Graz University of Technology
Institute for Computer Graphics and Vision

Master Thesis

---

# Parallel River Networks

---

**Andreas Kurz**
Graz, Austria, August 2020

*Advisor*
**Ass.Prof. Dipl.-Ing. Dr.techn. Markus Steinberger**
Institute for Computer Graphics and Vision, Graz University of Technology

To my parents

All things are difficult before they are easy.

Thomas Fuller

## Statutory Declaration

*I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used.*

| **Graz, Austria** | **September 01, 2020** | |
|---|---|---|
| Place | Date | Signature |

## Eidesstattliche Erklärung

*Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.*

| **Graz, Österreich** | **01. September, 2020** | |
|---|---|---|
| Ort | Datum | Unterschrift |

# Abstract

Computer generated terrain data takes on increasingly more importance in simulation and entertainment related applications. Most procedural terrain generation methods do not include mechanisms to create river networks. This thesis investigates if there is an efficient way to add river networks to already generated terrain data. For this purpose we start with six quadtrees arranged like a cube. These quadtrees, once projected on the unit sphere, represent a planet. By combining these quadtrees with heightmaps, we arrive at a rather well known version to represent terrain on planetary scales, which allow for adaptive level of detail by further subdividing the quadtrees based on camera position. The terrain heightmaps used in such applications can be either precomputed or generated on-the-fly by procedural generation algorithms. In this thesis, we present an approach to add river geometry to this simple heightmap based approach, by adding a second texture to the quadtree patches. These textures contain all the necessary data to modify the existing terrain based on the representation of our river network. We use a simple spline based representation for our rivers, but any representation can be utilized as long as sample points can be generated from it. These sample points consist only of position, first derivative, river height and river width at each sample point. By taking advantage of modern graphics hardware, and the NVIDIA CUDA Toolkit, each texture of river data can be computed independently of any other patch texture, and can thus be easily parallelized, because all our patches have the same geometry of a regular mesh. In this way it is also possible to increase the level of detail further after the input data has been exhausted by adding procedural noise on top of the input data, with increasing subdivisions in the quadtrees.

**Keywords.** procedural generation, river networks, planet generation, parallel, GPU, splines, fractals

# Kurzfassung

Computergenerierte Geländedaten gewinnen immer mehr an Bedeutung in Simulations- und Unterhaltungsanwendungen. Die meisten prozeduralen Methoden Geländedaten zu generieren beinhalten keine Mechanismen zur Erstellung von Flussnetzen. In dieser Arbeit wird untersucht, ob es einen effizienten Weg gibt, Flussnetze zu bereits generierten Geländedaten hinzuzufügen. Zu diesem Zweck beginnen wir mit sechs Quadtrees, die wie ein Würfel angeordnet sind. Diese Quadtrees, nachdem diese auf die Einheitskugel projiziert wurden, repräsentieren einen Planeten. Wenn man nun diese Quadtrees mit Höhenfeldern kombiniert, kommen wir zu einer bekannten Version zur Darstellung von Terrains in der Größe von Planeten. Diese Darstellung ermöglicht einen adaptiven Detailierungsgrad durch weitere Unterteilung der Quadtrees anhand der Kameraposition. Die Geländedaten in diesen Höhenfeldern können entweder vorberechnet oder spontan durch prozedurale Algorithmen generiert werden. In dieser Arbeit stellen wir einen Ansatz vor, wie man diesen höhenfelderbasierten Ansatz um Flussgeometrie erweitert, indem man eine zweite Textur zu den Quadtree Knoten hinzufügt. Diese Texturen enthalten alle erforderlichen Daten, um das vorhandene Terrain basierend auf der Repräsentation unseres Flussnetzes zu verändern. Wir repräsentieren unsere Flussnetze durch einfache Splines, aber jegliche beliebige Repräsentation von Flussnetzen ist möglich, solange sichergestellt wird, dass die nötigen Daten aus der gewählten Repräsentation generiert werden können. Diese Daten beihnalten die Position, die erste Ableitung, die Flusshöhe und die Flussbreite am jeweiligen Prüfpunkt. Durch die Nutzung moderner Grafikhardware und des NVIDIA CUDA Toolkits, kann jede Textur von Flussdaten unabhängig von jeder anderen Textur berechnet werden, und kann somit leicht parallelisiert werden, da alle unsere Knoten der Quadtrees die gleiche Geometrie besitzen. Auf diese Weise ist es auch möglich, den Detaillierungsgrad weiter zu erhöhen, nachdem die Eingabedaten erschöpft wurden, indem man bei weiteren Unterteilungen der Quadtrees prozedurales Rauschen hinzufügt.

# Contents

# List of Figures

# Part I

# Overview

# Chapter 1

# Introduction

The performance of graphics hardware is increasing with every single generation of graphics cards. Additionally, storage mediums are getting increasingly larger and faster. This increase in graphics performance and storage leads to more and more detailed computer generated imagery being presented to the user. These images, usually made up of many assets, have to be created with all their details either by a user or the computer in a predetermined way.

These assets can range from textures to very detailed three-dimensional models or whole universes to be explored. Unfortunately, the increase in performance and storage also means that the expected quality of assets increase alongside them. This leads to more and more time of the development process spent on creating larger and larger numbers of assets as the virtual realities grow in size, complexity and, most often than not, the expectation of realism by the user. This, coupled with the recent increase in indie game development, makes it necessary to decrease the overall time spent on creating all those assets to make the development of games with very small teams or an astronomical number of assets feasible.

Procedural generation techniques are the solution to that problem. They take a small number of inputs and can generate assets algorithmically. In this way it is possible to generate arbitrary outputs which can be used to create 3D models, textures and scores of other necessary assets. There are no limits to the types of assets which can be generated using procedural techniques.

In the beginning, procedural generation techniques were mainly used to create realistic-looking terrain and textures, but were soon used for many different applications. They can also be used in scenarios where storage is limited, because while it may not be possible to store every single necessary asset on the provided memory, it is usually quite easy to store the algorithm which can generate a version of that asset, or one which can approximate it well enough.

It is also possible to create entire universes procedurally by using a single input to the procedural generation algorithm. One of the most famous examples of games which took advantage of these techniques in recent years was *No Man's Sky*. It features a wholly procedurally generated universe with, allegedly, over 18 quintillion of differing planets, each containing its own unique ecosystem. If a game development studio wanted to create this entire universe by hand, their game would never be released.

As these procedural generation algorithms create their outputs based on a small number of inputs, different versions of these outputs can be generated by simply altering the inputs. If, for example, the output of your algorithm would be a procedurally generated universe, it would be possible to create a completely different universe by using a different input. This is used in games to increase re-playability because the whole world can be different in the next play-through. The inputs in procedural generation techniques are usually called *seeds*.

On other thing to note about procedural generation algorithms is the fact that they will always yield the same output when supplied with identical inputs. This way people can share the same world, level, or universe as other players by simply sharing the same inputs to the algorithm responsible for generating that asset.

One more important application of procedural generation techniques is in Level of Detail (LoD) schemes. Level of Detail refers to having multiple representations for a single asset, each being more detailed than the preceding one. An example of this would be that from far-away a tree could be just a single texture. The closer one would get to said tree, the more detailed versions of that tree would replace the previous, more coarse version, thus adding more and more detail to the tree. Creating such detailed assets would also lead to very large consumption of memory, thus it is far easier to simply create more detail for an asset algorithmically when it is necessary.

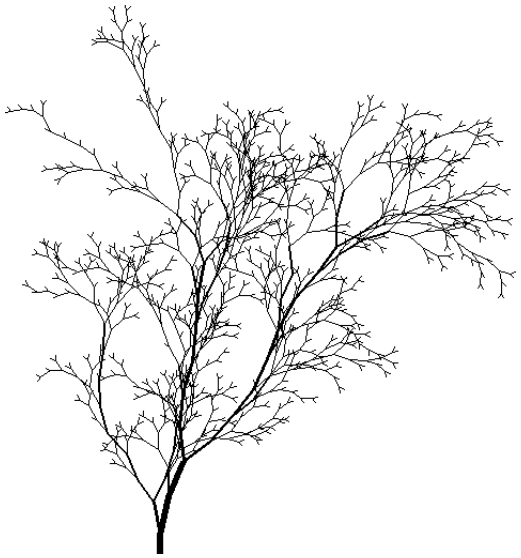
In general, procedural generation techniques can be applied to create assets where there are rules dictating the final appearance or behavior of that asset. In Figure 1.1, two examples of procedurally generated assets can be seen. Figure 1.1(a) shows a tree generated by using a Lindenmayer system or *L-system* [34], while Figure 1.1(b) shows a procedurally generated building. Figure 1.2 shows a procedurally generated city, which contains not only procedurally generated buildings, but also procedurally generated road networks. In general, if one is able to simplify their assets down to the core rules and behaviors, one can write a procedural algorithm which creates different versions of that asset algorithmically, without the need to create these by hand.

With these different applications of procedural algorithms, it is now easy to see that they can have vastly different scales they are operating on. One of these algorithms could only be responsible for a single texture, while another could create a complete universe. It is up to the individual developer team to choose and implement the appropriate version for their application.

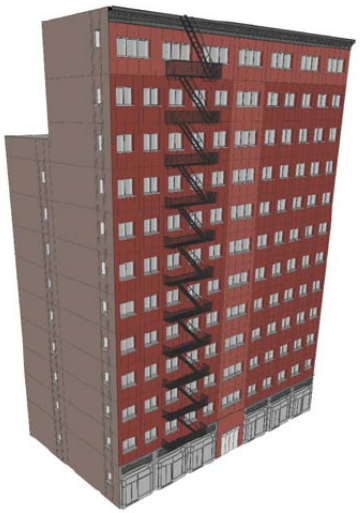
In this work, our focus is on generating terrain on a planetary scale, but in particular on the creation of river networks which span the continents of this planet. Because of the difficulty of creating realistic looking river networks, a lot of procedural terrain generation applications opt to not create river networks, but it has been an intensely studied subject over the last few decades.

The reason we are interested in rivers and river networks in terrain generation is, for one part, because many applications try to replicate our home-planet, where river networks are abundant. The video game and movie industry also mainly create worlds with river networks present, because of the importance of water for living organisms. This makes the presence of tools, which are able to help create these worlds easier, very important.

There are many considerations when choosing the appropriate process for generating the assets for each different application. Many procedural terrain generation techniques,



(a) A procedurally generated tree taken from [33].



(b) A procedurally generated building taken from [24].

**Figure 1.1:** Examples of procedurally generated geometry.

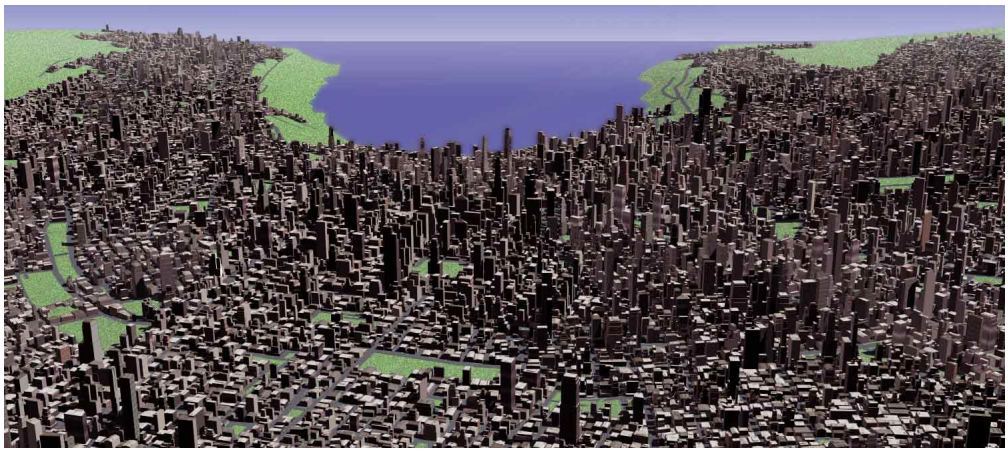


**Figure 1.2:** A procedurally generated city taken from [30].

for instance, place a lot of importance on being real-time. What this means is that a near infinite, or in our case spherical, terrain can be created near instantly and viewed in many different levels of detail. The more detailed geometry needs to replace the coarser geometry without the user noticing at the appropriate distance so no artifacts of this switch are present. To achieve this constraint of being real-time, it is usually very effective to take advantage of the graphics hardware. They are optimized to perform tasks in parallel. Thus it is necessary to create an approach for the application which can be easily parallelized.

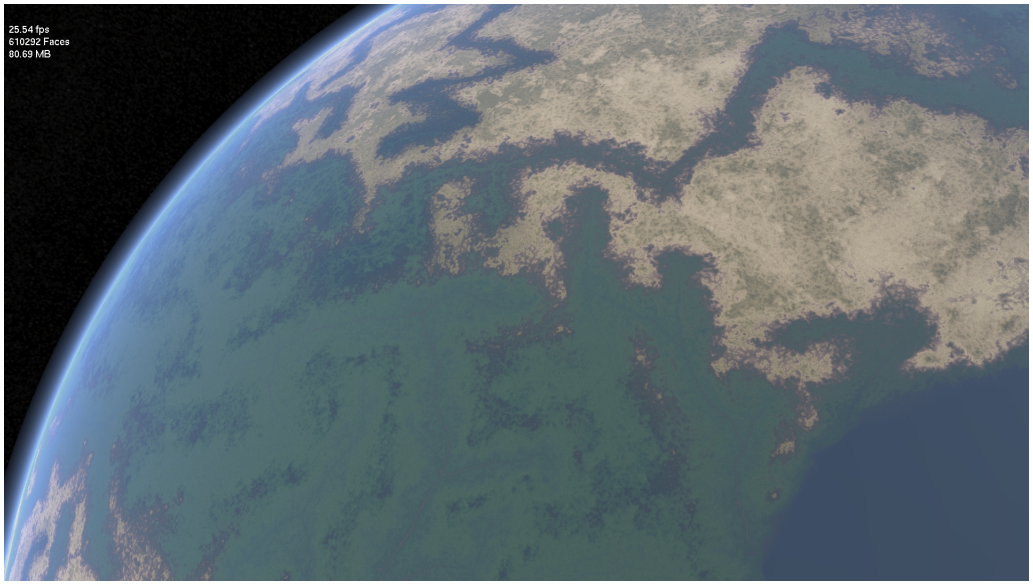



**Figure 1.3:** A procedurally generated planet with river networks taken from [11].

The more parts of an algorithm can be performed in parallel, the more one can expect to gain from switching to a parallel version of said algorithm.

The main inspiration for this work can be found in the work by Derzapf et al. [11]. In that work the authors create river networks on a planetary scale. They do this by representing the planet as a singular irregular mesh and adding and/or removing detail when necessary based on user input. They are also able to perform their approach in real-time by performing said operations in parallel. Additionally, They are able to achieve seamless transitions while approaching their planet with the camera, which is also part of what we wanted to achieve in our work. Their river networks are created along the edges of their irregular mesh, thus combining the data used for the river networks with their already existing planetary data structure. This helps to decrease the overall complexity of the task of creating the river networks. While the utilization of an irregular mesh helped in this respect, it made the adding and/or removing of detail much harder to parallelize. They also use more computational time per frame to determine whether detail has to be added or removed.

## 1.1 Contribution

The purpose of this thesis is to find out whether it would be possible to achieve similar results as in the work by Derzapf et al. [11] by using a regular mesh instead of an irregular one. The hypothesis is that a regular mesh may be more suited to parallelization on a GPU. What we present here is an approach which takes an initial base terrain as an input and deforms that terrain through the river network. The input to this deformation operation are also only sample points along the river paths. This makes it possible to have a wide variety of sources which can be used to generate the deformation. One would be

able to generate sample points from real world rivers, or create artificial rivers and then generate sample points from them. This makes the actual representation of the rivers used in the implementation irrelevant, only the ability to create sample points has to be assured to utilize the method. The consequences of using a regular mesh instead of an irregular mesh will be discussed later in this thesis. Additionally, the initial base terrain which will be deformed by our approach, can also be of any source, such as real scanned height data, created by hand or any number of procedural approaches, as long as one is able to generate heightmaps of the created terrain. This leads to our contribution being an algorithm which takes an arbitrary base terrain, as well as an arbitrary river representation and uses them to deform the base terrain through the sample points created by the river representation to create river networks.

## 1.2 Structure

In Chapter 2 we will talk about procedural generation and other publications which are relevant to this thesis. We will shortly discuss them and highlight their connection to this thesis. After the discussion on our related work, in Chapter 3, we will move on to briefly discuss all the techniques necessary for our method to work. In Chapters 4 and 5 we will talk about how our initial terrain and rivers are created respectively. The most important part of this thesis will be discussed in Chapter 6, which describes the runtime algorithm created during the course of working on this thesis. How the algorithm was adapted to take advantage of graphics hardware to improve performance will be discussed in Chapter 7. The last remaining practical aspect of our implementation, namely rendering, is discussed in Chapter 8. Finally, in Chapter 9 and Chapter 10, we will talk about our results and conclusions drawn while working on this thesis.

# Chapter 2

# Related Work

In this chapter we will firstly talk about a few previously proposed procedural terrain generation methods, and afterwards talk about river generation methods. For a thorough comparison of varied terrain generation with additional considerations, we refer to the work by Galin et al. [16].

## 2.1 Procedural Generation

As previously explained in Chapter 1, procedural generation techniques can create any type of output, provided that an algorithm for the creation of such an output can be created. As can be seen in Figure 1.1, procedural generation can be used for plants, buildings, road networks, cities and many other applications.

One example of a procedural generation technique are L-systems [33]. The central concept of L-systems is rewriting. A complex object is created from an initial simple object by successively using rewrite rules. An example of this can be seen in Figure 2.1.

L-systems can also be adapted to create road networks for complete cities [30]. An example of such a road network can be seen in 2.2. While L-systems are capable of creating architecture, it has since been replaced by something called split grammars [42]. This has been made necessary by the fact that L-systems simulate growth in open spaces, whereas buildings are built in constricted locations, and they are not created by a growth process. Split grammars are built on the basis of shape grammars [39], which are a powerful tool to create buildings. This grammar has since been improved to something called *CGA Shape* in the work by Müller et al. [24], to further increase the quality of the generated buildings. The newest incarnation of CGA, called CGA++, has been introduced in the work by Schwarz et al. [35]. Issues concerning parallelization and limitations of shape grammars have been discussed in the work done by Steinberger et al. [37]. It is also possible to create infinite cities on the GPU [38].

One thing all these architectural approaches need is a terrain to place the generated architecture on. As the cities mentioned previously can be generated infinitely, we also need a way to generate infinite terrain. We will discuss a few different approaches on how to generate terrain in a procedural way in the next section.

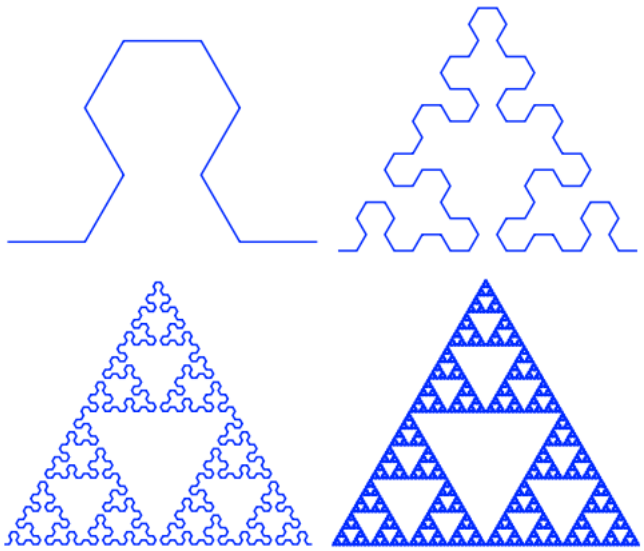


**Figure 2.1:** An example of repeated application of the rewriting rules when using an L-system. *

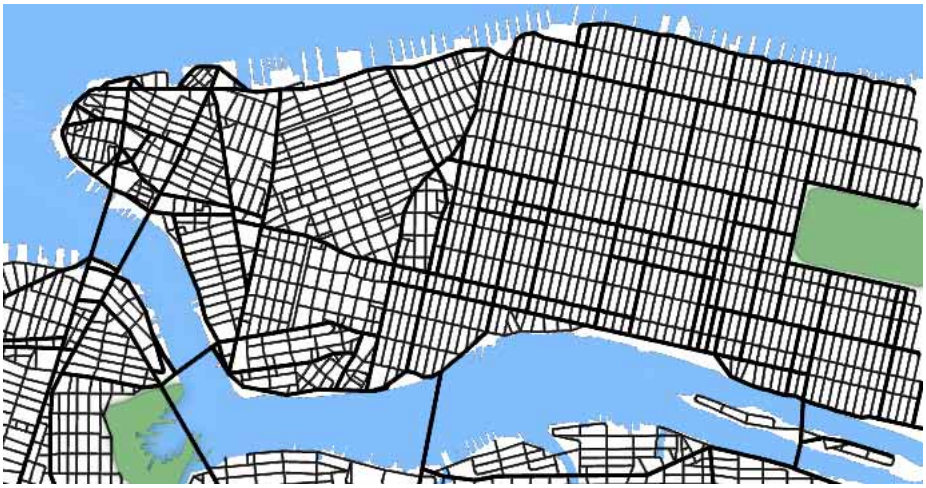
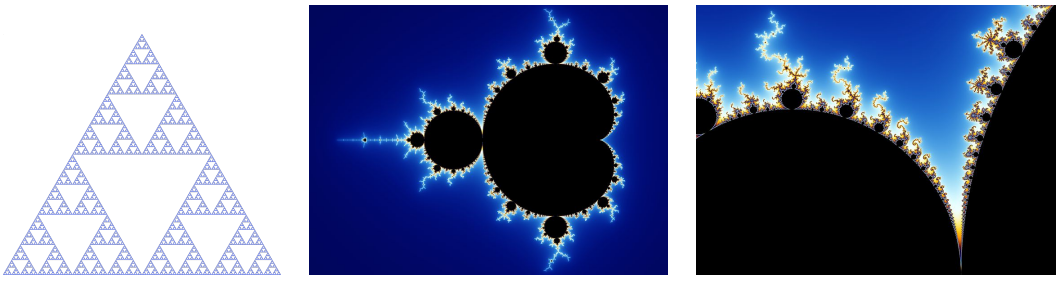


**Figure 2.2:** An L-system based street creation system applied to Manhattan taken from [30].

## 2.2 Terrain Generation

The generation of realistic-looking artificial terrain has been an intense topic of research over the last few decades. The existing terrain generation techniques can be broadly subdivided into procedural, erosion-based, sketch-based and example-based techniques. Erosion-based techniques are generally computationally expensive, but lead to geologically correct terrains. Erosion-based methods will be discussed later in Section 2.3.1. Sketch-based systems generate an artificial terrain based on user-generated input. These inputs

*Original image created by Antónia Miguel de Campos.



(a) The Sierpinski Triangle.[‡]

(b) The Mandelbrot set.[§]

(c) Zoomed in on the Mandelbrot set.[§]

**Figure 2.3:** These figures show examples of fractals.

can range from terrain region and ridges as in the work by Teoh [41], to an uplift map that specifies where the process of tectonics combined with erosion is used to create the terrain as in the work by Cordonnier et al. [6]. Sketch-based techniques are usually used in an interactive setting, where the generated terrain can be directly manipulated [15]. Example-based approaches generate artificial terrain from a sample terrain. The *terrain features* are extracted from the sample terrain. Terrain features are features such as rivers, valleys and mountain ridges. These features are then placed on the new terrain to create a completely new terrain [43]. Guérin et al. [17] expand this idea idea of example-based techniques to represent the whole terrain as elevation functions that are build from linear combinations of such terrain features. They store all their examples in a small dictionary, from which new terrain can be generated. There are also example-based approaches that take advantage of *neural networks*, where the networks are trained with real-world terrains, and the network then generates the artificial terrain [18].

Most commonly used procedural terrain generation methods create something called a fractal terrain. In this context a fractal is defined as "a geometrically complex object, the complexity of which arises through the repetition of a given form over a range of scales." [26] This repetition leads to the object possessing *self-similarity* at all scales. A few examples of fractals can be seen in Figure 2.3.

The reason we are using fractals to generate terrain is because fractals can generate artificial terrain that looks convincing to the casual observer [22]. Some of the earliest images were created by such terrain generation algorithms [21].

As further detail of fractals is not necessary for the purposes of this thesis, we refer to the work by Musgrave et al. [26] and the work by Mandelbrot [22] for more thorough information regarding fractals. What most methods want to imitate is something called fractional Brownian motion (fBm), which are a family of Gaussian random functions. For an in-depth look at fBm, we refer to Mandelbrot et al. [23].

One way to create fractal terrains is by using noise functions. One of the earliest such functions was created in the work by Perlin [31]. As a thorough evaluation of different

[‡]Image by Beojan Stanisiaus, distributed under a CC BY-SA 3.0 license

[§]Image by Wolfgang Beyer with the program *Ultra Fractal 3* distributed under a CC BY-SA 3.0 license

noise functions could fill a thesis by itself, only the bare minimum will be explained here. For an overview of procedural noise functions, we refer to the work by Lagae et al. [19].

To generate terrain by using noise functions, multiple layers of noise are added together, each layer with a different frequency and amplitude. How a heightmap of such a generated terrain may look like, and the different layers it is composed of can be seen in Figure 2.4.

Another algorithm to create fractal terrain is the *Diamond-Square Algorithm* [13]. As this algorithm is used in this thesis, it will be explained later in Chapter 3.6.

