003.

[5] Inigo Quilez. `http://www.iquilezles.org/www/articles/terrainmarching/terrainmarching.htm` (Accessed 2016-04-09).

[6] R. Geiss, "Generating Complex Procedural Terrains Using the GPU.," in *GPU Gems 3*, pp. 7–37, Addison-Wesley Professional, 2007.

[7] A. Santamaría-Ibirika, X. Cantero, M. Salazar, J. Devesa, I. Santos, S. Huerta, and P. G. Bringas, "Procedural approach to volumetric terrain generation," *The Visual Computer*, vol. 30, pp. 997–1007, Sept. 2014.

[8] J. K. Helsing and A. C. Elster, "Noise Modeler: An Interactive Editor and Library for Procedural Terrains via Continuous Generation and Compilation of GPU Shaders," in *Entertainment Computing - ICEC 2015* (K. Chorianopoulos, M. Divitini, J. Baalsrud Hauge, L. Jaccheri, and R. Malaka, eds.), vol. 9353, pp. 469–474, Cham: Springer International Publishing, 2015.

[9] D. Grelsson, *Tile Based Procedural Terrain Generation in Real-Time : A Study in Performance*. 2014.

[10] D. S. Ebert, ed., *Texturing & modeling: a procedural approach*. Amsterdam ; Boston: Academic Press, 3rd ed ed., 2003.

[11] J. Doran and I. Parberry, "Controlled Procedural Terrain Generation Using Software Agents," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, pp. 111–119, June 2010.

[12] H. Hnaidi, E. Guérin, S. Akkouche, A. Peytavie, and E. Galin, "Feature based terrain generation using diffusion equation," *Computer Graphics Forum*, vol. 29, pp. 2179–2186, Sept. 2010.

[13] J.-D. Génevaux, r. Galin, E. Guérin, A. Peytavie, and B. Beneš, "Terrain generation using procedural models based on hydrology," *ACM Transactions on Graphics*, vol. 32, p. 1, July 2013.

[14] S. T. Teoh, "RiverLand: An Efficient Procedural Modeling System for Creating Realistic-Looking Terrains," in *Advances in Visual Computing* (G. Bebis, R. Boyle, B. Parvin, D. Koracin, Y. Kuno, J. Wang, J.-X. Wang, J. Wang, R. Pajarola, P. Lindstrom, A. Hinkenjann, M. L. Encarnação, C. T. Silva, and D. Coming, eds.), no. 5875 in Lecture Notes in Computer Science, pp. 468–479, Springer Berlin Heidelberg, Nov. 2009. DOI: 10.1007/978-3-642-10331-5_44.

[15] J. C. Hart, "Perlin Noise Pixel Shaders," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, HWWS '01, (New York, NY, USA), pp. 87–94, ACM, 2001.

[16] Shadertoy. `https://www.shadertoy.com/view/4slGD4` (Accessed 2016-04-09).

[17] . `https://creativecommons.org/licenses/by-nc-sa/3.0/deed.en_US` (Accessed 2016-11-09).