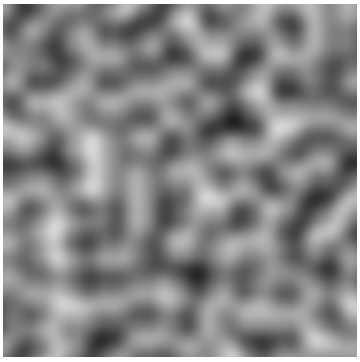
As stated previously, self-similarity is an intrinsic part of fractals. This property however, is the cause for only being able to create uneroded terrain, as the valleys are statistically the same as mountain tops, whereas in reality, valleys and other depressions smooth out over time, as sediment gets deposited in them through erosion. This fact leads to the increased study of erosion algorithms for use on uneroded terrains, which we will discuss in the next section.

## 2.3 River Generation

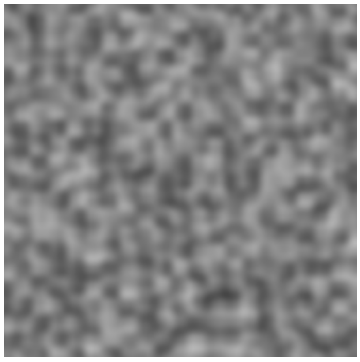
The topic we are focused on in this work is the generation of river networks. The terrain generation methods introduced in the previous section only serve to create uneroded terrain, which means that they do not contain any intentional river networks. There may be configurations present that resemble a river network, but were only generated by chance. We want to focus on the reliable creation of such a network.

As mentioned previously, the main inspiration of this thesis is the work done by Derzapt et al. [11], where they create the river network by using the advantages of their irregular mesh to ease the creation of the river networks. They start by labeling one face on their planet as a continent, and repeatedly add faces to their continents until a user generated threshold has been reached. Afterwards, the initial river network is generated by looking for continental vertices next to coastal vertices and labeling the edge between them as part of the river. The rivers are then grown out from this river mouth onto the remaining continent along the edges. This process is repeated until all continental vertices are part of the river network. During this process, terrain and water heights are also added to each vertex. Finally, the different rivers are separated by inserting continental vertices on the continental edges. They also introduce mountain vertices, that make sure that their created rivers always flow down the steepest decline.

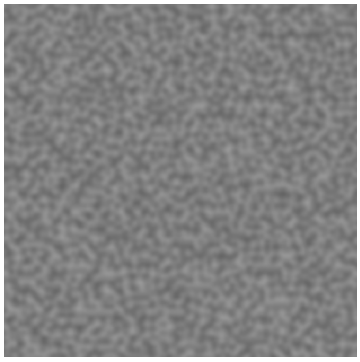
The main difference between this and our work is the type of mesh used. While Derzapf et al. use an irregular mesh, we wanted to examine if it would be possible to achieve similar results by using a regular mesh. By utilizing a regular mesh we can perform the necessary operations of our approach on each part independently of the remaining planet, thus allowing for easy parallelization of our approach. Unfortunately, the regular mesh prevents us from combining our river data structure with our terrain data structure, thus we had to search for an alternative. More about our chosen river representation in Chapter 5.



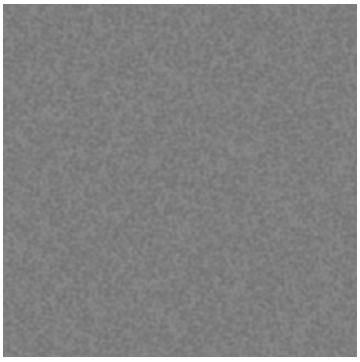
(a) The first layer of noise of the heightmap.



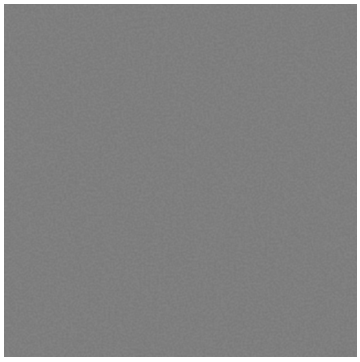
(b) The second layer of noise of the heightmap.



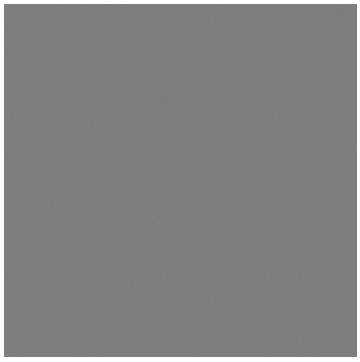
(c) The third layer of noise of the heightmap.



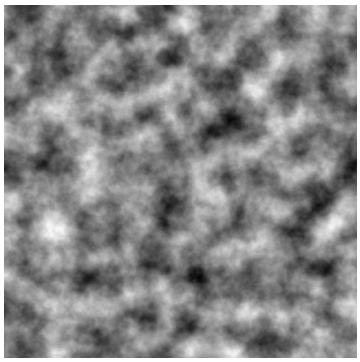
(d) The fourth layer of noise of the heightmap.



(e) The fifth layer of noise of the heightmap.



(f) The sixth layer of noise of the heightmap.



(g) The complete heightmap generated by the 6 layers.

**Figure 2.4:** An example of a heightmap generated via noise functions and all the layers it is comprised of.

### 2.3.1 Erosion Simulation

Erosion, as used in our work, refers to the process of eroding slopes and transporting the eroded sediment to regions of lower elevation. this process creates steeper slopes and makes low terrain more flat. Erosion can be subdivided into different subtypes, depending on the cause that forced the sediment to shift from its place. There is hydraulic erosion for water based erosion, aeolic erosion for wind generated erosion, and many more. What we are most interested in is hydraulic erosion. Of course, all erosion processes are necessary to create a truly realistic terrain.

To perform erosion simulation, one has to first generate a base terrain and perform the erosion process on it afterwards. Depending on the implementation of the erosion algorithm, this can be a very expensive process, as repeated looping over the whole terrain may be necessary.

One of the earliest approaches to creating eroded terrain is the work done by Musgrave et al. [25]. In this work, they create a simple hydraulic erosion process, which consists of depositing water on vertices and then moving it and sediment contained in the water to lower neighboring vertices depending on the water present on that vertex, and a few constants, for example the *soil softness constant*. They also model *thermal weathering*, which refers to other erosion processes, which move sediment to lower altitudes. This is done by calculating the differences between the altitude of each vertex and its neighbors. If the difference is bigger than a constant talus angle, some sediment has to be transported to that neighboring vertex. This paper has become the base from which many improvements have since been suggested [3–5, 9, 27]. While erosion simulation can create very realistic looking terrain, as well as river networks, the cost of performing erosion simulation on the uneroded terrain is simply too high for us to use it in our approach.

A more thorough look at erosion processes in procedurally generated terrain, and concerns regarding implementation and optimization issues, we refer to the work by Olsen [29].

### 2.3.2 Fractal River Generation

Simulation based river generation always consists of two different processes. The first one is base terrain generation and the second is the actual simulation of the erosion processes. In the work done by Prusinkiewicz [32], a modification of the midpoint-displacement method has been proposed, where not only new vertices are calculated, but the triangle edges for rivers are also created and subdivided into *entry*, *exit* and *neutral* edges. This way the creation of rivers can be incorporated into the terrain generation step, thus only needing one method to get to the desired terrain including rivers.

Another paper also inverts the order of generation [2]. In a first step of the process, *Ridge Particles* are placed on a heightmap and moved around to imitate *fBm*. After this process is done, a similar pass is done by placing *River Particles*. This way, they first create the skeleton of the ridges and river network. The missing data in their heightmap is then filled with a modified midpoint-displacement algorithm, thus creating their terrain with ridges and rivers in one integrated process.

As we want to preserve the ability to use pregenerated heightmaps, or heightmaps which define the overall shape of our continents as input to our approach, we do not

**Figure 2.5:** The Stanford bunny in different levels of detail. ¶

use a fractal approach to generate our terrain, but rather stick with a simulation based approach.

## 2.4 Level of Detail

Level of Detail (LoD) is a technique to increase or decrease the resolution of a mesh in accordance to the distance to the user. For example, a mountain in the distance does not need a lot of geometry to represent it well enough, whereas an object that is very close to the camera has to be rendered in detail to appear real. If everything would be rendered in the highest detail possible, even modern graphics hardware would be hard-pressed to create the necessary frames per second (fps) for real-time processes to appear smooth.

In videos, this is not such a big concern, because the images are not created in real-time, but rather created before and simply saved to be shown later. As our focus is on the ability to be real-time, we have to consider how to reduce the strain on our graphics hardware and create an algorithm which can be run in real-time.

In our work, the level of detail algorithm is performed on a planetary scale. A more detailed geometry replaces the coarser geometry of the planet once the viewer gets suitably close to the planet. If the viewer is too close to the geometry being replaced, a noticeable visible artifact, called *popping*, is introduced. Thus it is crucial to the level of detail scheme to perform this switch of geometry at larger distances, but too large a distance leads to switching the geometry too early, which can be a strain on the graphics hardware and decrease the frames per second if a detailed mesh is rendered without it being necessary.

In Derzapf et al. [11], this has been achieved by using their irregular mesh data structure and performing *vertex split* and *edge collapse* operations on their mesh, if the user moves closer or farther away from their current viewpoint.

For a more thorough look at level of detail algorithms, we refer to the work done by Luebke et al. [20].

¶Original image created by Trevorgoodchild.

# Part II

# Method

# Chapter 3

# Overview

In this chapter we will briefly explain the general steps of the algorithm and introduce all the necessary technologies to implement this approach.

The first step of our approach is to obtain a base terrain. In our program this terrain is obtained by generating it in a preprocessing step. Alternatively, it can be obtained from other sources, such as digital elevation data, or can be computed on the fly. More about this in chapter 4.

In the second step we generate rivers on our terrain. For this we need the starting and end points. We obtain the starting points by placing them randomly on the terrain. From the starting points we step along the terrain in a random direction until we get to a point below a predefined sea level height. We decided to simply use splines to simulate our rivers so the generation of sample data along the river is an easy task to accomplish. How rivers are created in detail will be explained in chapter 5.

With the base terrain and our river sample data, we move onto the third step. The third step is comprised of calculating the height of every respective vertex if it is displaced by the river. This is the core of our new approach and is discussed in chapter 6.

As previously mentioned, one thing to note is that instead of a combined data structure for river and terrain data, as used in the work of Derzapf et al. [11], we use two different data structures. This is made necessary by the fact that our used terrain structure is a regular mesh so we can not add detail wherever the river needs it to be rendered.

In the following, we will briefly discuss all the techniques relevant to our approach.

## 3.1 Planet Representation

We want to avoid working on a round surface to simplify our work. The logical step from this choice is to work on a cube. All points on the cube can be normalized to get positions on the unit-sphere. Thus we have a mapping from a cube to the unit-sphere. Scaling the points with the radius of the desired planet, we get a round surface, consisting of six parts that can be handled individually, as long as care is taken across face boundaries. The result of this can be seen in Figure 3.1.

By looking closely at the resulting sphere, there is a noticeable distortion created by this mapping. Triangles further from the center of the quadtree cover a smaller region of

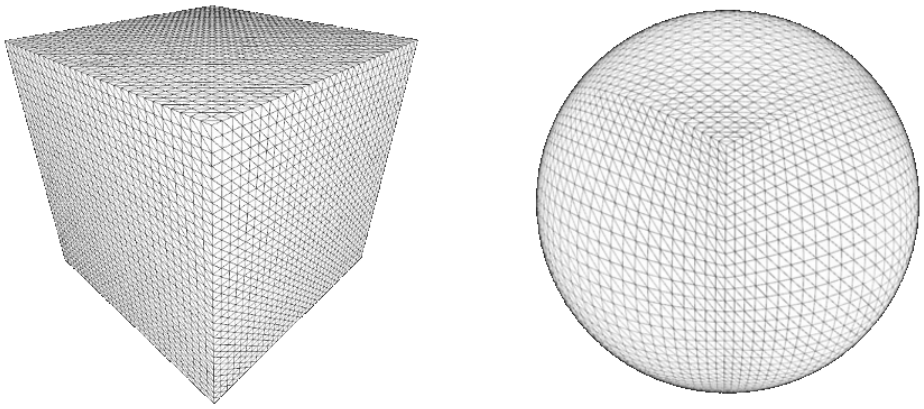


**Figure 3.1:** The cube made up of 6 quadtrees on the left and the normalized version on the right.

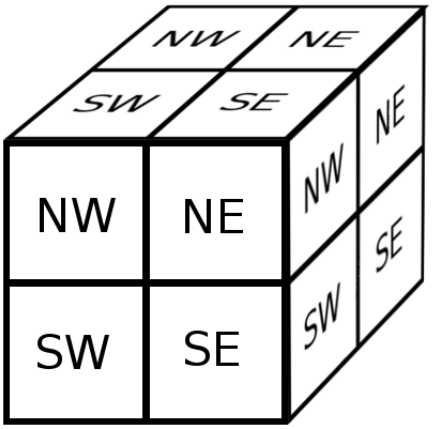



**Figure 3.2:** This figure illustrates the subdivision of the 6 cube faces into four quadrants and shows the problem with determining neighboring patches.

space than triangles closer to the center. This needs to be taken into account by the LoD algorithm when it is time to create the appropriate new geometry.

Unfortunately, there remains another problem with this representation, as great care needs to be taken in determining neighboring patches across face boundaries. To make this problem more apparent, we subdivide each face of the cube into four quadrants (northwest, northeast, southwest, southeast). By looking at the top face in Figure 3.2, it is easy to see that if you step in a eastward direction across the top face, you ultimately end up stepping southwards on the right face of the cube.

(a) An example of a quadtree with a few subdivisions.

(b) Subdivision of a quadtree in a LoD method. The red dot indicates observer position.

**Figure 3.3:** Quadtree examples.

## 3.2 Quadtrees

As a next step we need to choose a suitable LoD algorithm for our planet representation, so we can add and remove detail based on the observer location. As we are now in the position of having 6 cube faces, we opt to use *quadtrees*. Quadtrees are a tree data structure, where each node has exactly four children. They are most commonly used, as they are in this work, to recursively subdivide regions in two-dimensional space as can be seen in Figure 3.3(a). Every node (and leaf) in the quadtree will be called a *patch* in the remainder of this work.

While the data the nodes contain are different for every problem they are used to solve, in our case we use quadtrees to implement a Level of Detail (LoD) method, as such they contain position and size information for each patch. Additionally, they contain a heightmap, which is discussed later. The LoD method is achieved by subdividing the space more, the closer the observer is to the object one wants to observe, which yields a more subdivided quadtree and thus, in our case, a more detailed mesh as can be seen in Figure 3.3(b).

As each patch in the tree works on a separate region of space, parallelization can be easily achieved as long as we take care of issues along borders of regions. By replacing each of the six faces of our planetary cube, we end up with a planet representation that is very easy to work with. For more information on quadtrees, we refer to the work by Finkel et al. [12].

## 3.3 Mesh

Every patch in our work is rendered by using the same basic mesh. It is a regular quadratic triangle mesh with the vertices arranged in grid form. This mesh is moved, scaled and rotated depending on the location and size of the patch it represents. So each face of our cube can be made up of a large or small number of meshes, depending on the LoD algorithm. When more meshes are displayed for the same region of space, more detailed geometry will be shown on that part of the planet.

The rendering mesh contains a total of $2^n + 1 \cdot 2^n + 1$ vertices. In our case, we find that a 33x33 mesh yields sufficient detail for rendering, but this can be easily changed. The number of vertices in each dimension stems from the used method to generate our terrain discussed later in section 3.6.

## 3.4 Heightmaps

Heightmaps are two-dimensional arrays used to store data. In our case, we store elevation data to implement displacement mapping in the heightmap. They are often expressed as a gray-scale image as seen in Figure 3.4(a).

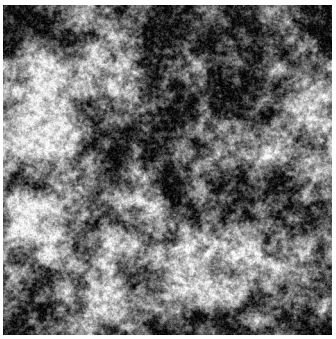
Displacement mapping is done by displacing the vertices of the mesh according to the values stored in the heightmap, thus turning a flat two-dimensional mesh into a three-dimensional mesh as seen in Figure 3.4(b). As the values stored in the heightmap are between zero and one, to calculate the actual displacement, we need a reference value in order to determine the actual displacement at run-time. This difference can be seen in Figure 3.4(b) in comparison to Figure 3.4(c). Heightmaps are also used in Digital Elevation Maps (DEM), where the heights of our planetary surfaces are scanned and stored for use in arbitrary applications.

Combining heightmaps with quadtrees yields a way to create terrains at adaptive levels of detail by either precomputing all the heightmaps or computing them on the fly as needed as determined by the level of detail method.

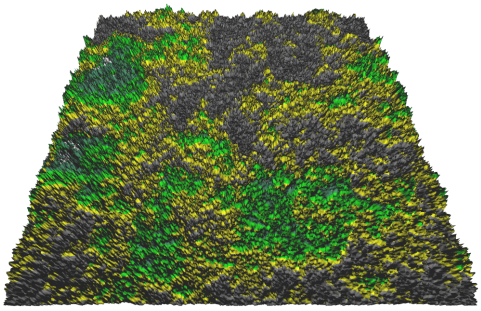
However, there are a few issues heightmaps face that need to be taken care of if complete realism of the terrain is the goal. There is only one height stored per vertex in a heightmap, so there can be no caves or overhanging terrain, as multiple heightmaps would be needed to create this geometry. As the focus of this work was not on the generation of caves or overhanging terrain, we opted to accept this limitation, because heightmaps are otherwise very easy to work with.

## 3.5 Voronoi Diagrams

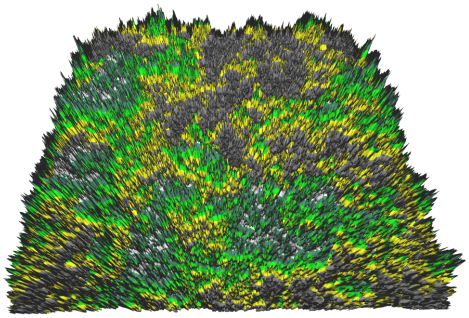
We want a simple way for a user to decide how many continents the planet created by our method should have. To achieve this, we use Voronoi diagrams. A Voronoi diagram is made by dividing a plane into different regions. The way these regions are formed is by starting with certain seed points on the plane. Each seed point creates its own region, which contain all points that are closer to the seed point of the region than any other seed point. Each of these regions is also called a Voronoi cell. An example of such a diagram



(a) A heightmap expressed as a gray-scale image.



(b) The heightmap in 3.4(a) displaced to get a three-dimensional mesh.



(c) The same heightmap displaced with a larger reference value.

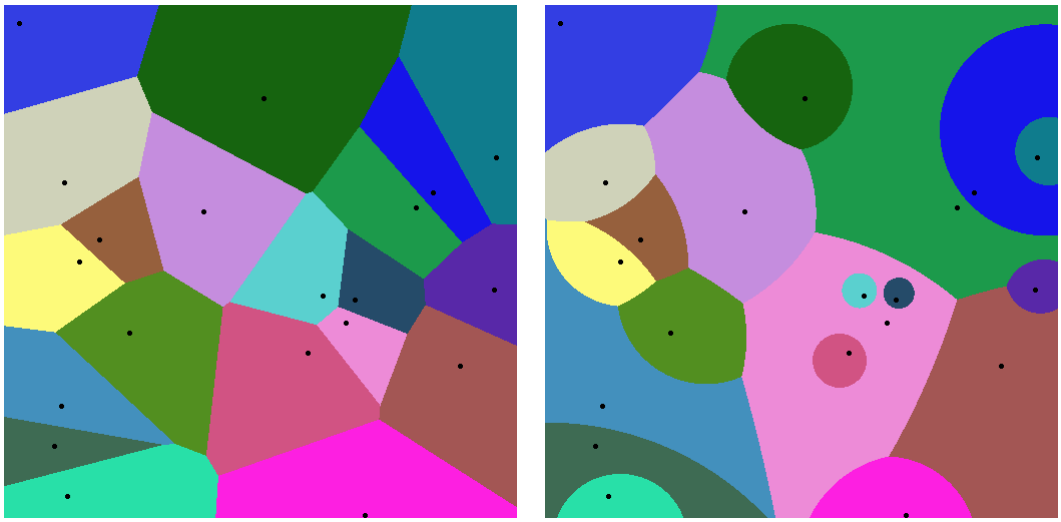
**Figure 3.4:** A heightmap as a gray-scale image and displacement mapped with 2 different values. The different colors are used to differentiate elevations more easily.

can be seen in Figure 3.5(a). By creating a seed point for each continent, we can assign different vertices to different continents, but we still need water to separate the different continents. How this is done will be explained later in Chapter 4.1.

For a more in-depth look at Voronoi diagrams, we refer the reader to the work by Aurenhammer [1].

### 3.5.1 Weighted Voronoi Diagrams

We also want to give the user control over the percentage of the terrain that is covered by the created continents. With normal Voronoi diagrams, new seed points would have to be generated until this condition is satisfied. The solution to this is something called a weighted Voronoi diagram. In contrast to a normal Voronoi diagram, in the weighted case, weights are applied to the distances. Depending on the weights, regions can thus become larger or smaller. An example of this can be seen in Figure 3.5(b) By using weighted Voronoi diagrams we can decrease the weights for seed points to enlarge the resulting



(a) An example of a Voronoi diagram with 20 feature points.

(b) The same Voronoi diagram with weights added to the distances.

**Figure 3.5:** Example of a Voronoi diagram with and without weights.

Voronoi cell, or increase the weight to reduce the cell. In our case, this directly leads to shrinking or enlarging the resulting continents on our planet.

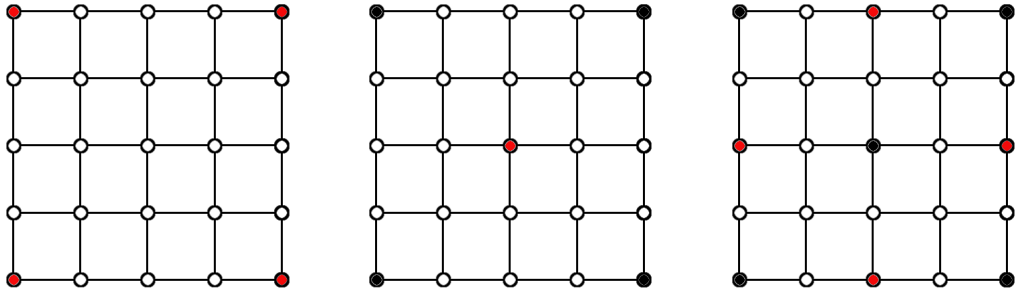
## 3.6 Diamond-Square Algorithm

The diamond-square algorithm [14] is an algorithm to create artificial terrain. The algorithm needs a terrain with sizes of $2^n + 1$ in each dimension to work, which is the reason for our chosen mesh dimension as mentioned previously in 3.3. We use this algorithm in our work, because performing both the diamond and the square step create exactly the data needed for the next subdivision in our quadtrees. So to create the necessary data for the next LoD for each patch, it is only necessary to perform the diamond-square algorithm once. This makes it easy to implement it for on-the-fly generation of additional detail. The resulting terrain created by this algorithm is then later modified by our river networks.

The algorithm starts with the corner values set to some initial (random) value. Afterwards two steps are repeated until all the needed data has been created.

### The diamond step

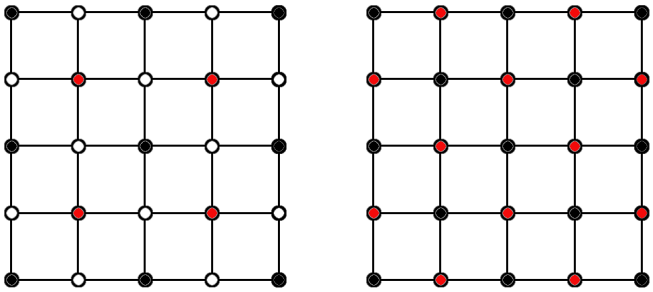
For each square in the terrain (made up of already assigned values) the middle point of the square is determined by calculating the average of the four corners plus a random value.



(a) Only the initial corner values are assigned.

(b) First diamond step

(c) First square step



(d) Second diamond step

(e) Second square step

**Figure 3.6:** These figures show two iterations of the diamond-square algorithm on a 5x5 grid (heightmap). The first iteration is comprised of Figures 3.6(b) and 3.6(c) and the second of Figure 3.6(d) and 3.6(e). The new calculated points are indicated by the red dots.

### The square step

For each diamond in the terrain (made up of already assigned values) the middle point of the diamond is determined by calculating the average of the four corners plus a random value. When performing the square step on corner values, only three of the necessary four points are present. In order to get correct values, we saved data from the neighboring heightmap to use in this step. Without doing this, there would be holes along the patch borders through which one would be able to see into the planet.

The randomly added value in each step has to become smaller the more often the two steps have been executed, as large changes in height should only occur at coarse resolutions in the terrain. Figure 3.6 illustrates how the algorithm would be done on a 5x5 grid.

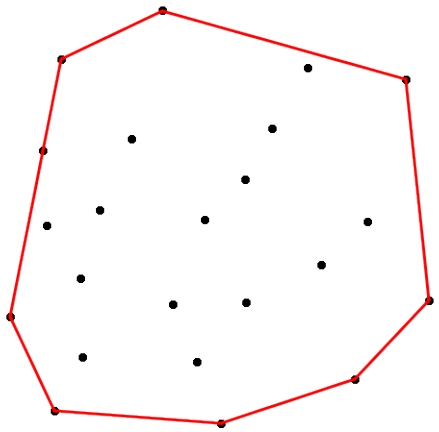


**Figure 3.7:** The convex hull in red defined by the points in black.

## 3.7 Convex Hull

The *convex hull*, as used in this work, is defined as the smallest set of points $Y$ of a given set of 2D points $X$, which contains all points of $X$. This is usually illustrated by a rubber band stretched around the set of points. An example of a convex hull created by a number of points can be seen in Figure 3.7. There are many applications for convex hulls, but in our work we use the convex hull to determine whether or not a patch is modified by our river network. Each river has a convex hull which is used to calculate intersections between itself and any given patch. If there is such an intersection, that patch will be modified by the river and thus has to get the necessary data from our algorithm.

There are a few ways to calculate the convex hull of a given set of points. We opted to implement a simple gift wrapping algorithm, which can be found in Algorithm 1. While this algorithm is not the best from a performance standpoint, we use it because it is only run once for each river, and is also very easy to implement. For a more in-depth look at convex hulls we refer to the work done by Cormen et al. [7].

## 3.8 Splines

As mentioned previously, our approach does not depend on any particular representation of our river networks. We use simple splines for our implementation because they have some nice properties to take advantage of. In this section we will briefly discuss what splines are and the properties we utilized in our work.

Splines are parametric curves which are used for interpolation. They are widely used by artists for animations, fonts and other applications. The type of splines we are interested in are defined piecewise by polynomials and there is a wide range to choose from when trying to interpolate data.

Multiple splines can also be joined together to make more complex paths or to achieve more control over the overall shape. For more information on splines we refer to the work

**Procedure 1** Gift wrapping algorithm for the convex hull.

```
Input: X, the set of points
Output: Y, the set of points on the convex hull
  procedure convexHull(X)
      pointOnHull = point with lowest y-coordinate
      i = 0
      repeat
          Y[i] = pointOnHull
          endPoint = X[0]
          for j = 0 to |X| do
              if endPoint == pointOnHull OR X[j] on the left of Y[i] to endPoint then
                  endPoint = X[j]
              end if
          end for
          i = i + 1
          pointOnHull = endPoint
      until endPoint == Y[0]
  end procedure
```

by De Boor et al. [10].

### 3.8.1 Bézier Splines

We use cubic Bézier splines because they are easy to work with and have nice properties we can use for level of detail. A cubic Bézier spline is defined by four points $(P_0, P_1, P_2, P_3)$, which are called control points. The spline starts and ends in the first and last control point, while the other points are generally not on the spline.

The explicit form of the cubic Bézier curve is as follows:

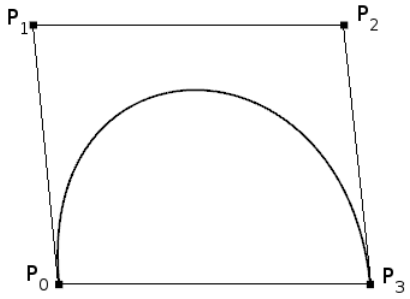
$$B(t) = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t) t^2 P_2 + t^3 P_3, 0 \leq t \leq 1. \tag{3.1}$$

This is an *affine combination*, which means that all coefficients sum to one. This has the nice property that the curve is fully contained in the convex hull created by the four control points as can be seen in Figure 3.8.

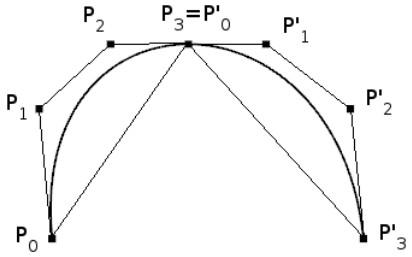
#### 3.8.1.1 Subdivision

While splines can be joined together to create larger splines, it is also possible to subdivide a single spline into multiple smaller splines without changing the overall shape of the original curve.

An important part of this subdivision is continuity. Continuity is a measure of the smoothness of a curve. The smoothness of a curve can be divided into different classes based on how many derivatives of the curve are continuous. Depending on how many derivatives are continuous, we talk about $C^n$ continuity, where $n$ is the number of continuous derivatives.

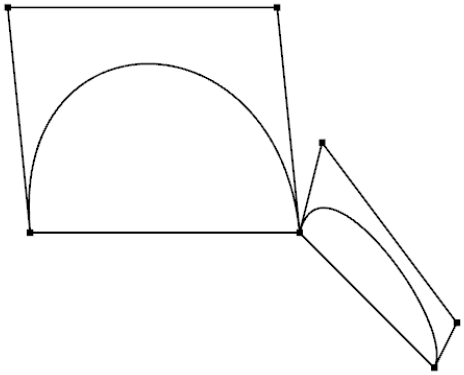



(a) A basic cubic Bézier curve with its convex hull.

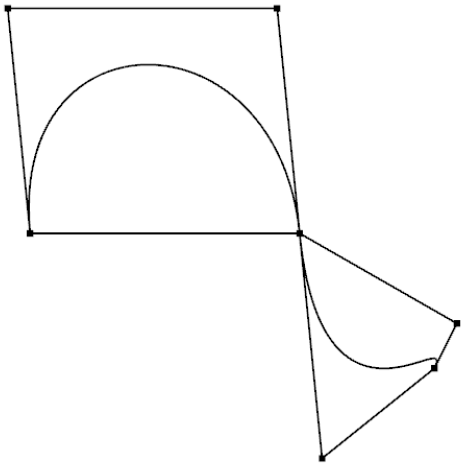



(b) The same curve as in 3.8(a), but subdivided once at $t = 0.5$. Again with convex hulls.

**Figure 3.8:** A Bézier spline



(a) A $C^0$ continuity curve. Only share one point, not the first derivative.



(b) A $C^1$ continuity curve. Shared control point and first derivative is continuous.

**Figure 3.9:** This figure shows a $C^0$ continuity curve in 3.9(a) and a $C^1$ continuity curve in 3.9(b). Note the collinearity of the control points in the $C^1$ case.

A $C^0$ curve, as seen in Figure 3.9(a), only shares one control point, but the first derivative is not continuous.

We are interested in $C^1$ curves because they can be joined together easily and we are able to subdivide a cubic Bézier curve at any $t$ to form multiple curves, which still retain the same overall shape. This subdivision is accomplished simply by calculating the point

of the curve at $t$, which is the new end point of the first spline and the start point of the second spline. Additionally, we need the first derivative at the chosen $t$ with which we build the control points we need to complete the two curves. This derivative also needs to be scaled according to the chosen $t$ so the overall shape is kept. This subdivision can be done an arbitrary number of times. One such subdivision with $t = 0.5$ can be seen in Figure 3.8(b).

The first derivative of a cubic Bézier curve is calculated by Equation 3.2.

$$B'(t) = 3(1-t)^2(P_1 - P_0) + 6(1-t)t(P_2 - P_1) + 3t^2(P_3 - P_2). \tag{3.2}$$

### 3.8.2 Cubic Hermite Splines

Cubic Hermite splines are another representation for cubic splines. In contrast to the Bézier splines, they are not defined by four control points, but rather by two points $(p_0, p_1)$, and the tangents at those two points $(d_0, d_1)$.

Cubic Hermite splines and cubic Bézier splines produce the same curves and are easy to convert from one representation to the other.

This representation is better for us to work with, which will become obvious in Section 5.2.1. The calculation of the convex hull will be performed on the Bézier representation.

For completeness, we provide the equations for the Hermite spline (Equation 3.3) and the first derivative (Equation 3.4).

$$p(t) = (2t^3 - 3t^2 + 1)p_0 + (t^3 - 2t^2 + t)d_0 + (-2t^3 + 3t^2)p_1 + (t^3 - t^2)d_1, 0 \leq t \leq 1. \tag{3.3}$$

$$p'(t) = (6t^2 - 6t)p_0 + (3t^2 - 4t + 1)d_0 + (-6t^2 + 6t)p_1 + (3t^2 - 2t)d_1, 0 \leq t \leq 1. \tag{3.4}$$

# Chapter 4

# Base Terrain Generation

In this chapter we will discuss how the base terrain for our approach is generated. As the first step of this process, continents are created on the planet and the initial height data is determined. Afterwards mountains are distributed on the continents. These mountains are the starting points of our rivers, and will be passed onto the next step in Chapter 5. Note that this terrain generation step is not necessary for our approach to work, as we only need a base terrain, which can come from any source in the form of heightmaps.

## 4.1 Continents

As was already mentioned in Section 3.5, we use Voronoi diagrams to create our continents. We do this by attaching a type to all our feature points. Every feature point is either a continental or water feature point. All points in the Voronoi cell described by the feature point can then be classified as belonging to a continent or the sea. One important thing to keep in mind during the remainder of this chapter is the fact that we are now working on a sphere and not in two dimensions. This can lead to giant Voronoi cells, if there is only a small number of feature points present.

In a first pass, we randomly distribute continental feature points on the sphere representing our planet. Some care needs to be taken to ensure that these continental feature points are not created too closely together, which could lead to them merging very easily when creating the Voronoi cells.

To separate the continents from each other and to attempt to prevent them from merging with each other, we surround each continental cell with a number of water feature points. The distribution of the water feature points is thus a more involved process. If there is only a single continent configured, we create a single water feature point on the opposite side of the planet. This creates only two Voronoi cells on our planet, one being the continental cell, the other being the cell containing all the vertices making up the part of the planet that is below sea level. An example of what the Voronoi cells could look like with only one created continent can be seen in Figure 4.1. If more continents have to be created, we create a total of four water feature points for every pair of continental feature points. The whole process is illustrated in Figure 4.2. The first of these points is generated in the middle of this pair of continental feature points. This creates an initial

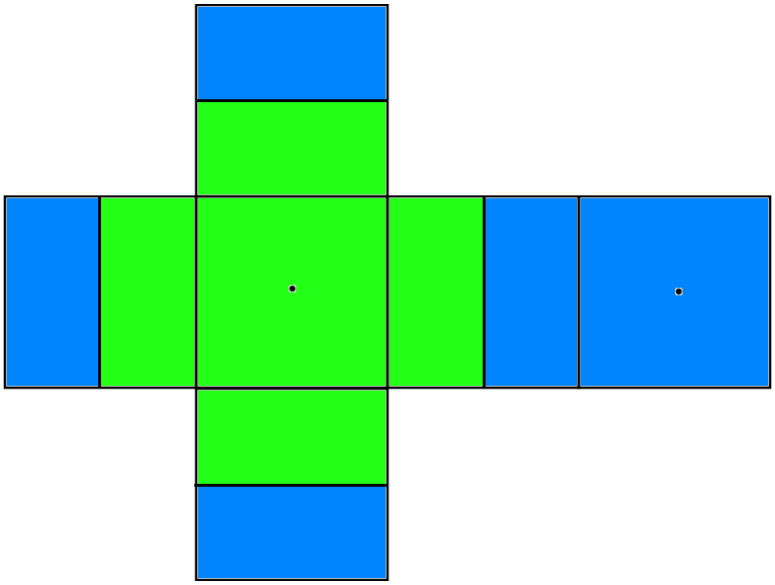
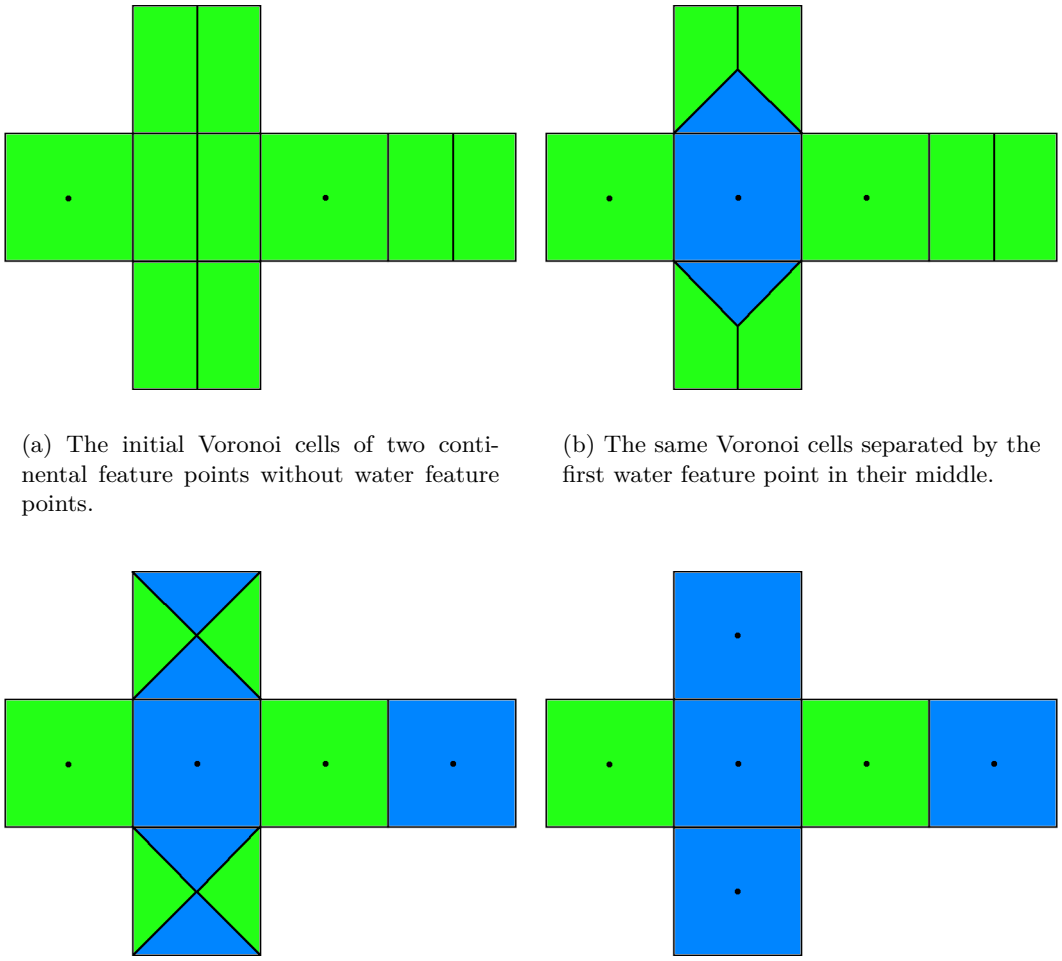


**Figure 4.1:** The six heightmaps of the planet unfolded with only a single continent. The green Voronoi cell represents the continent, while the blue one represents water.

separation between these two continents. If we only had these two continents and this single water feature point, there would only be a small patch of water, while the rest of the sphere would be covered by the continental Voronoi cells, as nothing would further restrict their growth. This can be seen in Figure 4.2(b). The second water feature point is used to prevent the pair of continents from merging on the opposite side of the planet. This second point is thus placed on the negative of the first water feature point. An illustration of this can be seen in Figure 4.2(c). If you were to think of our two continental feature points in two dimensions aligned on the x-axis, we would now have separated them along the x-axis. We still have to make sure that they do not merge along the y-axis. In three dimensions, we do this by adding the third water feature point, belonging to this pair of continental feature points, to the cross product defined by this pair of points. The fourth and final water feature point is added at the negative of this cross product. These four water feature points are able to restrict the growth of continents reasonably well for our purposes. The addition of the last two water feature points can be seen in Figure 4.2(d).

Now that all the necessary feature points have been distributed on the sphere, the Voronoi algorithm is performed on the coarsest level of the six heightmaps. For each point in the heightmap, the corresponding Voronoi cell is determined, and the height of the feature point is stored. This splits all points in the heightmaps into having one of two heights. One signifies that this point is part of a continent, while the other is below the supplied sea level.

As we want to have a variable ratio of continents to ocean, the simple Voronoi diagram was not enough for our purposes. We thus opted to use a weighted Voronoi approach. As already mentioned in Section 3.5.1, this means that we do not use the distance to every feature point in our Voronoi algorithm as is, but instead use a scalar to scale this distance. Because we only have to differentiate between continents and water cells, we only apply



(a) The initial Voronoi cells of two continental feature points without water feature points.

(b) The same Voronoi cells separated by the first water feature point in their middle.

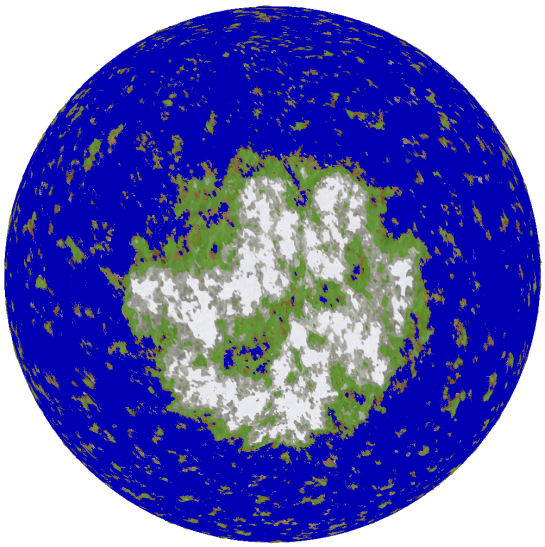
(c) Second water feature point added on the opposite side of the planet.

(d) The last two water feature points added ad the cross product and its negative created by the two continental feature points.

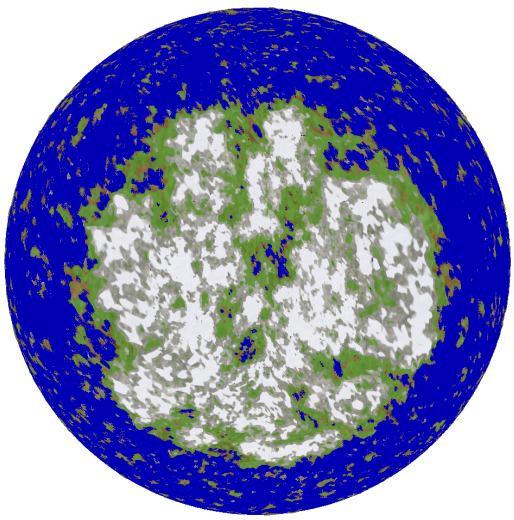
**Figure 4.2:** An illustration of how water feature points are added for every continent feature points pair and the resulting Voronoi cells on the six heightmaps of our planet.

a single weight to the continental feature points. By increasing or decreasing this weight, the cells of the continents can thus shrink or expand to satisfy the percentage of landmass required by the user. As it can be quite hard to achieve the exact percentage supplied by the user, a small epsilon is used to get a result that is close enough. An example of a single continent with different percentages of coverage of the generated planet can be seen in Figure 4.3.

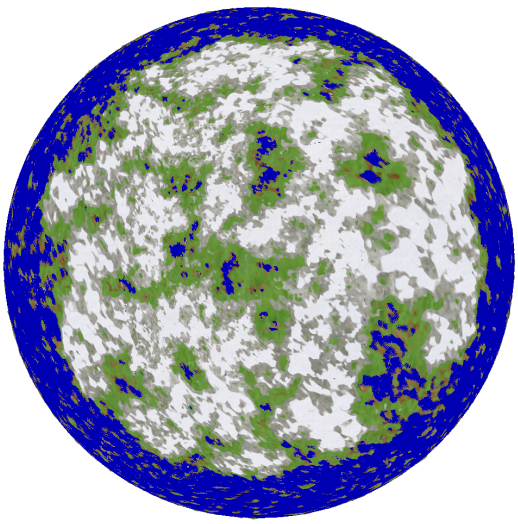
Unfortunately, this weighted approach has one downside, as continents can still merge with one another, if the weights applied to the continental feature points become small enough. The smaller the weight applied to the distance, the larger the cell will become.



(a) The continent created with 5% coverage.



(b) The continent created with 10% coverage.



(c) The continent created with 15% coverage.

**Figure 4.3:** A single continent created with different percentages of landmass coverage.

As our focus was not on the generation of continents, we opted to accept this limitation.

Another issue arises from the usage of Voronoi diagrams for creating continents, namely the regularity of the continent edges, which can be seen in Figure 4.4(a). To remove the regularity of these edges, we perform an additional filtering step on the heightmap after the Voronoi algorithm has been performed. For each point in the heightmap, we step in a random direction for a random distance. The value in the heightmap stored at that target point is then assigned to the value of the original point. This *perturbation* operation, which can also be found in the work by Musgrave et al. [26], is thus used to create irregular edges along the continent boundaries. An example of this in an early prototype with two different maximum distances for the filtering step can be seen in Figure 4.4.

## 4.2 Height Data

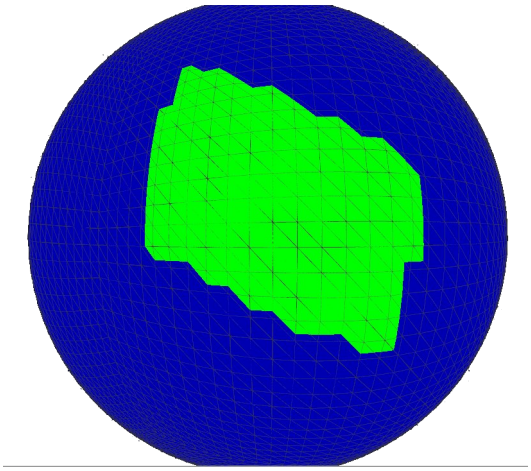
At this stage in the base terrain generation process, there are only two different values stored in the heightmaps. These are, as mentioned previously, used to separate continental points from points belonging to the sea. This means we are faced with completely flat terrain with sharp edges along the continent borders, and flat terrain, albeit with lower height, for our sea vertices.

To create a more interesting looking terrain, and also to have a starting point for our rivers, we seed mountains onto our continents. We do this by randomly distributing the mountains across our continent. A point is randomly chosen on our heghtmap. After the point is chosen, we investigate its neighborhood, i.e. the 8 points surrounding it. If all eight neighboring points are continental points, and are not part of any other mountain, we increase their heights, to a predetermined mountain height, while the initially chosen center point gets an even larger height, that has also been predetermined. This is comparable to repeatedly applying a mountain *mask* to the six heightmaps. An example of what this can look like can be seen in Figure 4.5.

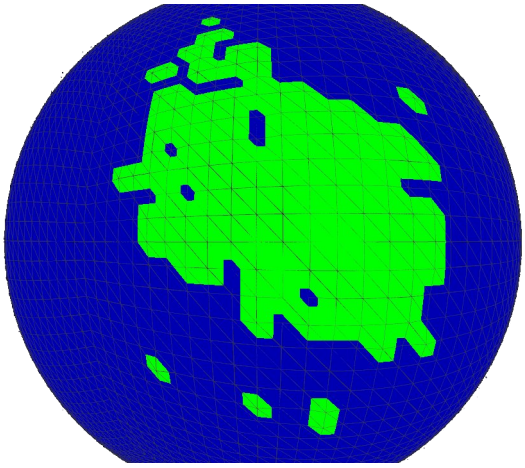
After the mountains have been distributed, we are still only stuck with a very coarse level of detail. To create more detailed terrain, we decided to use the diamond-square algorithm, already discussed in Section 3.6. This algorithm suits our application very well, because a single execution of the algorithm on our heightmap created the next, more detailed, level of detail, that corresponds to exactly one subdivision in our quadtree patch.

One important thing to note is that for the diamond-square algorithm to be consistent across heightmap and patch boundaries, we need to store more information per heightmap, than is actually used for rendering later. By looking closely at Figure 3.6(c), it should become clear that we need a point outside of the grid to accurately calculate the average of the four points. This data needs to be stored in the heightmap, to ensure that patch borders line up correctly on the planet and not create holes, through which one would be able to see into the planet.

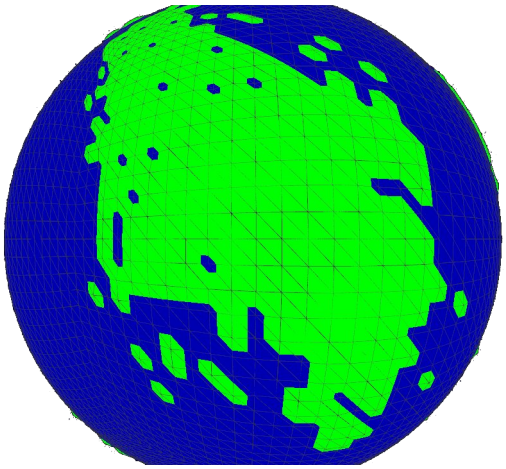
By using the diamond-square algorithm, and taking care of the before-mentioned issues, we end up with a process to create arbitrary levels of detail for our base terrain. It would also be possible to precalculate a number of levels of detail and store them, rather than to calculate them on-the-fly. In our particular implementation of this base terrain generation method, we precalculate all the levels of detail used, but it would not be hard to also adapt



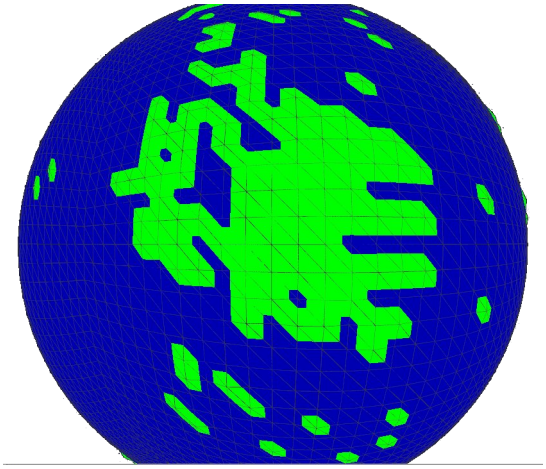
(a) This figure shows a basic continent created by the Voronoi algorithm.



(b) The filtered heightmap with a distance of 4 vertices.



(c) The filtered and weighted heightmap with a distance of 4 vertices.



(d) The filtered heightmap with a distance of 9 vertices.

**Figure 4.4:** Screenshots from an early version with different values for continent generation.

the method to start generating additional detail, when the precalculated highest level of detail has been reached.

As mentioned previously, the height data can also come from another source rather than be generated during the start of our algorithm. In Figure 4.6, we show our approach applied to initial continents formed by using height data of a picture of Earth. The subsequent application of the diamond-square algorithm have distorted the original continent shape, but they continents are still visible. By utilizing more height data, thus eliminating the use of the diamond-square or similar techniques, the actual desired shape of a terrain

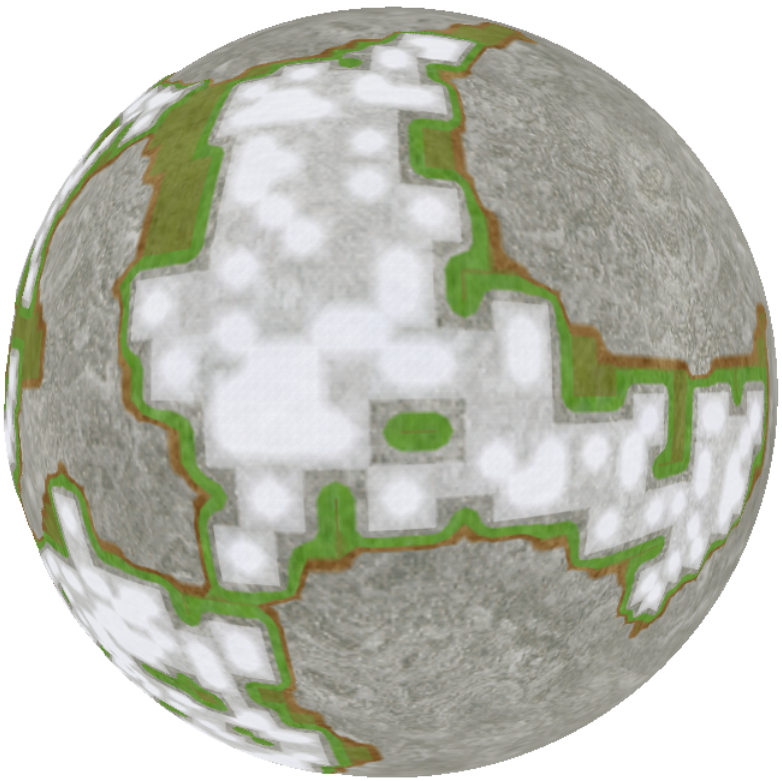


**Figure 4.5:** Continents with seeded mountains (white) and disabled water rendering.

can be created. With such a source of height data it is still possible to generate detail on-the-fly, once the detail in the source has been exhausted.

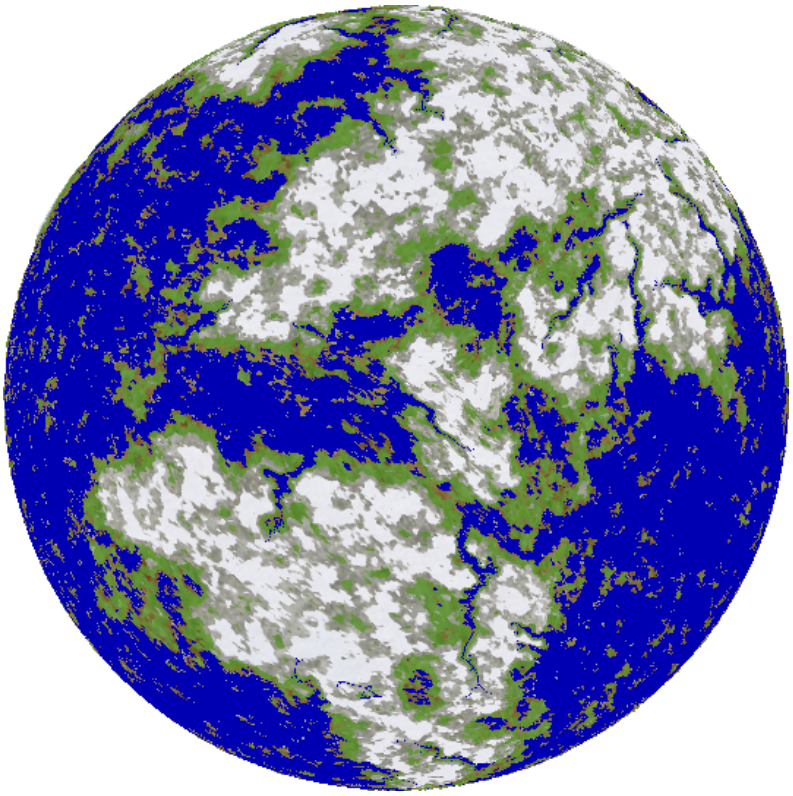


**Figure 4.6:** Planet created with Earth's continent shape.

# Chapter 5

# River Network Generation

In this chapter we will talk about how we generate the rivers used in our implementation. Firstly, we will talk about our chosen river representations and afterwards we will talk about the generation of the rivers, the LoD creation and the generation of tributary rivers.

## 5.1 River Representation

We use a very lightweight representation for the rivers. Rivers are initially made up of a single cubic Bézier spline, which were already briefly discussed in Section 3.8.1. The ease of splitting Bézier splines into multiple splines, thereby retaining the original shape is an important feature we want to have in our river representation to be able to create the appropriate level of detail in a straightforward manner.

It is important to note that for our approach the actual representation of the river is not important, because it only works on sample points created from the river representation. The river implementation could thus be switched as long as it is able to produce the necessary sample data needed for the height calculations discussed later in Chapter 6. This data includes the position of the sample point, the first derivative of the river at this point, as well as the height and width of the river at this sample point. Any arbitrary river representation capable of supplying these four values in the appropriate level of detail are able to be adapted to work in our approach.

We simply store a starting and end height in our rivers, and linearly interpolate between them, to create all the necessary heights in-between these two starting values. An initial width for the river representation can be supplied via a configuration file as well as a scalar, which will be used to increase this initial width until the river meets the river mouth. This is used to simulate the broadening of rivers as more and more tributaries join the river and thus becomes more broad. This way, we similarly linearly interpolate the width between the starting width and the end width. An example of how such a river could look like can be seen in Figure 5.1.

**Figure 5.1:** A sample river becoming thicker towards the end.

## 5.2 River Generation

In this section we will talk about the generation of the rivers, how the necessary levels of details are created, and how we attach tributary rivers to the initial rivers.

### 5.2.1 Initial Rivers

At this stage in our implementation, we have 6 heightmaps, making up a cube that represents our planet, filled with the differentiation between continental and sea points, as well as the mountains created in the last part of Chapter 4. The mountains are created in part to serve as the starting points of our rivers. From that mountaintop, we step into a random direction along the heightmap, until we get to a point that is no longer a continental point. This point will be the end point of the generated river.

As mentioned previously in Section 3.8.2, we use the Hermite spline representation. This is done to make the creation of the necessary levels of detail easier, which will be discussed in Section 5.2.2. What is missing now for our complete river representation are just the tangents at the start and end points. We can obtain these simply by taking the vector $(end\_point - start\_point)$ as a base and rotate it with a random number to get a curve and not a straight line. The rotation is only done for the starting tangent and not the tangent at the end point because that is the point where the river will meet the sea and this way the river will flow straight into the sea and not at an angle.

After this initial river has been created, we subdivide this river into a number of river segments, while still retaining the overall shape of the river. In our case, we find that a subdivision into five smaller segments yields sufficiently good results. In the next step, we calculate the convex hull of this complete river.

This process is repeated until all possible rivers are created with one additional step for all rivers after the first. For each river, we need to make sure that there are no intersections between it and any other river. This can be done by calculating intersections between the convex hulls of the rivers. If such an intersection has been found, we choose a different direction for our river or, after a reasonable number of attempts, simply move on to another mountain starting point. An example of what such an initial river may look like, and its convex hull can be seen in Figure 5.2(a) and Figure 5.2(b) respectively.

### 5.2.2 Level of Detail for Rivers

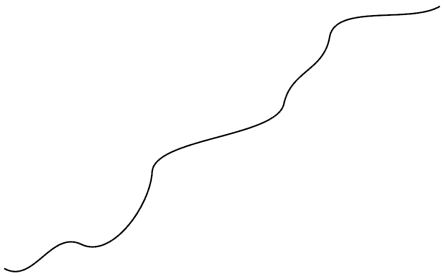
The river representation thus far lacks any interesting detail when the observer gets closer to the planet, as the river would simply remain the same regardless of observer input. Thus, we add more detail to the river representation to create more variety on smaller scales that become only noticeable when getting to the necessary distance.

This is done by taking each segment of our river and subdivide it into segments as well. This subdivision is done similarly as in Section 3.8.1, but we also translate the intermediate points by a random value and also rotate the tangent, while still remaining $C^1$ continuous. This means that the start and end points of the segment (and the tangents) remain the same, but the curve in between changes shape, which adds additional detail to our rivers. Of course this change in translation and rotation have to be made smaller and smaller the finer the level of detail for the river already is, as the reader may remember from the example of the diamond-square algorithm in Section 3.6. To assure that this change is not noticeable, we make it proportional to the length of the actual river segment which is being subdivided, as well as the number of segments it is subdivided into. This way we do not have to also consider the level of detail of the river, as the length of the segment becomes smaller the more detailed the river becomes. We also decided to restrict the rotation of the derivative between 0 and 30 degrees to avoid any major changes to the derivatives at any scale. Figure 5.2 shows a sample river subdivided twice, as well as the convex hull generated by the river.

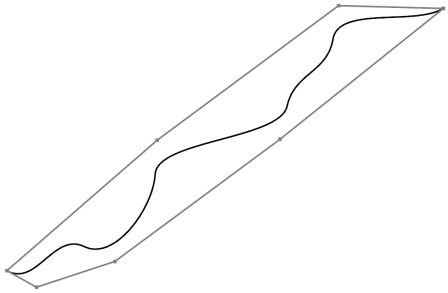
### 5.2.3 Tributary Rivers

As single rivers are not very visually appealing, and because they fit into our framework quite nicely, we add the possibility of generating tributary rivers during the subdivision process to our river representation. A tributary river is a smaller river which flows into a larger one.

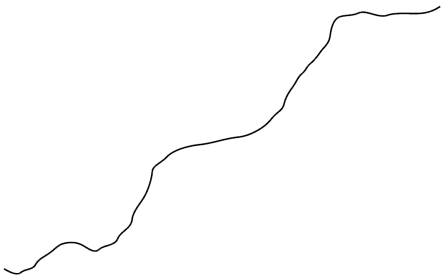
When subdividing the river segments, we randomly decide whether or not a tributary river will spawn on this segment. If a tributary river spawns on this segment, we create a new segment, which shares a point a point on the original segment with $C^1$ continuity. This will be the end of the new segment. The starting point is created by taking the negative derivative at the end point and rotating it. This rotation is kept in the bounds of 22 to 45 degrees, to keep the tributary from intersecting the segment in another place before the actual end point. The length of this newly created tributary will be close to the length of the segment it will be created on, to keep the sudden addition of this segment from being too noticeable to the observer. Thus we end up with an additional



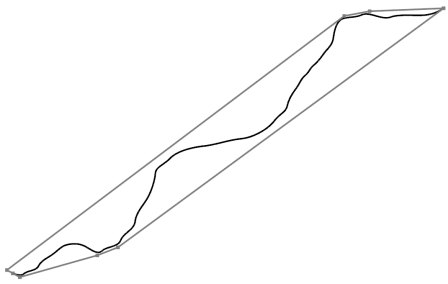
(a) The initial river consisting of 5 river segments.



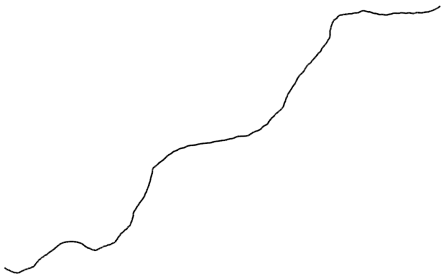
(b) The same river as in 5.2(a) with added convex hull.



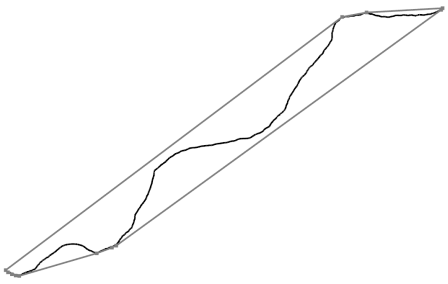
(c) The first subdivision of the river consisting of 25 river segments.



(d) The same river as in 5.2(c) with added convex hull.

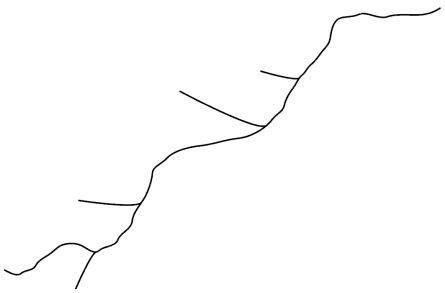


(e) The second subdivision of the river consisting of 125 river segments.

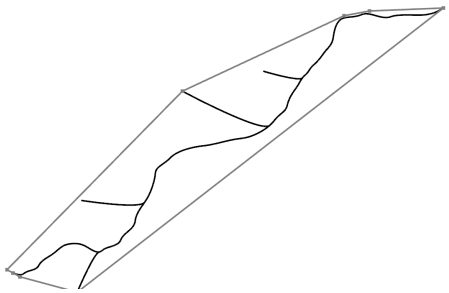


(f) The same river as in 5.2(e) with added convex hull.

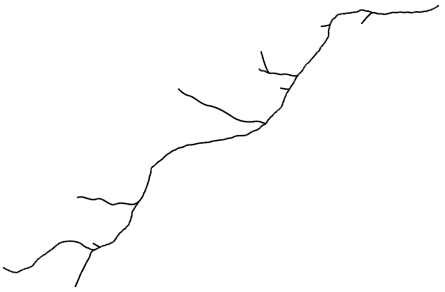
**Figure 5.2:** This figure shows a sample river with 2 subdivisions as well as the convex hull.



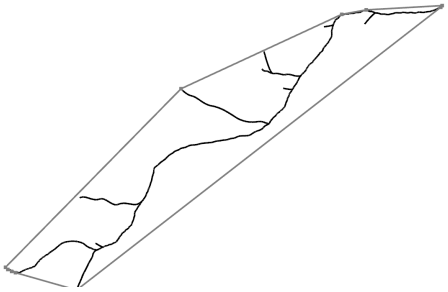
(a) The first subdivision of the river with tributaries.



(b) The same river as in 5.3(a) with added convex hull.



(c) The second subdivision of the river with tributaries.



(d) The same river as in 5.3(c) with added convex hull.

**Figure 5.3:** This figure shows the same river as in 5.2 with 2 subdivisions, the tributary rivers generated by the level of detail method as well as their respective convex hull.

river segment, that flows into the original river segment. An example of how this could look like applied to the sample river we have seen before can be seen in Figure 5.3.

Tributary rivers can impact the size of the convex hull of our river if they are not considered when creating the convex hull. This could lead to mesh patches being modified without their neighbors being affected by the rivers, which would lead to holes between patches. Thus we create a few levels of detail for rivers before calculating the complete convex hull. In our implementation, we chose 3 levels of subdivision, because testing has shown this to be enough as the tributary river size decreases after each subdivision. As this subdivision is only done once and the afterwards created convex hull stored per river, this does not impact run-time performance after the initial convex hull algorithm.

# Chapter 6

# Run-time Algorithm

In this chapter we will talk about the run-time algorithm of our approach. First we will show all the values defined in our configuration file and afterwards we will start with the algorithm. In the first step we determine the appropriate quadtree patches to render. In the next step we create the height data for each patch (if it has not been created yet) by retrieving sample data from all rivers that influence each patch. Lastly, we will show how to use the sample data to create the heights and combine it with the terrain generated by the diamond-square algorithm.

## 6.1 Parameters

In this section we will take a short look at the parameter which influence our approach. These values can be changed outside of our implementation in a configuration file. A sample of such a configuration file with values which work well can be found in Appendix A. Table 6.1 contains all the parameters available to the application, as well as a short description thereof.

## 6.2 Determine Relevant Quadtree Patches

In this section we will talk about how we determine the quadtree patches which are going to be rendered to the screen. This consists of mainly two steps. The first step is to determine the correct level of detail for all the patches for the current observer position, and in the next step remove all patches which will not be visible to this observer. At the end of this step, only the patches that will be visible to the observer are determined.

### 6.2.1 Obtain Necessary Level of Detail for Patches

We utilize a simple distance-dependent level of detail method. To determine the correct level of detail for all patches, we start with the six quadtree roots, which are the faces of the cube of our planet. We subdivide the roots into the four smaller patches of the next quadtree level. For each of these new patches, we determine the distance between the middle point of the patch and the viewer position. It is important to note that this

| | |
|---|---|
| radius | The radius of the planet. |
| continents | The number of continents on the planet. |
| landmass-percentage | The percentage of the planet surface used by the continents. |
| sea-level-height | The water height at sea level. |
| max-mountain-height | The maximum height of the mountains. |
| atmosphere-height | The height of the atmosphere. |
| master-seed | The seed used for all pseudo-random number generation. |
| influence-distance | The influence distance of the rivers on the patches. |
| basin-depth | The depth of the river basins. |
| basin-water-offset | The distance from the water height to the top of the river basins. |
| starting-width | The width of the rivers at the start. |
| min-width | The minimum width for rivers. |
| ending-factor | The factor by which the width is scaled along the rivers. |
| segments | The number of subdivisions for each LoD. |
| vertices | The number of vertices in each mesh direction. Needs to be in the form of $2^n + 1$. |
| mas-level-of-detail | The maximum LoD. |
| level-of-detail-switch-factor | The factor used in the LoD formula. |

**Table 6.1:** The parameters impacting our approach.

distance calculation is not done on the cube representing the planet, but rather the final coordinates these middle points will be projected to on the planet. If this distance is below a certain threshold, we recursively start this process again while utilizing this child patch as the base. If the distance is above or equal to that threshold, the level of detail of the patch is too high, so we do not render that child patch, but rather this quadrant of the base patch. The threshold we use for this level of detail method can be seen in Equation 6.1. The *radius* is the radius of the planet defined in the configuration file, the *switchFactor* is again a value defined in the configuration file and allows for easier testing, and *lod* is the level of detail for each patch. We use 0 to represent the coarsest level of detail, so the higher the level of detail the larger the number becomes. Depending on the size of the radius, the factor may need to be adjusted to achieve satisfactory results.

$$threshold = radius * switchFactor * \frac{1}{2^{lod}} \tag{6.1}$$

By doing this recursive operation on all six quadtrees, we end up with a buffer containing all visible quadtree patches in the correct level of detail as well was the quadrants for each patch which need to be rendered. A rudimentary version of this level of detail method can be found in Algorithm 2.

**Procedure 2** Quadtree Level of Detail algorithm.

```
Input: renderBuffer, the buffer containing all rendered patches
Input: position, the observer position
  procedure SELECTLOD(renderBuffer, position)
    if higher LoD not created then
      createHigherLOD()
    end if
    renderQuadrants = 0
    for all 4 quadrants do
      patch = quadrants(i)
      dist = distance(position, patch.middle)
      if dist < threshold then
        patch.selectLOD(renderBuffer, position)
      else
        renderQuadrants += i
      end if
    end for
    if renderQuadrants ≠ 0 then
      renderBuffer.add(this)
    end if
  end procedure
```

### 6.2.2 Discard Unneeded Patches

The level of detail method described above creates the relevant level of detail for all patches, regardless of whether they are visible to the observer or not. As it would be a waste of computational resources to create height data for patches, which would not be rendered, we need to determine if a patch is visible to the observer or not. In the following sections we will show two methods we use to discard patches which are not visible to the observer. At the end of this process, an estimation of the necessary work our approach has to do will be possible, but only a worst case estimation, as we do not yet have determined if the visible patches are influenced by the river network on our planet.

#### 6.2.2.1 Back-Facing Patches

The first method we use to discard non-visible patches is called back-face culling. What this method usually does is to remove polygons, which face away from the camera. This is done by examining the order the points of the polygon. Depending on their rotation, clockwise or counter-clockwise, the polygon is facing the camera or facing away from the camera. We do not want to cull single polygons, but rather patches that face completely away from the observer position. We do this by simply looking at the dot product of the patch middle and the observer position, which allows us to compare the angle between them. If the angle is reasonably small, we simply discard this patch and continue examining the next until all patches facing away from the observer have been removed from the render buffer.

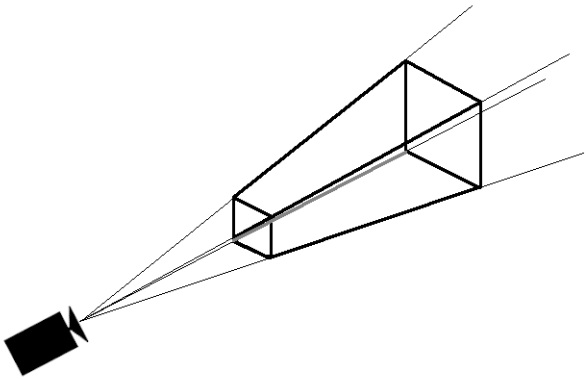


**Figure 6.1:** The view frustum represents everything visible to the camera.

.

#### 6.2.2.2 View Frustum Culling

As a final step in determining the visible patches, we only want to render patches which are in our view frustum. The view frustum can be described as a pyramid without the top, which encompasses everything the camera or observer is able to see. How this looks like can be seen in Figure 6.1. Everything outside this view frustum will not be rendered, and therefore not be visible to the observer. As we need to determine whether or not any part of a patch is inside the view frustum, we create the six planes of our view frustum. For each patch left over in our buffer after the culling of back facing patches, we compare the distances between the middle of each patch and each of the six planes. If the distance is too large, we discard this patch and examine the remaining ones, otherwise, the patch is inside the view frustum and will thus be rendered.

## 6.3 Determine Patches Influenced by Rivers

At this point in the algorithm we have all the patches that need to be rendered. We already have the height data for our patches precomputed, but this would be a good place to perform the calculations to obtain the height data if everything has to be created on-the-fly or memory limitations are a serious concern. We use the convex hull of our river representation to determine whether or not a patch is influenced by a river. The fact that the splines are always constricted to their convex hull is one of the reasons we use splines in our river representation. If we were to use the convex hull of our river representation as is, we would make a serious mistake. This stems from the fact that splines do not have an inherent thickness in contrast to our rivers. We need to extend the convex hull to account for the thickness of our river. We do this by creating something we call an extended convex hull. It is simply the convex hull of the river, but offset from the river with the influence distance defined in our configuration file. An example of how this extended hull may look like can be seen in Figure 6.2. Note that some points on the extended hull may be split

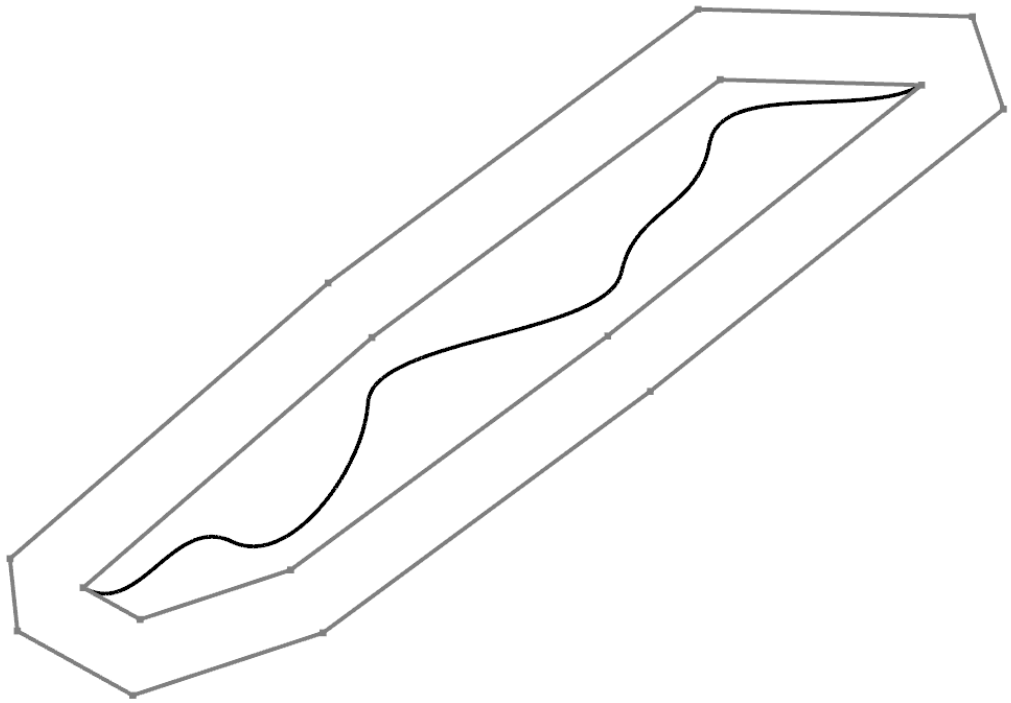


**Figure 6.2:** An example of the extended hull of a river. Note the split points at the far left and far right.

into two points if the offset point would end up too far away from the original convex hull.

For every patch in our list of patches to be rendered, we now calculate intersections between the patch and all rivers on the same cube face, as well as the adjoining cube faces. If such an intersection is found, then this patch is influenced by that particular river. As multiple rivers are able to influence any singular patch, we need to continue until we have examined all possible intersections. At the end of this operation we now have the same list of rendered patches as before, but we also end up with a list of rivers which influence each particular patch. As we only compare the extended convex hull to the patch boundaries, the rivers are not yet on the necessary level of detail for our patches. We now perform the level of detail scheme previously explained in Section 5.2.2 to create the level of detail for each river that corresponds to each patch it influences. A river may influence different patches at differing levels of detail. In Figure 6.3, the modified patches are highlighted in wireframe in red.

## 6.4 Obtain River Sample Data

As previously mentioned, our approach works on sample data obtained from any compatible river representation. Each sample consists of four values generated by the river representation. These four values are the coordinates of the sample point, the first derivative of the river at the sample point, the height as well as the width of the river at the

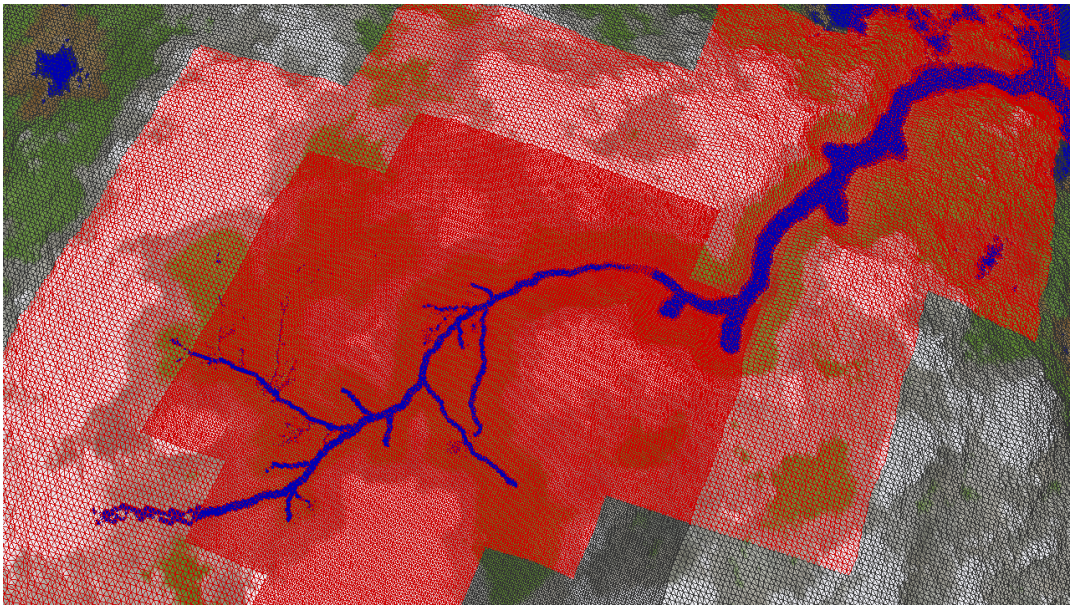


**Figure 6.3:** The modified patches highlighted in red wireframe by the river.

sample point. We currently have a list of patches which are rendered as well as a subset of that list which consists only of patches that are modified by rivers and their associated rivers which modify them. The next step in our approach is to generate the necessary sample data. We generate the sample data for each patch on each river segment at the appropriate level of detail. For our algorithm to correctly identify every vertex situated inside the river basin as being inside the river basin, we need to ensure that we have enough resolution in our sample points. By that we mean that the distance between each sample point and its neighboring sample points is small enough. If this distance is larger than half of the river width, vertices situated in the middle between the sample points and are part of the river basin, may be erroneously thought of to not be part of the river basin, as the distance to both sample points is larger than half the width of the river, and thus does not fall into the river basin. To achieve the desired results, we need to ensure that the maximum distance between two neighboring sample points is at most half the width of the river, but our approach works best if there are three sample points for every distance half the width of the river along the river. This ensures that each vertex, which is part of the river basin, is found to be part of the river basin at least two times. An example of the distribution of samples along the river segment path can be seen in Figure 6.4. At the end of this step we end up with the list of patches which are modified by rivers and the necessary sample data to perform our computations on.

## 6.5 River Height Data Generation

In general, our approach can be described as a weighted sum algorithm applied to all the heights of the relevant sample points. Firstly, we will discuss the calculations done for each vertex of every influenced patch with the appropriate sample points, and afterwards

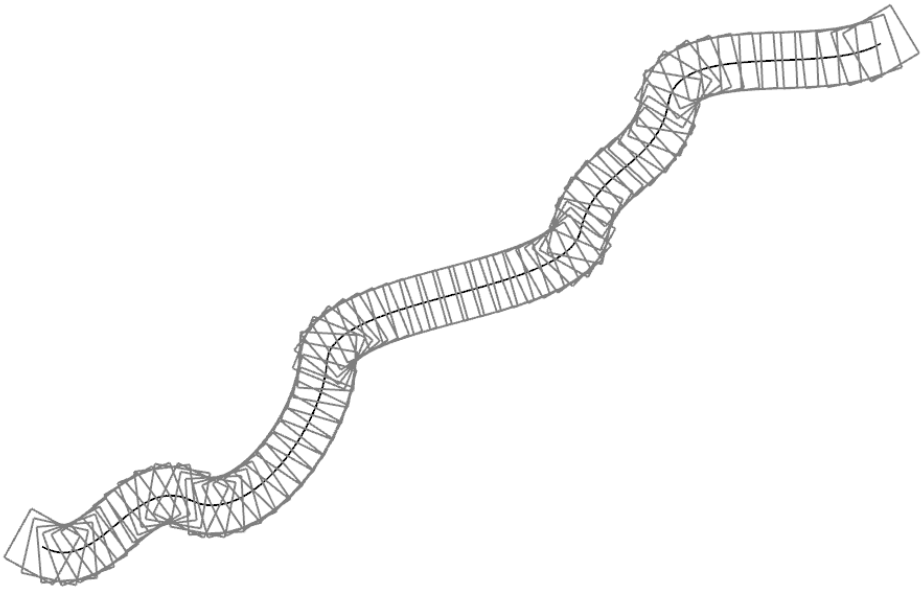


**Figure 6.4:** The samples distributed along the river path. The squares represent the river basin described by each sample.

we will describe how the generated data is utilized to calculate the heights for each vertex to be used during the rendering process.

Before we start discussing the algorithm in detail, we need to define some values used in our algorithm. The core of our approach relies on distance calculations. For every vertex $v$ and every sample point, from here on called $p(t)$, we need to calculate the distance between these points. If this distance is smaller than our predefined influence distance, the sample point is relevant for that vertex. In this case, we need to calculate some additional distances. Firstly, we determine the normal vector at this sample point by rotating the derivative $p'(t)$ by 90 degrees. This is easily achieved by simply exchanging the x- and y-values and negating one of them. As a next step, we project the vertex on the line formed by $p(t)$ and the normal $n(t)$ we just created. With this point, which we will call $v_{n(t)}$, we can now calculate the two remaining distances we need for our approach. The first of these two remaining distances is what we call the *projection distance*. It is the distance between the vertex $v$ and its projection on the sample point and its normal, $v_{n(t)}$. The last remaining distance calculation is the distance between the sample point, $p(t)$, and the projected point $v_{n(t)}$, which we will call the *normal distance*. An illustration of all these values and distances can be seen in Figure 6.5.

With the definitions of these distances out of the way, we can now start to discuss the algorithm in detail. We will present the height calculation as three distinct parts, to make them easier to understand. The first part of the algorithm is performed on each vertex for every sample point. In the beginning, we decide if the given sample point is relevant to the vertex as described previously. We then perform the distance calculations to get the projection distance as well as the normal distance. As mentioned previously, our method

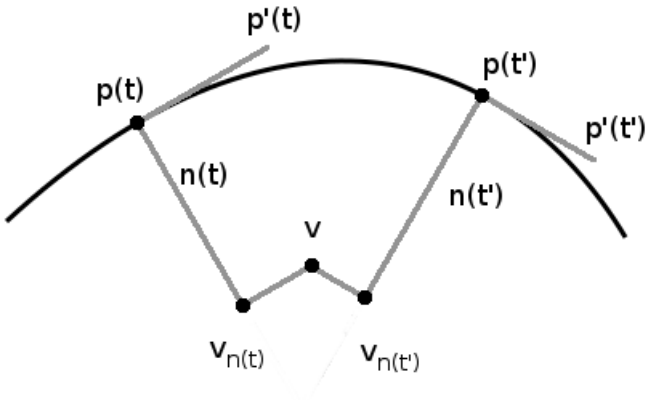



**Figure 6.5:** This figure shows the distances calculated for two different sample points $(p(t), p(t'))$ for one vertex $(v)$. Given the locations of the vertex, the point $(p(t))$, and its first derivative, we calculate firstly the distance between $v$ and $p(t)$, and secondly the distance between $v$ and $v_{n(t)}$, which is the vertex projected on the line formed by $p(t)$ and its normal $n(t)$. Lastly, the distance between $v_{n(t)}$, and $p(t)$ is calculated.

uses weighted sums. There are two different weights used in our approach. The first weight is stronger the closer the vertex is to the sample point. How this weight is calculated can be seen in Equation 6.2. The second weight is used to determine the distance from the middle of the river to the vertex. Its calculation can be found in Equation 6.3.

$$w_{dist} = 1 - \frac{distance(v, p(t))}{influenceDistance} \tag{6.2}$$

$$w_{river} = 1 - \frac{distance(v, v_{n(t)})}{riverWidth * 0.5} \tag{6.3}$$

We now apply $w_{dist}$ to all the necessary sample data, which means the supplied river height, the river basin depth, and the river width at this sample point. As the derivative was only used for the aforementioned distance calculations, it will no longer be used after this point in the algorithm. After this is done, we determine if the vertex is in the river basin defined by the given sample point, as they are treated differently, to create the river basin correctly. We do this by checking if the projection distance and the normal distance are smaller than half of the width of the river at the sample point. If that is the case, the vertex is part of the river basin of the given sample point. We then calculate the river weight we introduced before $(w_{river})$, and determine the depth of the vertex in the given river basin. The weight is also applied to this value. As a last step in this part of the algorithm, we check on which side of the line defined by $p(t)$ and $p'(t)$ the vertex is situated. Depending on which side the vertex is located, we negate the distance between the vertex and the sample point. Why this step is necessary will be explained later in Section 8.3. This part of the algorithm can be found step-by-step in Algorithm 3.

Now that we understand all the calculations performed on each sample point per vertex, we can move on to the next step in our approach. This next step consists of

**Procedure 3** Per vertex per sample algorithm.

**Input:** v, the vertex
**Input:** sample, the sample data
**Input:** influenceDistance, the influence distance of the rivers
**Input:** basinDepth, the basin depth
**Output:** $height, w_{dist}, basinDepth, riverWidth, inRiver,$
**Output:** $inRiverDepth, w_{river}, dist, relevant$

```
procedure CALCULATEHEIGHTBYSAMPLE(v, sample)
    p(t) = sample.location
    p'(t) = sample.derivative
    dist = distance(v, p(t))
    if dist ≤ influenceDistance then
        relevant = true
        halfWidth = sample.width * 0.5
        n(t) = (p'(t).y, −p'(t).x)
        v_n(t) = getProjectedPointOnLine(p(t), n(t), v)
        projectionDistance = distance(v_n(t), v)
        normalDistance = distance(p(t), v_n(t))
        w_dist = 1 - dist / influenceDistance
        height = w_dist * sample.height
        basinDepth = w_dist * basinDepth
        riverWidth = w_dist * halfWidth
        if projectionDistance ≤ halfWidth AND normalDistance ≤ halfWidth then
            inRiver = true
            normalFactor = normalDistance/halfWidth
            factor = normalFactor * normalFactor * normalFactor
            w_river = 1 − projectionDistance/halfWidth
            inRiverDepth = w_river * factor * basinDepth
        end if
        side = pointOnWhichSideOfLine(p(t), p(t) + p'(t), v)
        dist = side > 0 ? -dist : dist
    end if
end procedure
```

accumulating all the calculated data and store them accordingly. How this is done can be seen in Algorithm 4. One important thing to note is that we also store the smallest distance between the vertex and all relevant sample points. This is used later in the final height calculation for the vertex.

The most expensive step of the algorithm are the three distance calculations which are performed for each vertex and each point within the influence distance. While in other applications the computational impact of these calculations can be mitigated by switching to the squared distance which eliminates the necessity to calculate the root of the distance, this is used in many ordering schemes and provides a good speedup, but we need the exact distances for our calculation so we have to use the more expensive operation.

**Procedure 4** Per vertex algorithm for all samples.

**Input:** v, the vertex
**Input:** samples, all sample points influencing the vertex
**Output:** $heightSum, weightSum, basinSum, widthSum, inRiver,$
**Output:** $inRiverSum, inRiverWSum, minDistance$

```
procedure ACCUMULATEHEIGHT(v, samples)
    for all sample in samples do
        returnValues = calculateHeightBySample(v, sample)
        if returnValues.relevant then
            heightSum += returnValues.height
            weightSum += returnValues.w_dist
            basinSum += returnValues.basinDepth
            widthSum += returnValues.riverWidth
            if returnValues.inRiver then
                inRiver = true
                inRiverSum += returnValues.inRiverDepth
                inRiverWSum += returnValues.w_river
            end if
            if abs(returnValues.dist) < abs(minDistance) then
                minDistance = returnValues.dist
            end if
        end if
    end for
end procedure
```

## 6.6 Combine Height Data

At this stage in our approach, we have all the accumulated data from all relevant sample points for a given vertex. We will now show how to calculate the final height of this vertex, as described by all the sample points of the river which influence it. The accumulated height data we collected gives us the height of the river when divided by the accumulated weight value. The value we accumulated here is the height of the middle of the river basin. If we were to set this height for the vertex, it would be below the final river height, thus, in a realistic scenario, the water would cover the vertex even if it were not part of the river. Thus, we add the accumulated river basin depth to the calculated height after similarly being divided by the accumulated weight. By adding these two values together, we ensure that vertices outside the river are always at a higher elevation than the river basin itself, thus ensuring that no water would be able to leave the river basin. We also introduce a linear interpolation factor, which is used later during the rendering process to merge the precalculated terrain with the terrain we calculated during this step. This factor is proportional to the distance between the vertex and the distance to the nearest sample point. How this value will be used will be shown later in Section 8.3.

If the vertex is part of the river, however, we do not use the accumulated depth, but rather the depth we specially stored in an additional accumulator. To calculate the correct

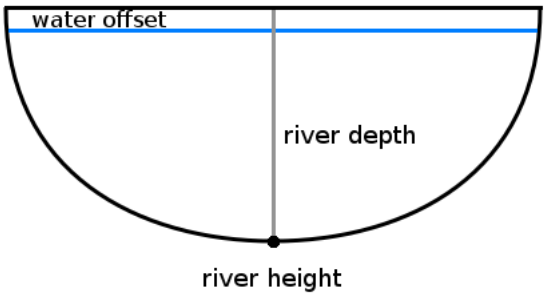



**Figure 6.6:** This figure shows the heights calculated in our approach and how they are combined to create the river basin heights.

height for these vertices, we add the calculated height to this accumulated depth after dividing the accumulated depth by the second weight we stored for these kind of vertices. An illustration of these height calculations can be found in Figure 6.6. Additionally, we also calculate the water height for vertices inside the river basin. This is done by simply calculating the height as for vertices outside the river basin and subtracting the river basin water offset defined in the configuration file. Lastly, we pass the calculated height value, the linear interpolation factor, the water height, the minimum distance and the calculated river width, after its accumulator has also been divided by the weight, to the next step of our approach, which is rendering. Why we need all these values for rendering will be shown later in Sections 8.3 and 8.4. This part of our approach can be found in Algorithm 5.

**Procedure 5** Per vertex algorithm to determine height.

**Input:** v, the vertex
**Input:** samples, all sample points influencing the vertex
**Input:** influenceDistance, the influence distance of the rivers
**Input:** seaLevel, the height of sea level
**Input:** riverWaterOffset, the river basin water offset
**Output:** $terrainHeight, lerpFactor, waterHeight, minDistance, riverWidth$

```
procedure CALCULATEHEIGHT(v, samples)
    returnValues = accumulateHeight(v, samples)
    minDistance = returnValues.minDistance
    lerpFactor = 0
    height = returnValues.heightSum / returnValues.weightSum
    basin = returnValues.basinSum / returnValues.weightSum
    riverWidth = returnValues.widthSum / returnValues.weightSum
    if returnValues.inRiver then
        inRiverBasin = returnValues.inRiverSum / returnValues.inRiverWSum
        terrainHeight = height + inRiverBasin
        heightOffset = basin - riverWaterOffset
        waterHeight = max(seaLevel, height + heightOffset)
    else
        terrainHeight = height + basin
        divisor = (influenceDistance - riverWidth
        lerpFactor = (abs(minDistance) - riverWidth) / divisor
        waterHeight = max(seaLevel, terrainHeight - riverWaterOffset)
    end if
end procedure
```

# Chapter 7

# Using the GPU

In this chapter we will talk about how our approach can benefit from taking advantage of the many core architecture present on modern day graphics cards and how our method can be adapted to increase in performance.

## 7.1 CPU vs. GPU

The central processing unit (CPU) is the core of every phone, tablet, computer, etc. It usually features only a small number of computation cores, each of them being very powerful and having a large instruction set, which can handle any task one can program them to perform. Of course there are very expensive enterprise versions of CPUs, which feature a very large amount of computation cores, but these are usually not affordable for a general user. In contrast to the few very powerful cores of the CPU, the graphics processing unit (GPU) features a very large number of computational cores which are, individually, slower than a computational core of a CPU. The more computational cores a processing unit has, the more instructions can be performed in parallel. By creating a task which can be easily run on multiple cores, utilizing the GPU can yield large performance increases over using the CPU. An easy example of how multiple computational cores can increase performance of an algorithm can be shown by looking at a summation algorithm. On the CPU, the sum of a number of values would usually be computed by iterating over every value and accumulating the values until we have reached the total sum of those values. It is quite easy to see that we need a total of $n$ additions, where $n$ is the number of values to be summed up. On the CPU this would need $n$ time steps to perform this summation. While we still need the same number of additions on the GPU, we can arrange them in parallel to reduce the overall time consumption of the algorithm. An illustration of how such a summation algorithm could be implemented on the GPU can be seen in Figure 7.1.

One factor to consider when adapting an algorithm for the GPU is the type of computations performed on the GPU. As most computations in our approach use floating point arithmetic, the speed at which these operations are conducted has a large influence on the overall speed of our approach. Fortunately, GPUs are very adapted to handle floating point arithmetic and thus fit into our approach quite nicely.

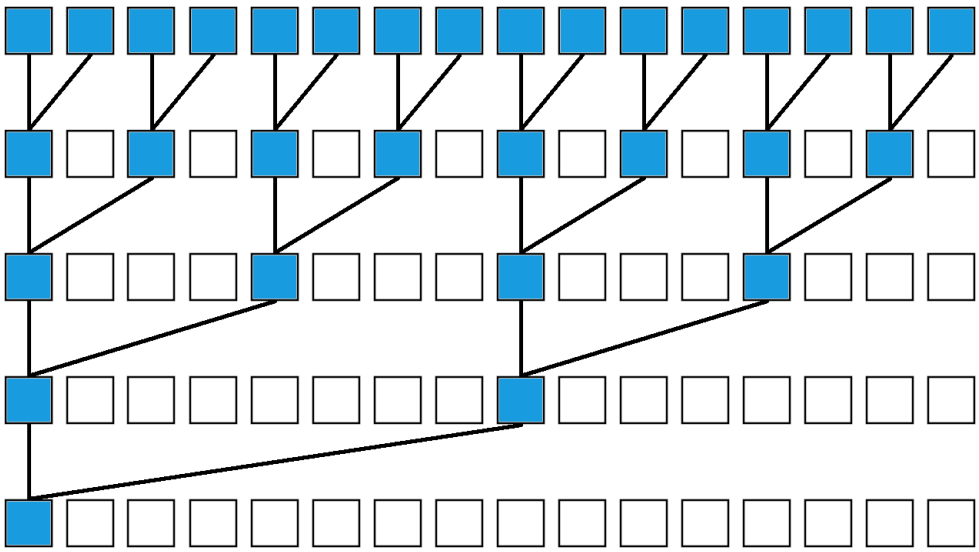


**Figure 7.1:** An illustration of how summation can be performed in parallel.

Careful study of the algorithms discussed in Chapter 6 shows that lots of operations are performed on the same sample data. Each vertex in a patch performs the same calculations on the same sample data. By utilizing this spatial relationship, we can increase the performance of the algorithms again if more worker threads can operate on the same data concurrently. This is exactly the application GPUs were designed for. One additional consideration when performing computations on the GPU is the fact that data does not have to be moved into the GPU memory after its creation, but can already be directly accessed for rendering or other purposes, thus eliminating the time needed for this transfer of data. This gives us another nice increase in performance over a CPU based algorithm with very little complexity.

## 7.2 CUDA

Just as one needs a compiler which translates readable code into instructions for the CPU, we need a toolkit which allows us to create code runnable on a GPU. For NVIDIA GPUs, this toolkit is called *CUDA*. As a general purpose parallel computing platform it contains all the necessary functionality to write code runnable on an NVIDIA GPU. We will only briefly introduce the most important CUDA features used in this work and refer to the CUDA C++ Programming Guide [8] for further reading.

Programs we want to run in CUDA on the GPU can be written similarly to a C++ function, which are called *kernels*. Each kernel is executed by multiple threads in parallel. The number of threads operating on a single kernel is defined when calling each particular kernel. Each thread which executes a kernel has a specific *thread ID*, which makes it possible for the thread to know on which data to operate on. For example, in the example of the summation algorithm above, this identifier would be used to find out if each of the sixteen threads has to perform an addition in the current loop and where to find the

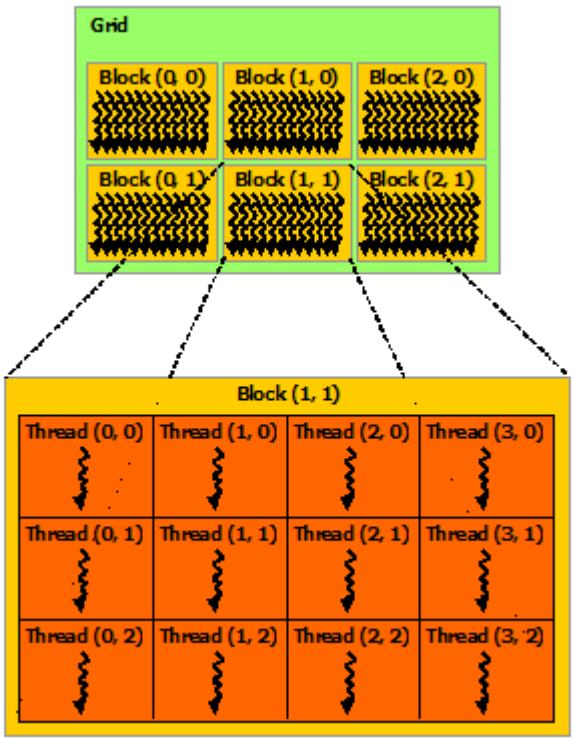



**Figure 7.2:** A grid of blocks and their threads. Image taken from [8].

necessary data. In CUDA, these threads are also grouped together into something called a *block*. A block can currently contain a maximum of 1024 threads, and all of these threads are present on the same computational core and share its memory resources. The threads inside a block can be either arranged one-, two-, or three-dimensional, whichever suits the application best. Additionally, the blocks themselves can also be arranged in any of these three configurations. Each block has similarly a *block ID*, which allows it to find the appropriate data to perform the necessary operations on. It is the responsibility of the user to subdivide the task at hand into the appropriate number of blocks and threads within these blocks to achieve the best performance for each individual application. An illustration of this thread and block hierarchy can be seen in Figure 7.2. The fact that each block can operate independently of any other block shows that the more blocks we can operate on in parallel, the faster all our blocks will be completed. Each block will be assigned to a *Streaming Multiprocessor* (SM), which will run the kernel for that particular block. The more SMs are present on the GPU, the fast the execution of the kernel will be finished. How more SMs help finish a task faster can be seen in Figure 7.3.

One last thing we need to talk about concerning CUDA is the memory hierarchy present in the application. Each thread has a local memory, which is not shared with any other thread. Each block has shared memory, which is available for all threads in the block, and lastly, there is the global memory, which is available to all threads, regardless of the block they are assigned to. An illustration of this memory hierarchy can be seen in Figure 7.4. In the most simple of applications, the data to operate on is usually just

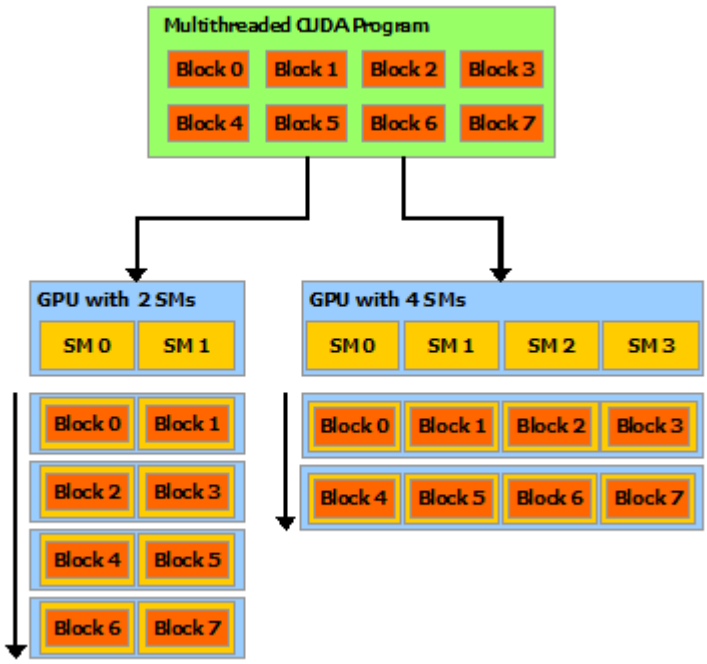



**Figure 7.3:** The same number of blocks assigned to different numbers of Streaming Multiprocessors. Image taken from [8].

stored to global memory, and each thread fetches the data it needs, performs the necessary operations and then saves the result similarly in global memory again. Global memory access is, unfortunately, rather slow when compared to the shared memory. If repeated access of the same data is needed by multiple threads, an alternative way of handling these memory accesses can be introduced. More on this later in Section 7.3.2.

## 7.3 GPU Algorithm

In this section we will discuss how the algorithm can be adapted to work with CUDA and how we can take advantage of the memory hierarchy and shared memory to get an improved version which introduces a nice speedup.

### 7.3.1 Simple GPU Version

The CPU version of the algorithms we discussed in Chapter 6, iterate over all patches. For each of these patches, it iterates over all vertices and for each of these vertices, it iterates over all the sample data present for this particular patch. This version of our approach is very slow, as no part of it has been parallelized. As each patch is independent of any other patch, this is the first hierarchy which can be parallelized. Similarly, each of the vertices in a patch can perform its operations in parallel, as no data has to be exchanged during the operation, as long as the sample data is available to all threads. In the simple version of the GPU algorithm, we simply port the same CPU algorithm to the GPU by making all

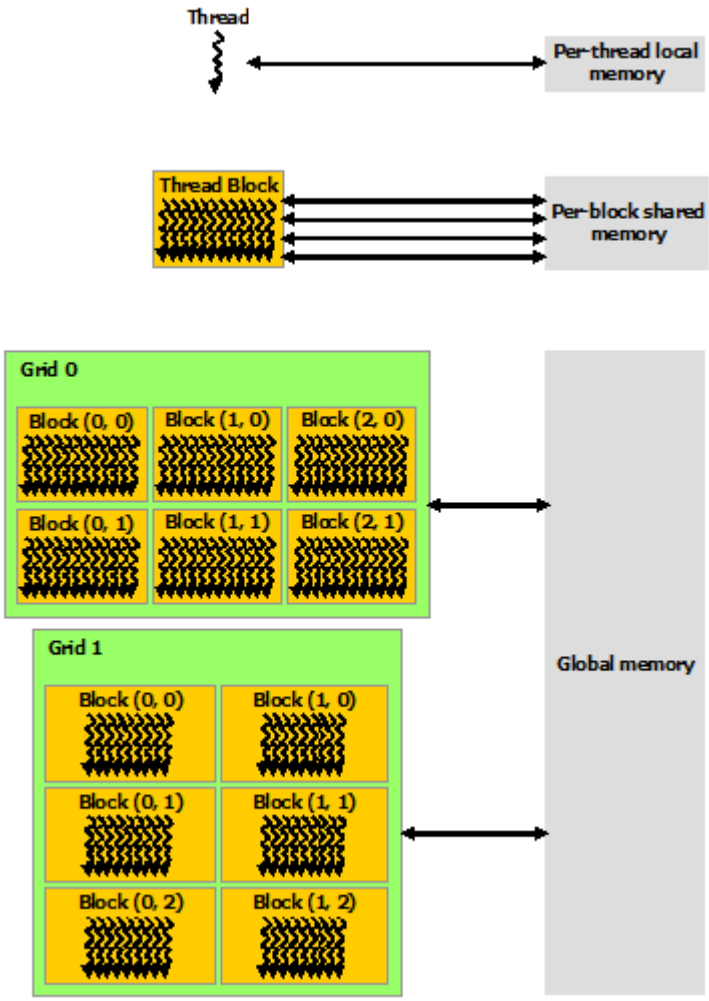



**Figure 7.4:** CUDAs memory hierarchy. Image taken from [8].

the necessary modifications for it to run on the GPU. No performance optimizations are done. We simply subdivide each patch into a number of blocks, each with the maximum number of threads (1024), and each thread calculates the height of the vertex its thread ID signifies. We simply use one-dimensional thread and block IDs. This version is already quite a lot faster than the CPU version, but there are still ways to increase performance. More on the exact performance data later in Chapter 9.

### 7.3.2 Tiled GPU Version

As previously mentioned, CUDA offers multiple memory spaces in its memory hierarchy to the user. The sample data lies in the slow global memory and each thread accesses each sample exactly once, performs the necessary computations and then moves on to the next sample. This fact can be taken advantage of by using a *tiled* algorithm. In a tiled algorithm, the data is subdivided into distinct tiles and the operations are performed on these

tiles iteratively. What this means is that before each thread performs its computations, it first store one sample in the shared memory. As each thread in a block needs to fetch this data from global memory, we have 1024 requests for each sample. By storing it in the shared memory once, and each thread only accessing the shared memory to get the sample data, the latency can be reduced by a considerable amount. These samples currently in the shared memory to be worked on are called a tile. The downside to tiling algorithms is the fact that we now have to be concerned with synchronizing all our threads. If we do not make sure that all samples have been stored in the shared memory, a thread may access data which has not yet been copied from the global memory, which would lead to incorrect results. CUDA has a way to make sure that all threads have reached the same state before computation starts again on any of the threads. This, unfortunately, has an impact on performance, because now some threads may have to wait for other threads to finish before they can resume their computations. Once operations on all threads for the whole tile of data has been finished, the next tile can be loaded and the computation resumes. Of course we need again to make sure that threads do not start loading the next tile of data before all other threads have finished their operations. An illustration of how such a tiled algorithm works can be seen in Figure 7.5. It shows a tiled algorithm applied to the multiplication of two matrices. If the size of the used matrices is exceedingly large, this will help with the computation of the multiplication.

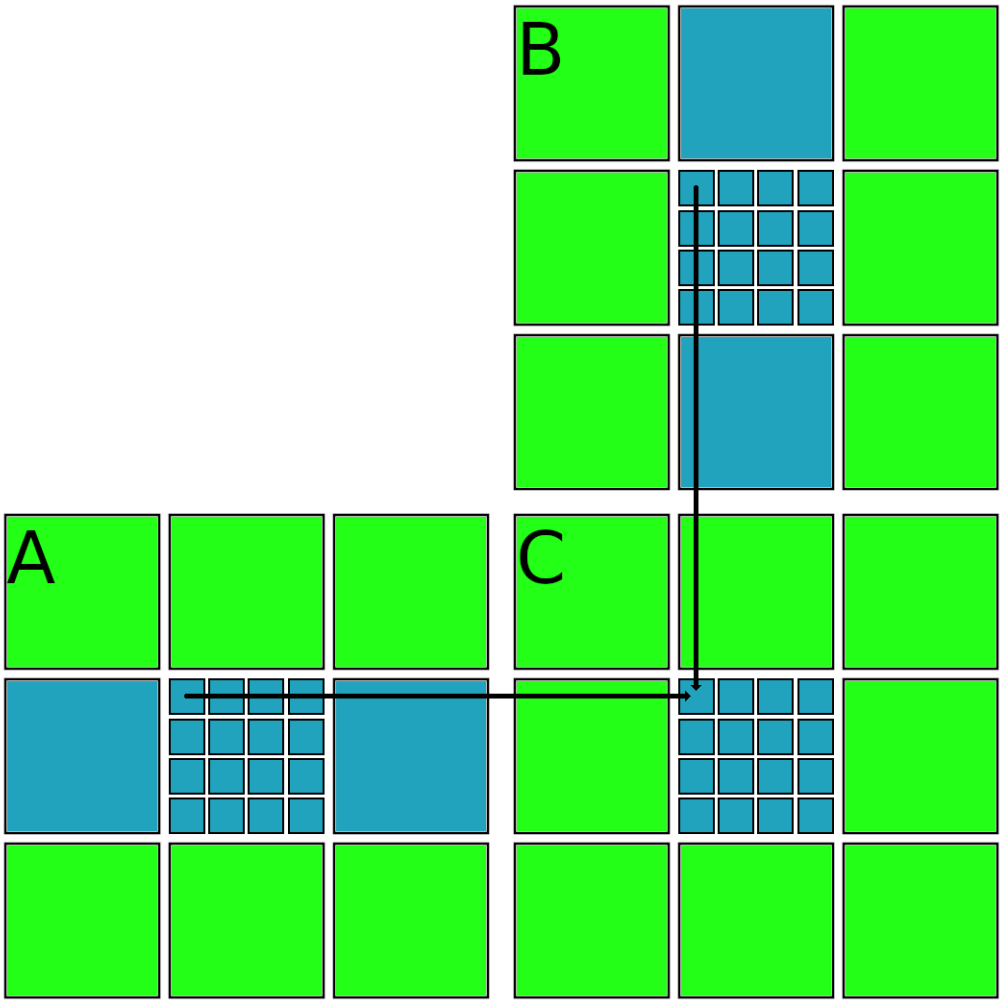



**Figure 7.5:** This image shows the matrix multiplication being done with a tiled algorithm. Each of the large squares represents a single tile. One tile of matrix A and one tile of matrix B are loaded at the same time to calculate the tile of matrix C. The results are then added over all tiles to get the correct values.

# Chapter 8

# Rendering

In this chapter we will talk about the rendering process in general, and at which steps during the rendering process we interfere to achieve our desired results. This contains, but is not limited to, the calculation of the actual vertex position for each vertex of each patch to be displayed, as well as the creation of the water geometry with the data provided to the appropriate shaders.

## 8.1 Rendering Pipeline

The rendering pipeline is the number of steps performed to translate the actual geometry in a scene to the image shown to a user. As there are different software and hardware configurations for such rendering purposes, unifying application programming interfaces (APIs) are created to abstract the hardware, which allows the user to only write the code once and allows that code to run on practically all modern GPUs. As our implementation uses OpenGL for rendering purposes, we will talk about the particulars of the OpenGL rendering pipeline. In Figure 8.1 an overview of the steps present in the rendering pipeline can be seen. We are particularly interested in the shader stages of the pipeline, as these are the programmable steps where we finish our height calculation and water geometry creation.

The responsibility of the *vertex shader* is to perform the transformation of each vertex. They are fed only a single vertex as input, and are expected to only similarly output a single vertex in return. They also receive any number of additional vertex attributes necessary to perform the transformations implemented in the shader. In contrast to the vertex shader, the optional *geometry shader*, takes a primitive as an input. In our case, this primitive is a single triangle. The geometry shader may output any number of triangles, which also includes not outputting a single one. We use the geometry shader to create the water geometry during the rendering process. The last shader we are interested in is the *fragment shader*. Its purpose is to receive a single fragment as input, and also output a single fragment. The fragment shader may also attach color values to the output fragment. For more information on the OpenGL rendering pipeline, we refer to the OpenGL programming guide [36].

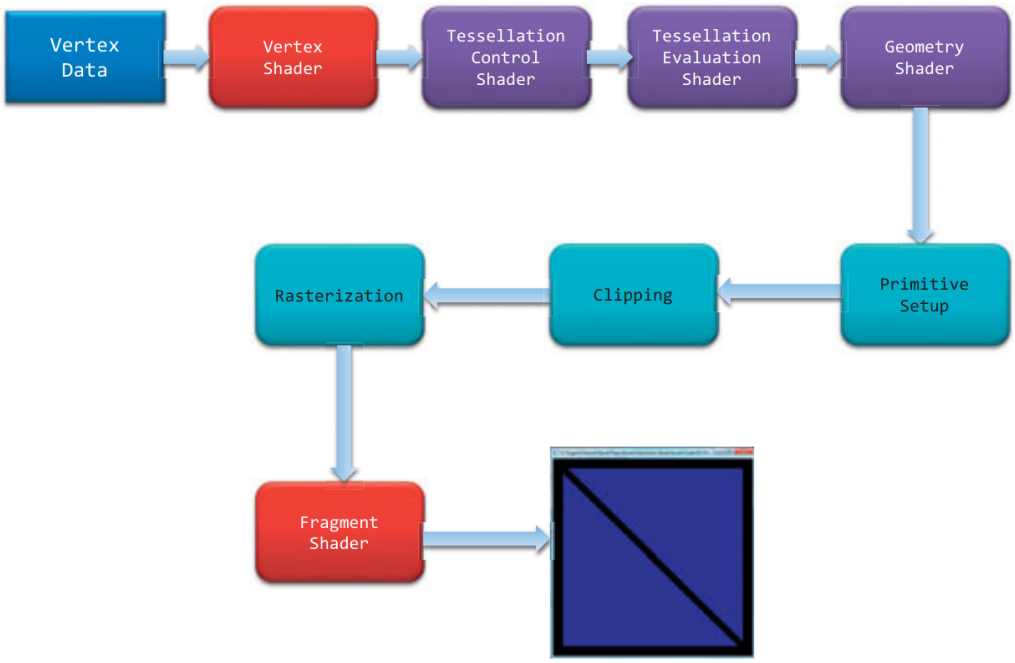



**Figure 8.1:** High-level overview of the OpenGL rendering pipeline. The boxes in red and purple are the programmable shaders. This image has been taken from [36]

## 8.2 Combine Data Using Shaders

In chapter 6 we discussed the creation of the height data for each vertex in a patch. At the end of that chapter, the calculated data has not yet been combined with the original height data which was precomputed. The data calculated by our approach is saved in a texture for each patch and passed to the vertex shader. Additionally, the vertex shader receives the regular mesh we use for rendering, as well as the precomputed height data as determined by the diamond-square algorithm. As the purpose of the vertex shader is to transform a single vertex, the operations we need to perform during its execution are already quite optimized and present to optimal place to perform our necessary calculations. The displacement mapping we are executing in our approach also presents as a vertex transformation, so the vertex shader is the ideal place for this operation. The two different height data textures will be combined to get the correct height for each vertex during the vertex shader execution.

The first step in the vertex shader execution is to determine the precomputed height for this vertex. This is done simply by a lookup in the supplied heightmap. If there is not additional data to be computed from our approach, we move on to normalizing the vertex, scaling it to the supplied radius of the planet, offsetting it to the correct height, and transform it with the appropriate matrices and output that single vertex. In the case that our approach generated height data, we have to calculate the height data differently. We linearly interpolate between the two height values we get from the two texture lookups, with the linear interpolation factor we calculated at the end of the algorithm. This allows

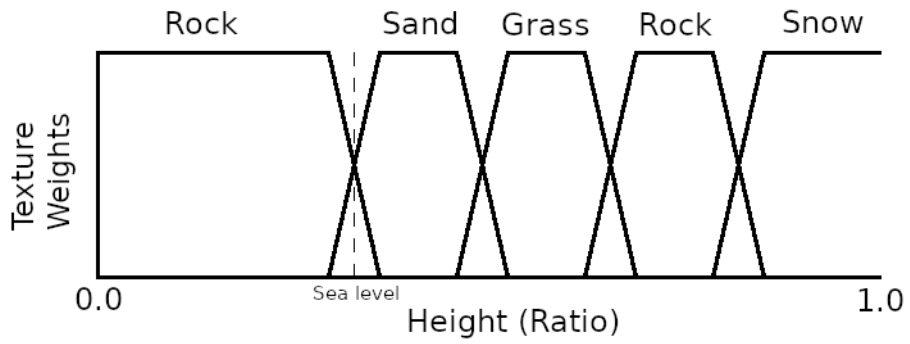



**Figure 8.2:** The weights of the different textures depending on the height.

us to blend our two heightmaps together in quite an easy way. After we have created this combined height data, we again normalize, scale and transform the vertex before outputting this vertex. As the linear interpolation factor is zero for all points situated in the river basin described by the sample data, only the calculated data of our approach is used during this vertex transformation.

## 8.3 Terrain Rendering

At this stage we enter the geometry shader. The geometry shader gets three vertices as input, which form a triangle. For triangles without water we simply output a primitive consisting of the three vertices we received as input. Finally, we arrive at the fragment shader. Here we color the fragments according to their heights. This coloring operation consists of a texture lookup according to each height in either a dirt, grass, rock or snow texture. Additionally, we blend the different textures if the height is in the transition region between the two textures. We do this by assigning weights to the different texture values as can be seen in Figure 8.2. We also take steep slopes into account and change the used texture to the rock texture, if the normal of the fragment sufficiently deviates from the original normal of the vertices. To achieve this, we modify the vertex shader to calculate the normal at each vertex (with the help of its surrounding vertices), and pass it through the geometry shader to the fragment shader. This normal is also later used in lighting calculations performed in the fragment shader. This texturing approach concerning blending and accounting for slopes is done similarly as in the work by Nicholson et al. [28].

There is one important fact we have not yet talked about concerning the rendering of the terrain. This fact concerns the mismatch between the mesh resolution and the size of the rivers. If we were to displace the terrain just as we discussed above at even the coarsest level of detail, we would not be able to see rivers, but rather only single vertices with lower elevations than their surroundings if they coincide with the river location. On finer levels of detail we may end up with terrain bridging the river basin, if an edge connects two vertices which are on opposite sides of the river basin. An example of this can be seen in Figure 8.3. How we remedy part of this situation will be discussed in the next section.

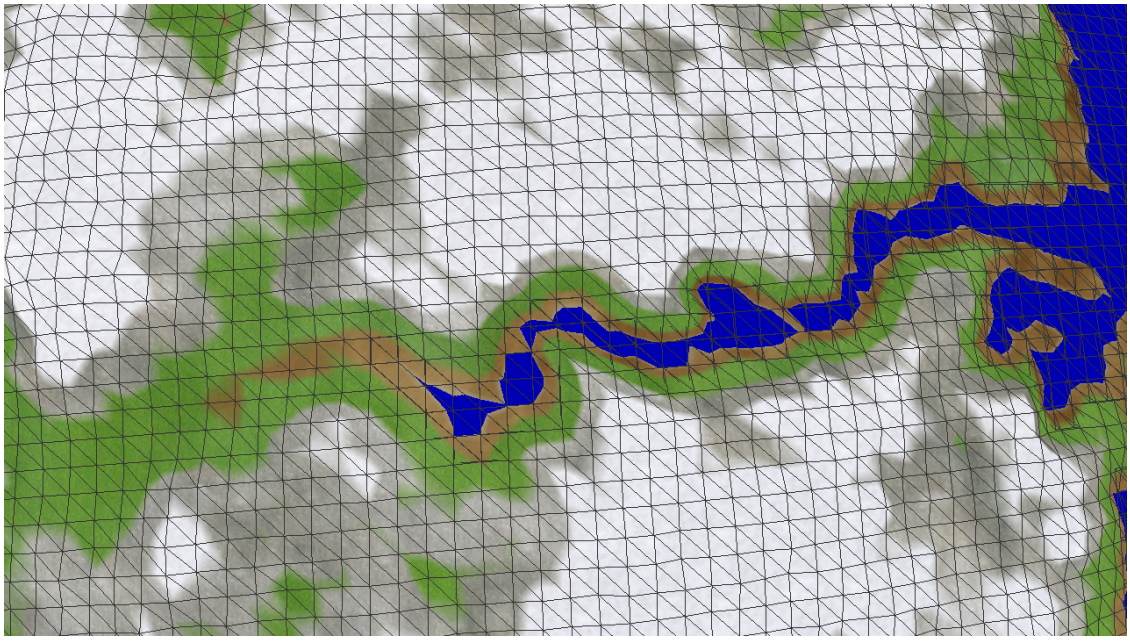


**Figure 8.3:** The displaced heights if patch resolution does not match river resolution.

## 8.4 Water Rendering

As mentioned previously, we only create the water geometry on the geometry shader with the data calculated by our approach and forwarded from the vertex shader to the geometry shader. We spawn an additional triangle with the water height calculated by our approach depending on the data supplied by the vertex shader. If any of the three vertices is inside a river basin, which we can check by examining the minimum distance we saved from our calculation and the river width we have determined in our algorithm, then we spawn a water triangle. Additionally, if any of the heights of the three vertices is below sea level, we also spawn a water triangle. This makes sure that water triangles pierce through the terrain created by our approach and thus cover all the terrain we created previously. In Figure 8.4, the camera is placed inside the planet and it is easy to see the water geometry passing through the terrain to cover the river basin with water.

As discussed previously, we end up with artifacts if our mesh resolution does not match well with the width of the rivers. We have to make sure that the distance between our patch vertices is smaller than half the width of any particular river to make sure that no artifacts can occur during rendering. This constraint, however, can never be satisfied on very coarse levels of detail, as the patch size is much too large. The initial patch covers the whole side of the planet, so any river present on this side of the planet will be smaller by necessity. To remove these artifacts, we are checking in the geometry shader if our patch has a high enough resolution to correctly display the river. If the patch has enough resolution to display our river, simply carry on as before. If the patch has not enough resolution, we paint the river on the terrain in our fragment shader. To achieve this, we pass the previously used distance and river width to our fragment shader. This painting

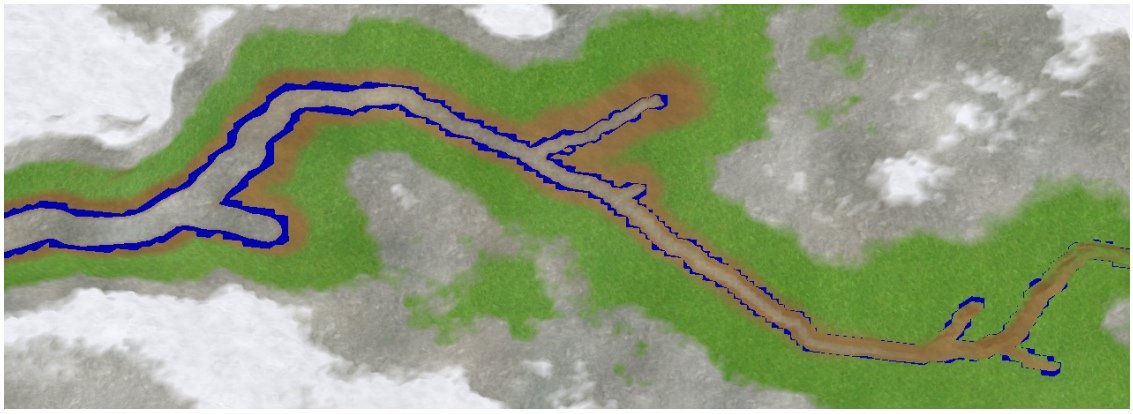


**Figure 8.4:** The camera placed inside the planet looking outwards. The water geometry passing through the terrain to cover all possible holes.

process is also the reason why we do not pass the absolute distance in the algorithm to our shaders, but rather the signed distance, because we would end up with artifacts during the painting process of our fragment. We paint the color of the water onto the fragment when the calculated distance is smaller than the width of the river. This way we get an approximation of the river, which is good enough to hide large artifacts when switching the level of detail. In Figure 8.3 we saw the mismatch the result of the mismatch between the mesh size and the river size, while in Figure 8.5, we can see an example of the painted river with the data we supplied to the fragment shader. Unfortunately, this is not a perfect solution, as, particularly at the start of the river (spring), artifacts from this painting process can occur.

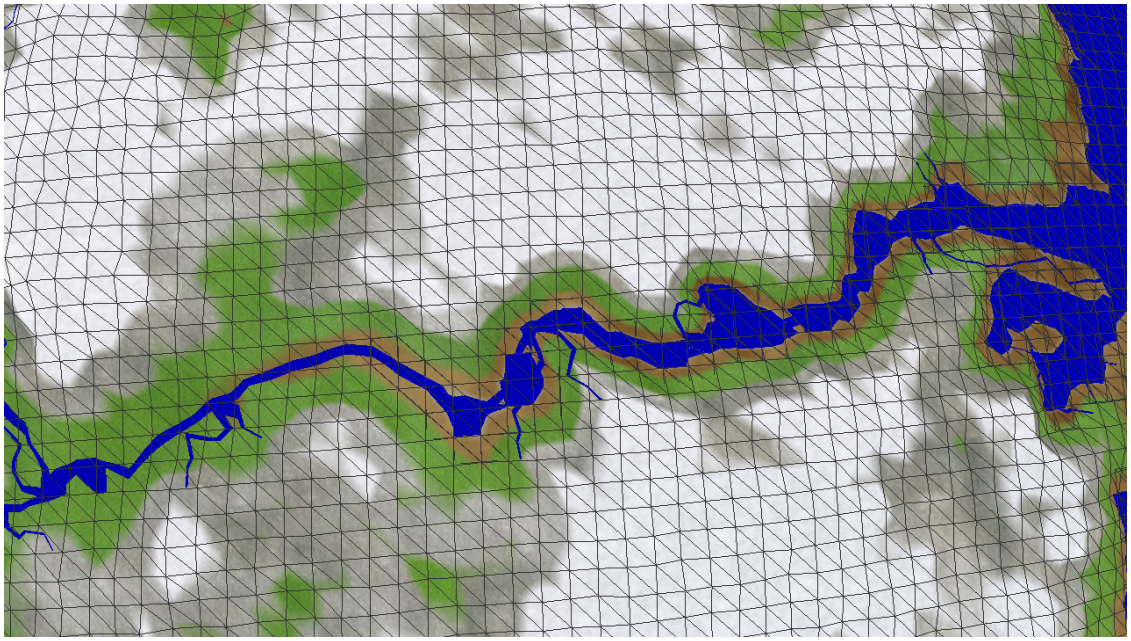


**Figure 8.5:** The river painted in the fragment shader when mesh and river size do not match.

# Part III

# Results

# Chapter 9

# Results

In this chapter we will briefly discuss the performance of our presented algorithm. We will also discuss the performance increase when using a tiled algorithm instead of using the CPU algorithm ported onto the GPU. We will also specify the testing environment, which was used during the evaluation of the algorithm.

## 9.1 Environment Specification

Our test system consists of a 3.5 GHz Intel Core i7-4771 for the CPU side of the algorithm, as well as an 1.5 GHz NVIDIA Titan X to evaluate GPU performance.

## 9.2 Performance Evaluation

The CPU version of the algorithm is implemented strictly single-threaded. It is possible to also implement a multi-threaded version of the algorithm, but the performance increase of such an implementation pales in comparison to the performance gained by switching to the simple GPU-based version. Figure 9.1 shows the difference in performance between the CPU and the GPU version of the algorithm. At the scale present in the chart, the difference between the normal GPU version and the tiled algorithm is not noticeable, which is the reason it has been omitted from the chart. From this chart it is obvious to see that the simple GPU version represents an enormous improvement in performance over the CPU version.

In Figure 9.2 we can see the number of patches (Figure 9.2(a)) and samples (Figure 9.2(b)) for each frame. The number of samples has been divided by 100 in the chart. If on compares the graph created by the CPU performance in Figure 9.1 to the graph created by the samples, one can quite easily see that they are almost identical. This stems from the fact that the number of samples directly influences the runtime of our algorithm on the CPU.

How the tiled algorithm performs in comparison to the simple GPU version, can be seen in Figure 9.3. In this chart we only see the time differences between the actual kernel calls of the different GPU versions, as the setup and cleanup times of the two algorithms perform the same exact code. While the difference in performance is not particularly large,

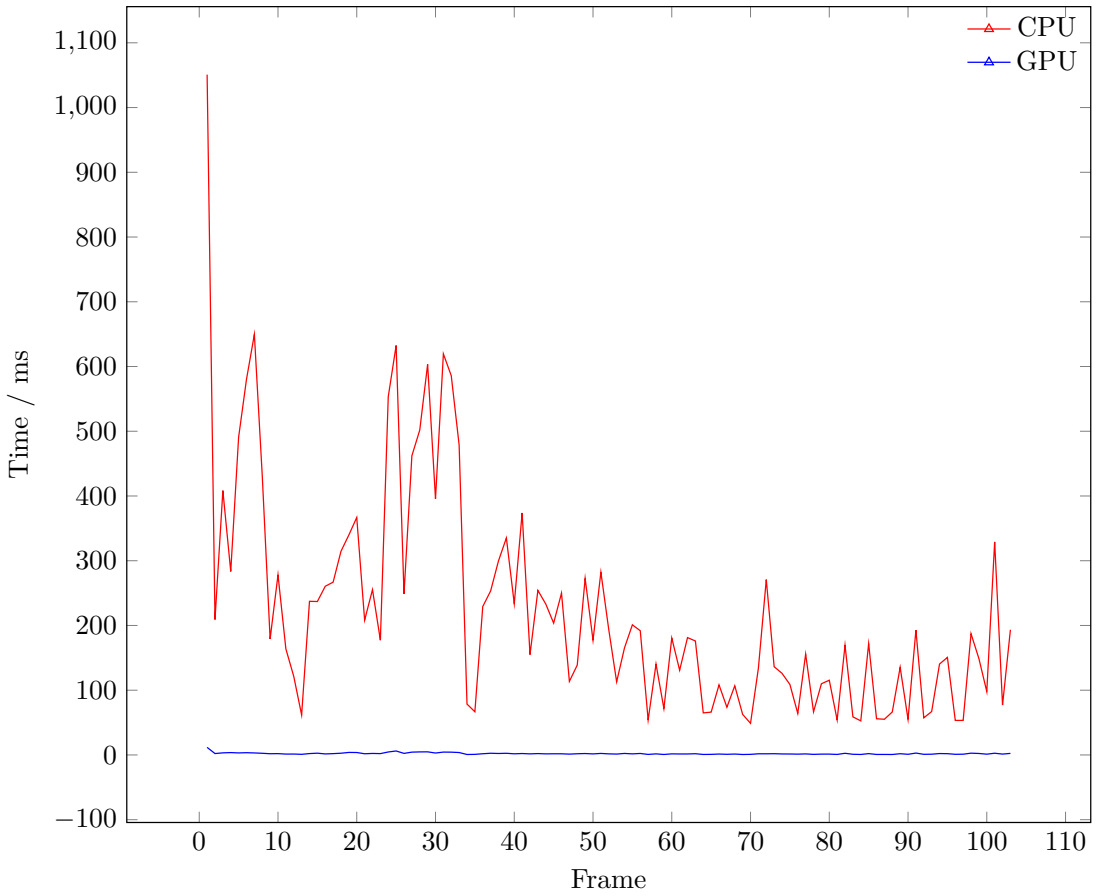
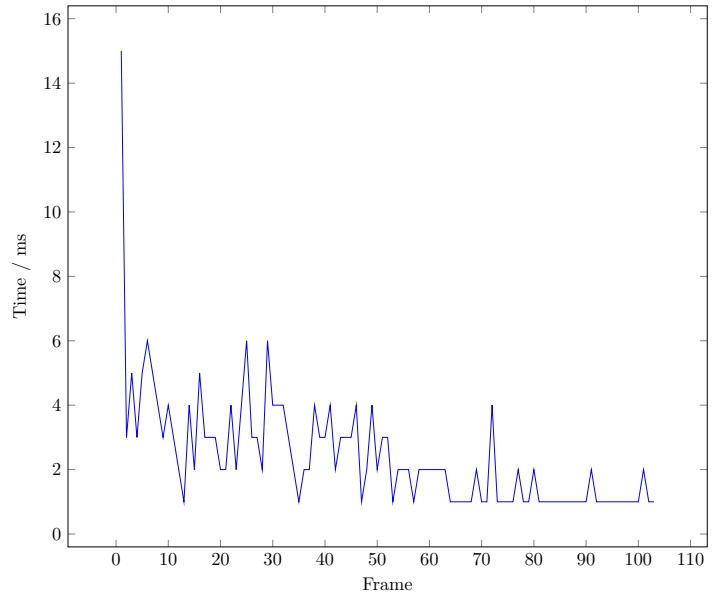



**Figure 9.1:** This chart shows the difference in performance between the CPU version of the algorithm and the GPU version.

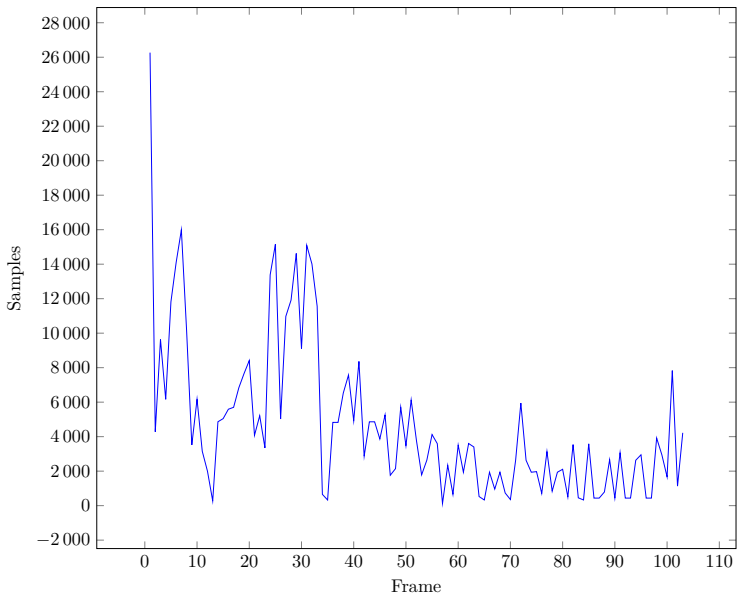
it is nevertheless a measurable increase in performance over the simple GPU version and even a small increase in performance can lead to large increases of frames per second in high-end hardware.

As discussed before, the graph created by the CPU performance matches the graph created by the number of samples. The GPU performance does not usually match the sample graph because of the inherent parallelism of the GPU. If only one patch has to be created, the two graphs would match up, as the number of sample points dictate the execution of the algorithm. In the case that more patches have to be created, the number of sample points gets distributed across all patches, which have to be created, and only the patch with the largest number of sample points is relevant for the overall runtime of the algorithm, provided that all patches can be created on the GPU in parallel.

To investigate the performance of the different algorithms, we look at the average calculation time. In Table 9.1, we have listed the average time spent on the calculation for all different versions we have implemented. From this table we can see that the GPU version is more than 100 times faster than the CPU version, while the tiled version is about a fifth of a millisecond faster than the normal GPU version, which is approximately




(a) This chart shows the number of patches used during the calculation of the new terrain.




(b) This chart shows the number of samples at each frame in the comparison of the different versions of the algorithm.

**Figure 9.2:** Figure showing the number of calculated patches and sample data for the initial performance evaluation.

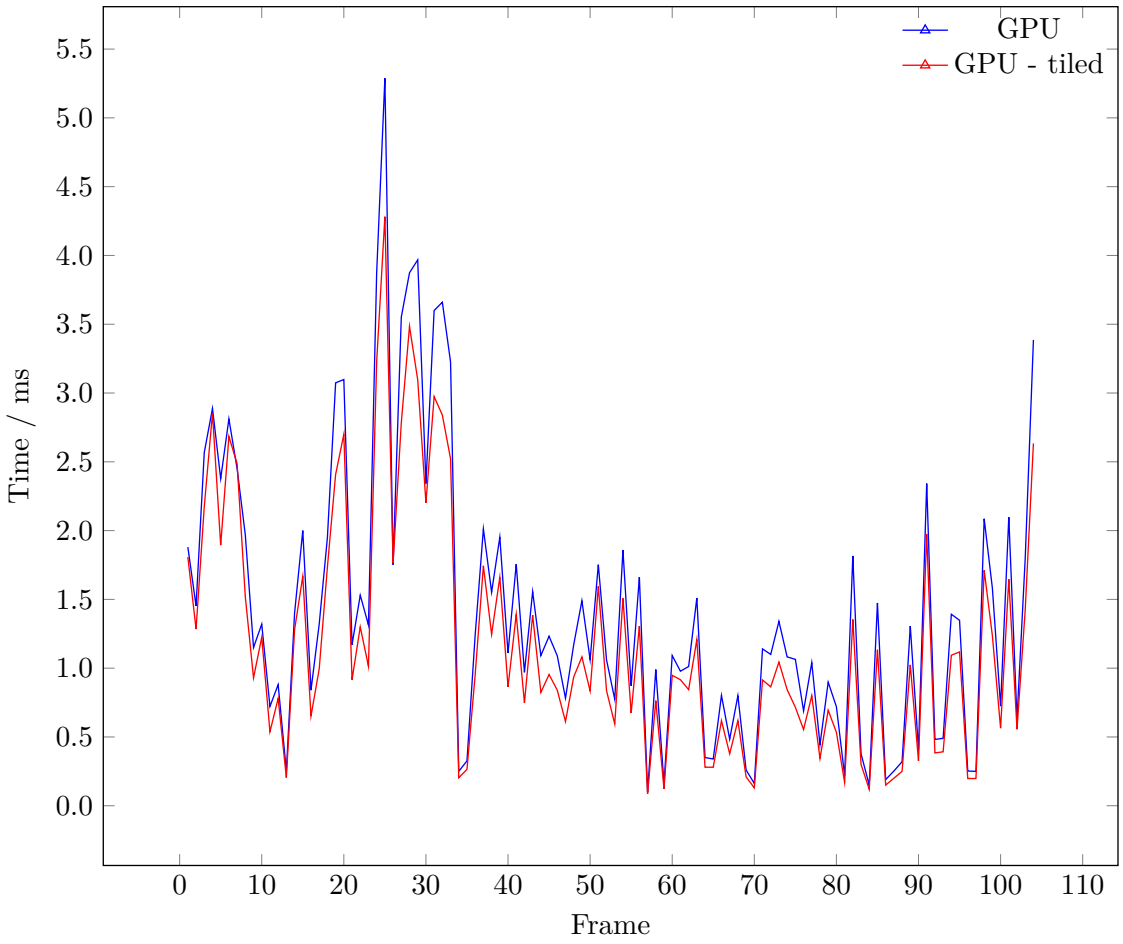



**Figure 9.3:** This chart illustrates the differences in performance between the simple GPU version and the tiled GPU algorithm. Comparison of the actual kernel execution time only.

| | |
|---|---|
| CPU | 234.742 ms |
| GPU | 2.136 ms |
| GPU kernel call | 1.424 ms |
| GPU tiled | 1.921 ms |
| GPU tiled kernel call | 1.174 ms |

**Table 9.1:** The average time spent on calculating the new terrain.

an increase of 10 % in performance.

In Figure 9.4 we take a look at the memory consumption of our approach. The thing we are most interested in is the dynamic texture memory, as the texture memory uses the most amount of memory. What we did not include in the chart is the memory used by our precalculated height data. The reason for this omission is the fact that these textures use 96 MB of data. These precalculated textures contain the height data for all patches in all levels of detail. If we were to calculate this height data during the execution of our

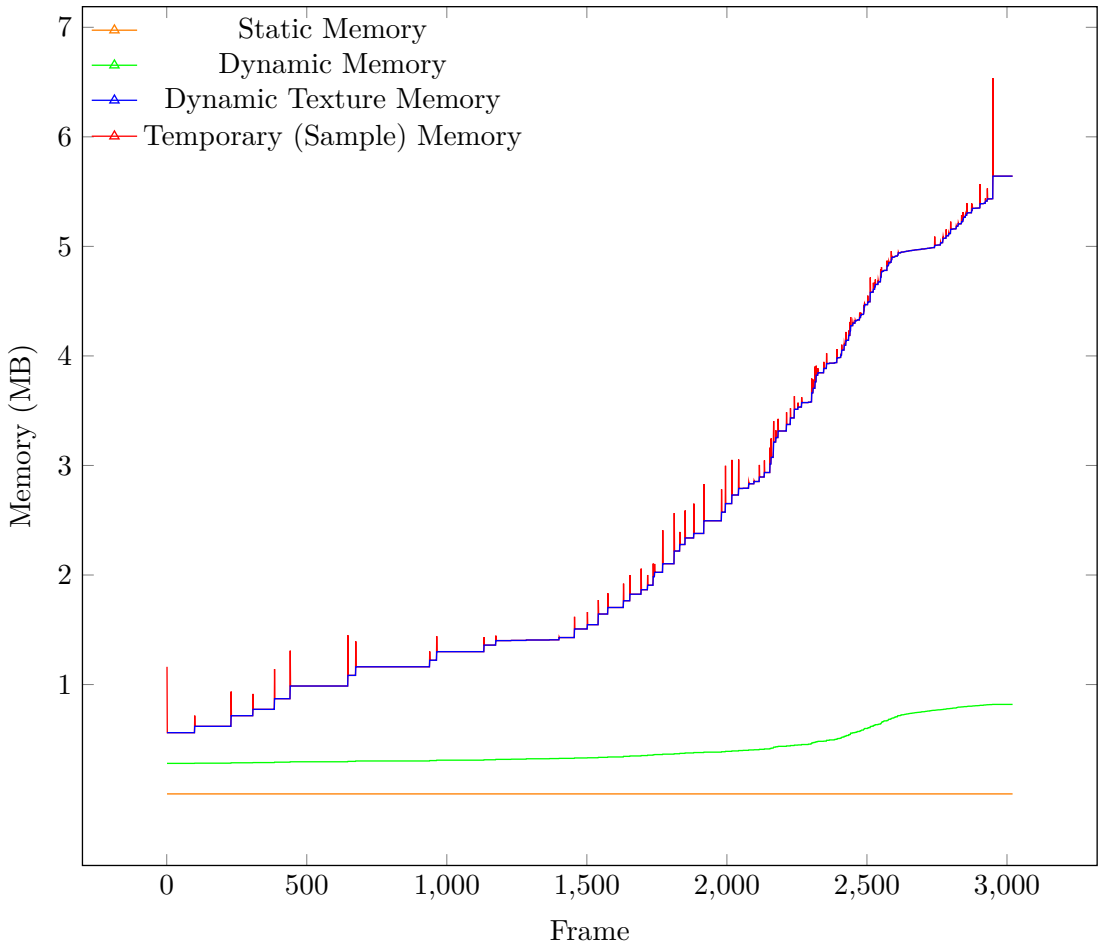



**Figure 9.4:** This chart shows the memory consumption of our approach in megabytes.

program, we would loose some performance, but gain a lot of memory, as the height data would only amount to an additional 25 % of the dynamic texture memory. The reason for that being the fact that we only store a single floating point value in every additional texture in contrast to the four values we store in the textures supplied to our shaders. Because of the slow camera speed, the temporary data for the sample data is only used for one frame and then discarded, which explains the small ticks on the graph.

To more accurately gauge the performance advantages of the tiled algorithm in comparison to the simple GPU version, we perform a stress test. This stress test consists of increasing the distance, which is used to switch to the next level of detail, by a factor of ten, as well as increase the camera speed by a factor of 25. This leads to a dramatic increase in patches to be computed, and as a consequence a large increase in the amount of samples. The number of patches and samples can be seen in Figure 9.5. With this much data to be computed the CPU version can not even approach real-time performance, but the comparison between the CPU version and the simple GPU version can be found in Figure 9.6. The comparison between the tiled and the simple GPU version can be found

| CPU | 8403.027 ms |
|---|---|
| GPU | 7.372 ms |
| GPU kernel call | 5.815 ms |
| GPU tiled | 5.521 ms |
| GPU tiled kernel call | 3.963 ms |

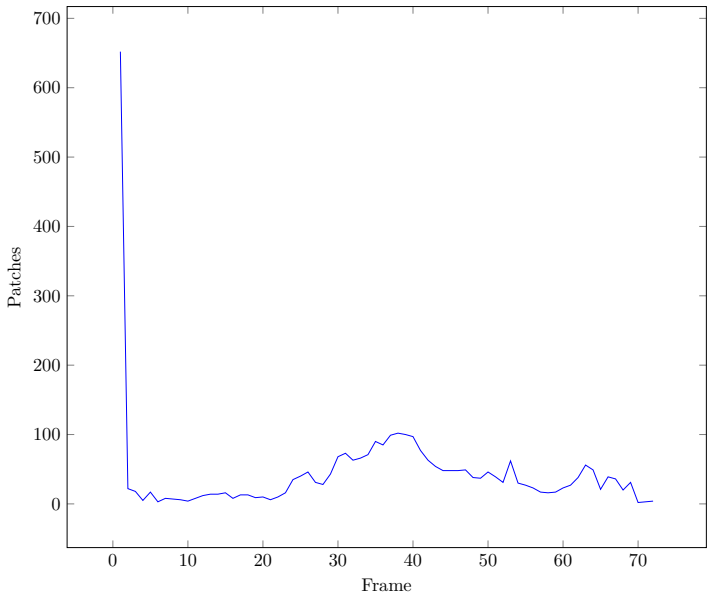
**Table 9.2:** The average time spent on calculating the new terrain during the stress test.

in Figure 9.7. Figure 9.7(a) contains the complete performance data, while Figure 9.7(b) removes the initial value to allow a more detailed look at the later values. What the memory consumption looks like during the stress test can be seen in Figure 9.8. The large increase in patches leads to a matching increase in dynamic memory, as well as a very large increase in the number of dynamic texture memory. Even in this case the switch to per patch heightmaps instead of one large heightmap per cube face would lead to a rather large reduction in memory footprint. During the stress test, the camera speed is increased by a large factor. This leads to some consecutive frames calculating new patches. This is the reason why we can see continuous temporary memory in the graph, in contrast to the usual tick-like appearance of temporary memory consumption in Figure 9.4.

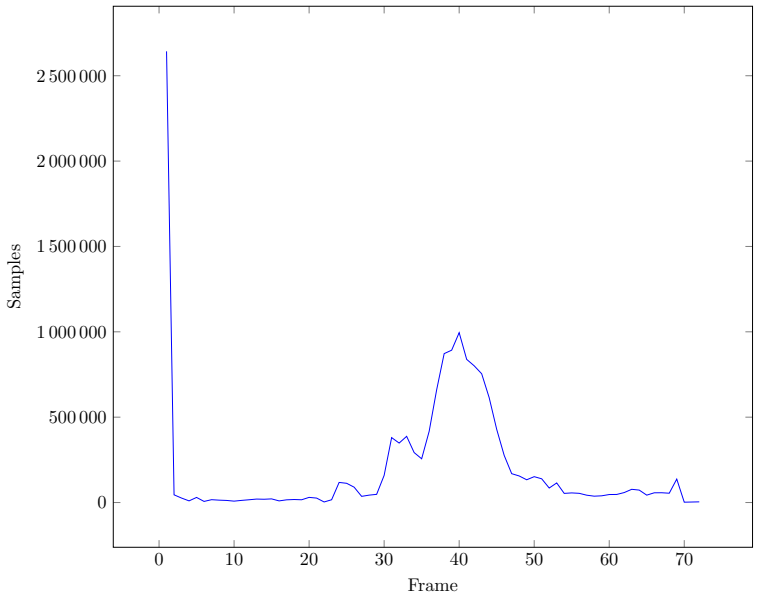
We again investigate the average performance of the different algorithms. The results can be found in Table 9.2. The CPU version is hopelessly outclassed with an average of over 8 seconds. In this stress test the tiled version of the algorithm yields a 25% increase in performance in relation to the simple GPU version.

## 9.3 Visual Results

In this section we present a few screenshots made with our application. In Figure 9.9, we first show the same river in multiple levels of detail. The painted parts of the river are colored red to differentiate the different techniques. Keen observer may see, that in level of detail 2 and 3 the painted river is not completely connected. This is an artifacts, which arises from the way the rivers are painted. In this case, all vertices of the triangles are found to be on the same side of the river, thus no river is painted. They are on the same side, but one vertex is on one side in relation to the original river, while the other is considered part of the tributary. Figure 9.10 shows level of detail 2 at the closest distance, where this disconnect is hardly noticeable. In Figures 9.11, 9.12, 9.13, 9.14 and 9.15, we show more result pictures made with our application. They all show the same generated planet, but different camera positions around the planet.




(a) This chart shows the number of patches used during the calculation of the new terrain of the stress test.




(b) This chart shows the number of samples at each frame in the comparison of the different versions of the algorithm during the stress test.

**Figure 9.5:** Figure showing the number of calculated patches and sample data for the initial performance evaluation.

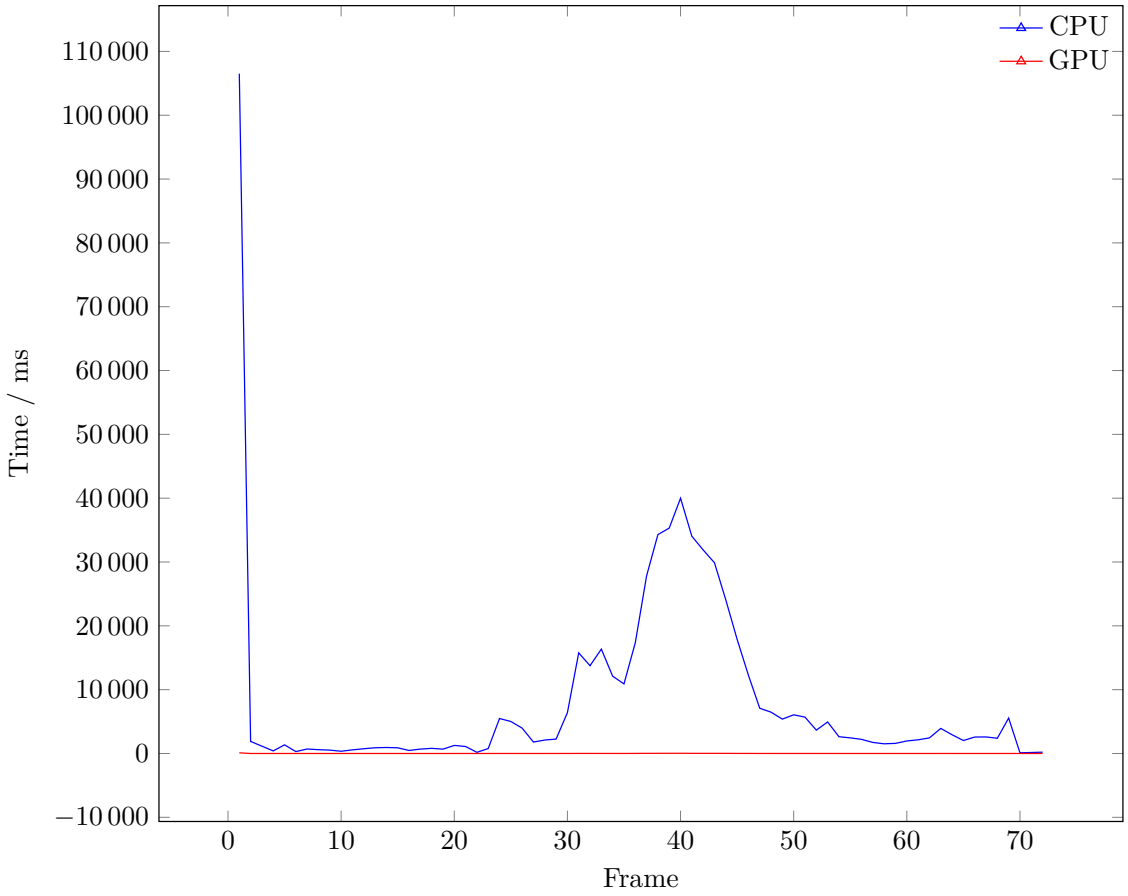
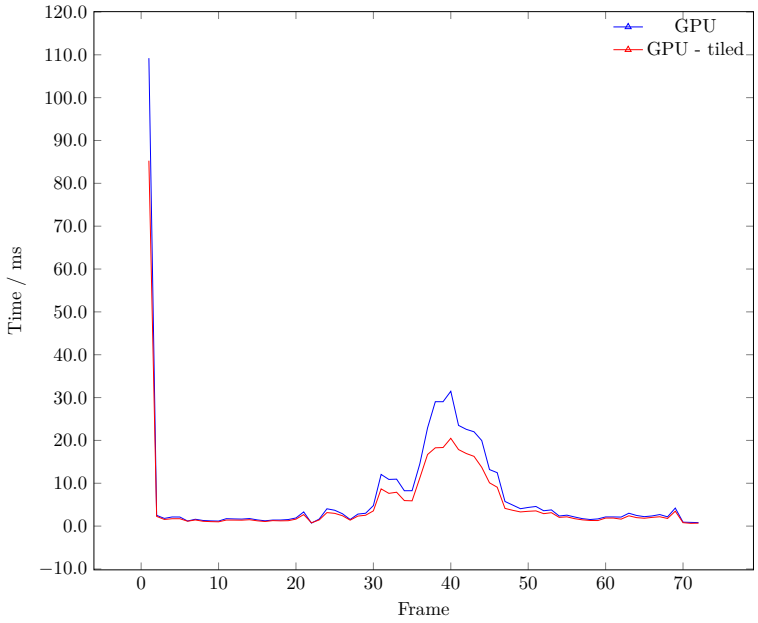
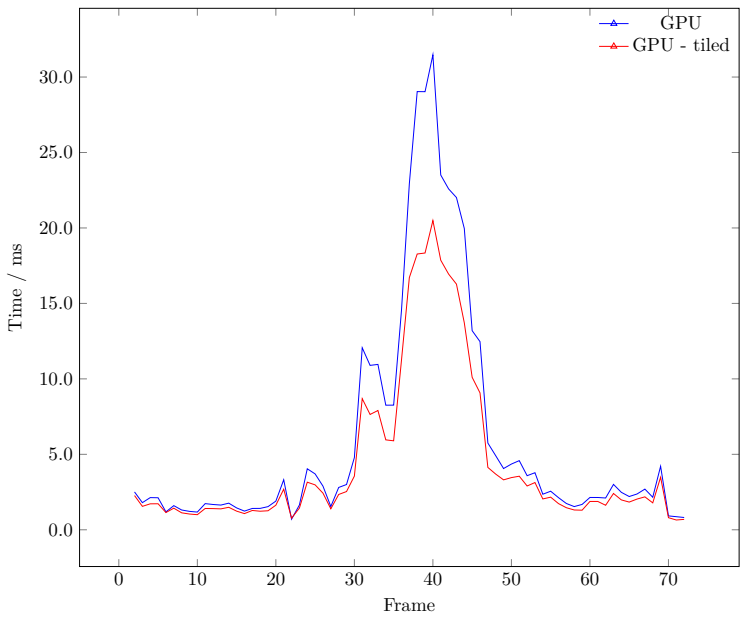



**Figure 9.6:** This chart shows the performance of the CPU and GPU algorithm during the stress test.




(a) This chart illustrates the differences in performance between the simple GPU version and the tiled GPU algorithm during the stress test.




(b) The same data, but the initial value removed to allow a more detailed look at the later values.

**Figure 9.7:** Performance data for the GPU versions during the stress test.

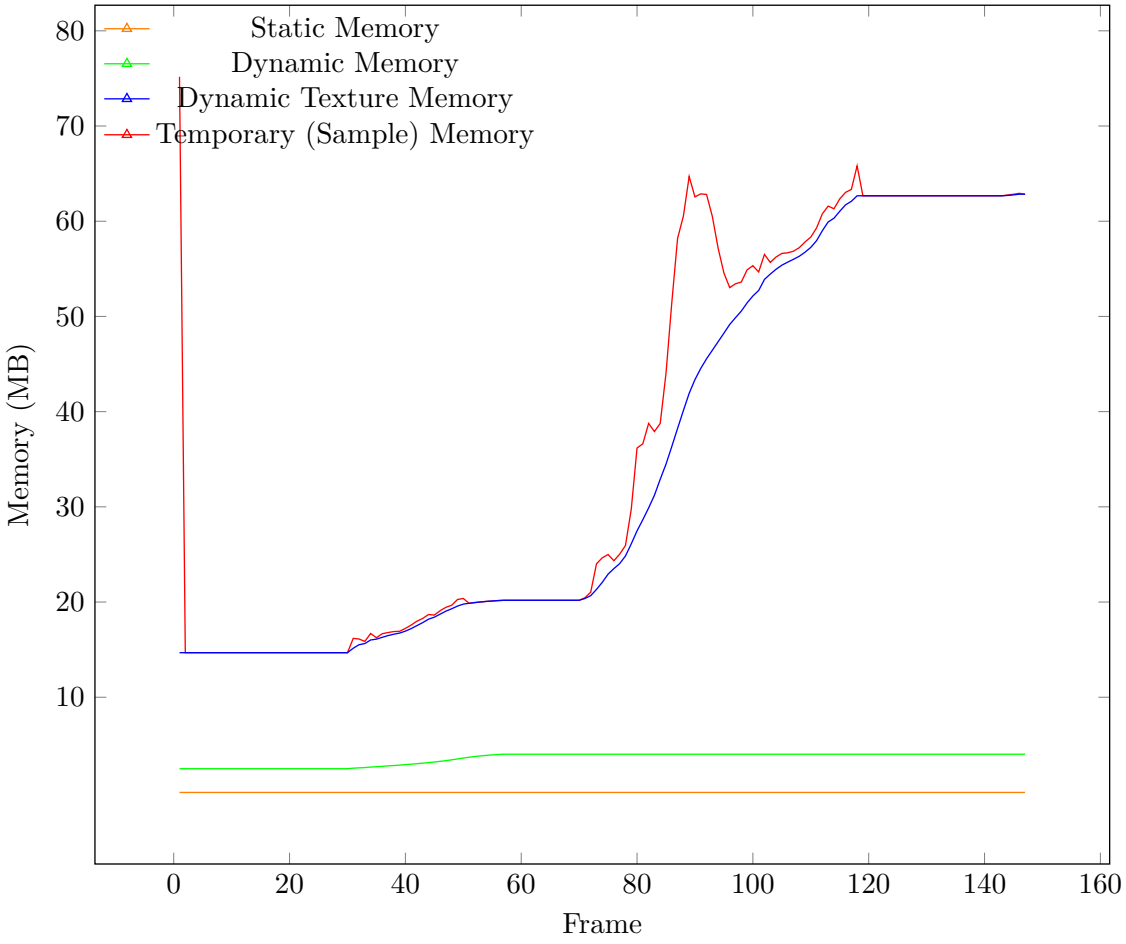
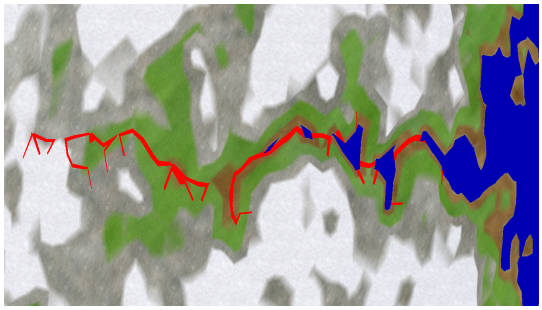
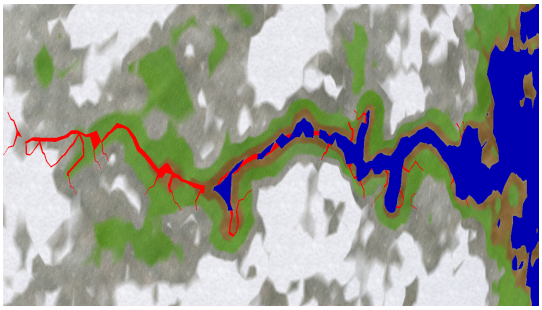


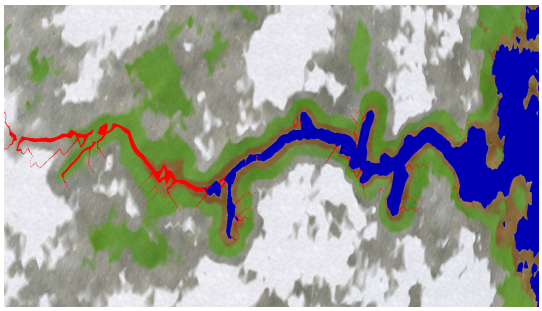

**Figure 9.8:** This chart shows the memory consumption of our approach in megabytes during the stress test.



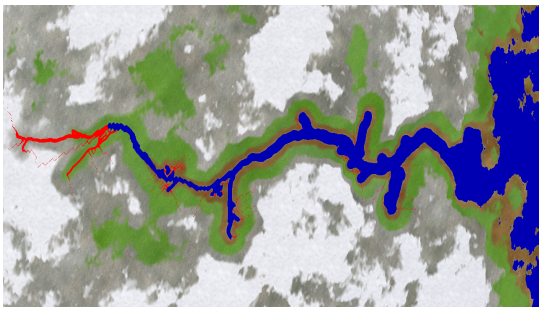
(a) River at level of detail 2.



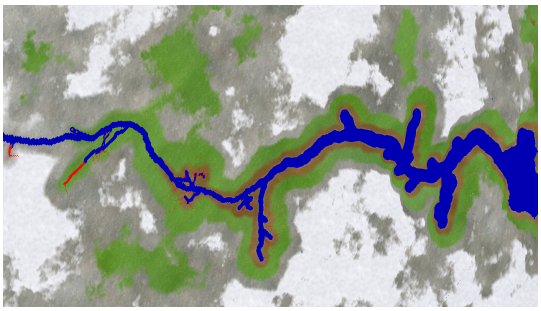
(b) River at level of detail 3.



(c) River at level of detail 4.



(d) River at level of detail 5.



(e) River at level of detail 6.

**Figure 9.9:** The same river shown in multiple levels of detail. The painted sections of the river are colored red. The lowest levels of detail are not shown as the vertices are too far apart to contain any meaningful data.

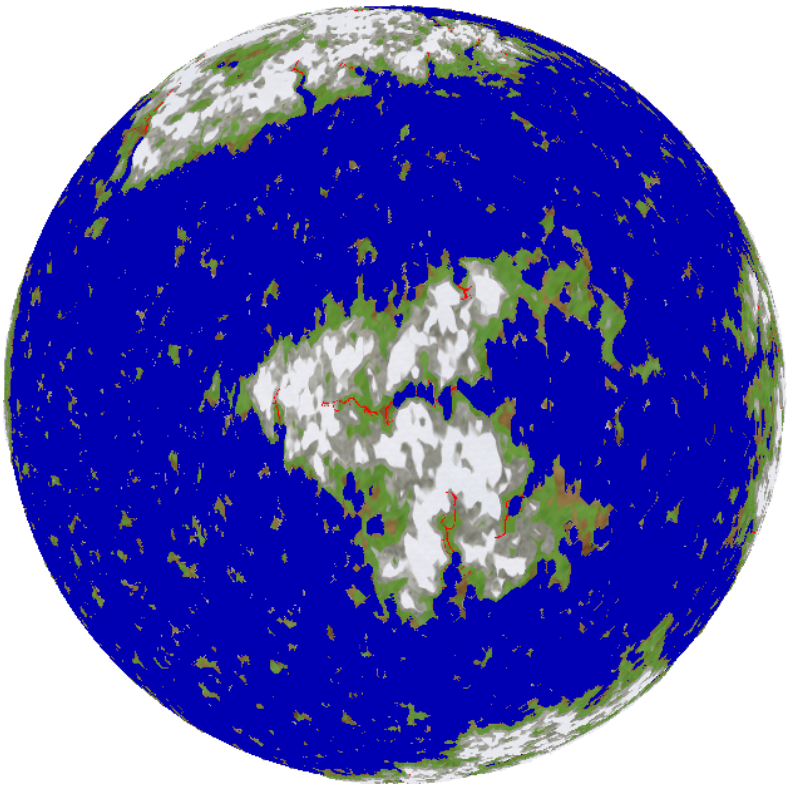


**Figure 9.10:** Level of detail 2 of the example river at the closest distance of this particular level of detail.

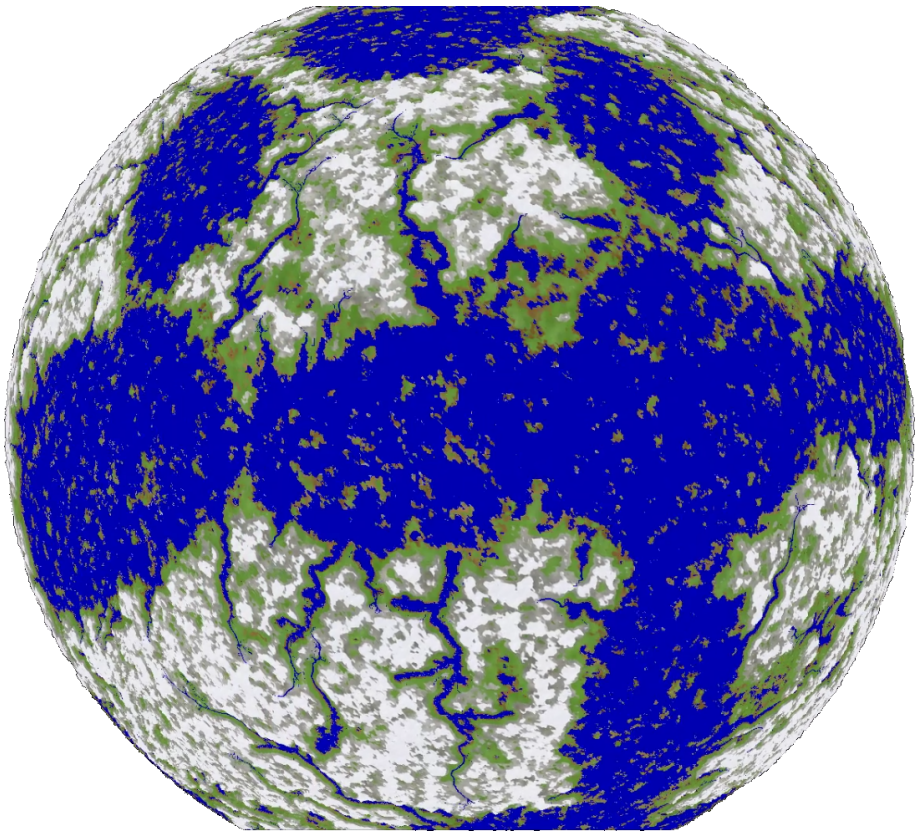


**Figure 9.11:** Screenshot of a generated planet and river network.

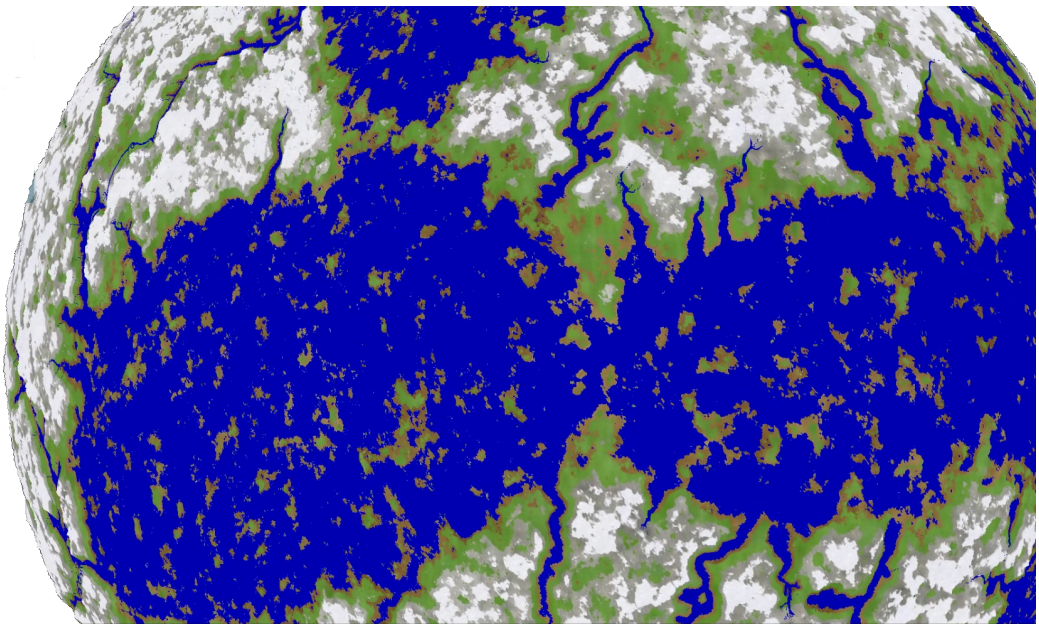


**Figure 9.12:** Screenshot of a generated planet and river network.

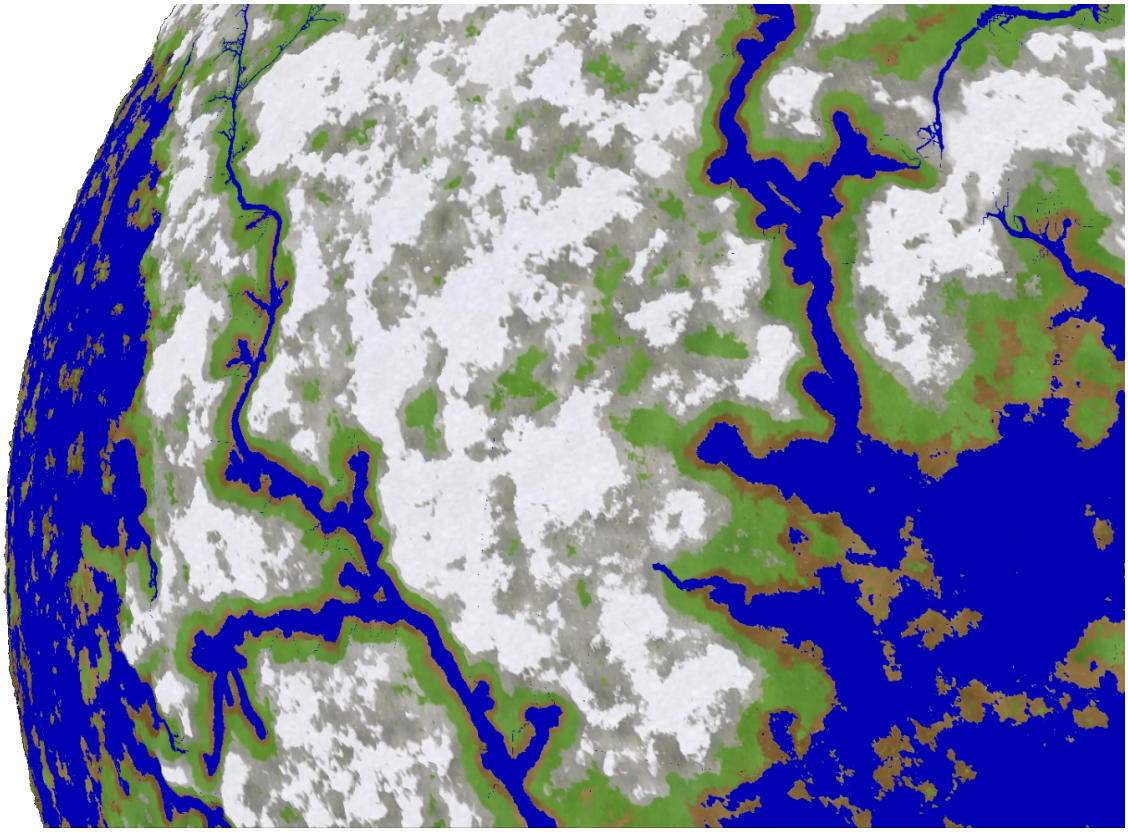


**Figure 9.13:** Screenshot of a generated planet and river network.

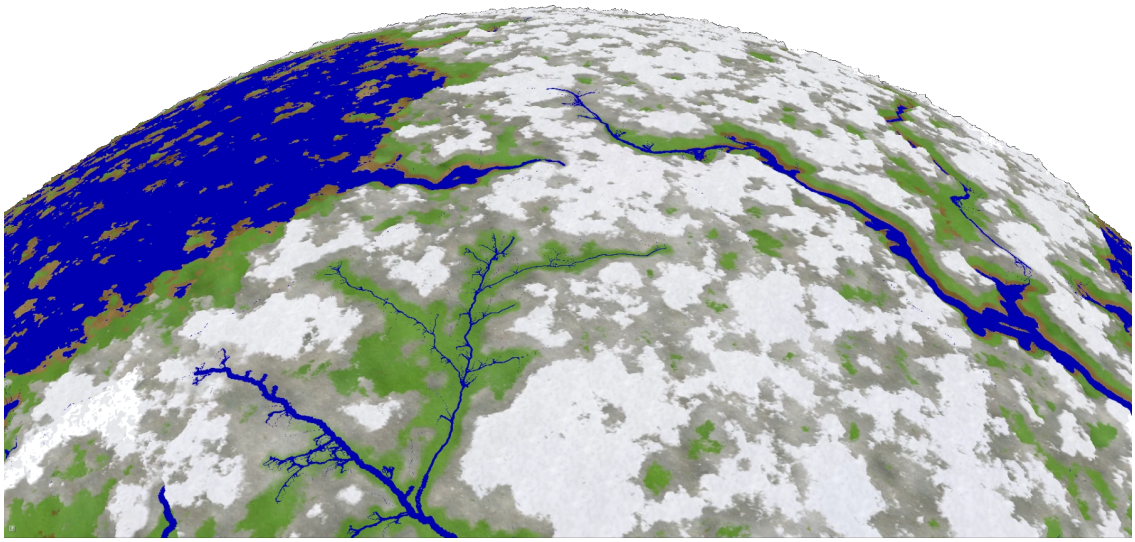


**Figure 9.14:** Screenshot of a generated planet and river network.

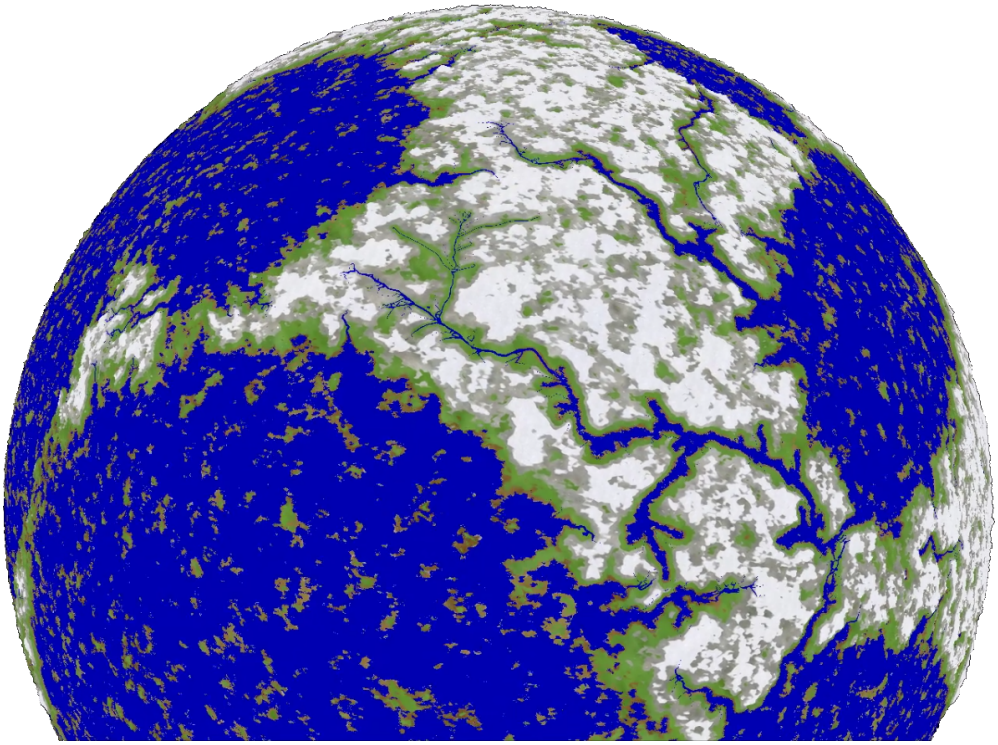


**Figure 9.15:** Screenshot of a generated planet and river network.

# Part IV

# Conclusion

# Chapter 10

# Conclusion & Future Work

In this chapter we will briefly discuss the advantages and limitations of our approach and ways to make it better in future work.

## 10.1 Conclusion

We present an approach, which can generate or work with an arbitrary input source of height data, and generate river networks on the resulting continents, described by the height data. Arbitrary levels of detail can be added to the resulting mesh by using the diamond-square algorithm, or any other applicable procedural technique. Utilization of parallel graphics hardware makes the approach feasible to perform in real-time. River and water data can also be generated in real-time with arbitrary levels of detail. The source of this water data can, as is the case with the height data, be of an arbitrary form, as long as we can generate the necessary sample data we describe in Section 6.4.

The fact that we use regular meshes to facilitate easier parallelization leads to incompatibilities between the mesh resolution (size) and the river representation. This limitation can be overcome by painting the river on the planet, once the mesh has enough resolution, but before the water geometry can be created. If the river is too small in comparison to the mesh, even this painting process does not achieve the desired result.

We look for intersections between the convex hull of each river and its neighboring rivers. This means that if a river has a very large convex hull, it may take up a large space on the continent, thus preventing other rivers from spawning on that same continent. This means that all rivers almost surely have some distance between them.

## 10.2 Future Work

Currently, the switch between levels of detail occurs instantaneously, once the distance of the individual patches pass a certain threshold. This means that patches with less detail may border patches with much greater detail. As this fact leads to holes along the patch boundaries, other techniques may be utilized to hide these artifacts. One of these techniques was presented in the work by Strugar [40], which morphs the less detailed mesh

into the more detailed mesh over a short time also depending on the distance, thus making the transition less obvious.

A further improvement could be made to our approach by adding detail to the water data not by computing the patch water data with the river at the same level of detail as the patch, but rather use the most detailed version of the river. This way, our approach could be used similarly to the diamond-square algorithm adds detail to our height data on-the-fly. The already computed water data is copied from the parent quadtree patch to its children, when necessary, and only the missing values have to be computed. This would lead to possibly reducing the number of blocks necessary per patch to generate the data, while increasing the number of sample points to be evaluated. As our algorithm is a weighted sum algorithm, we could split the necessary work per vertex across multiple threads. A balance between the increase in sample points and further splitting the work per vertex over multiple threads, with a final step to add the resulting data, may lead to an overall increase in performance.

# Appendix A

# Sample Configuration File

```
<?xml version="1.0"?>
<planet>
  <planet-attributes>
    <radius>1024.0f</radius>
    <continents>5</continents>
    <landmass-percentage>30.0f</landmass-percentage>
    <sea-level-height>0.4f</sea-level-height>
    <max-mountain-height>16.0f</max-mountain-height>
    <atmosphere-height>32.0f</atmosphere-height>
    <master-seed>-1</master-seed>
    <continents-from-file>false</continents-from-file>
  </planet-attributes>

  <river-attributes>
    <influence-distance>20.0f</influence-distance>
    <basin-depth>0.1f</basin-depth>
    <basin-water-offset>0.001</basin-water-offset>
    <starting-width>2.5f</starting-width>
    <min-width>0.075f</min-width>
    <ending-factor>2.0f</ending-factor>
    <segments>5</segments>
  </river-attributes>

  <rendering-attributes>
    <vertices>33</vertices>
    <max-level-of-detail>6</max-level-of-detail>
    <level-of-detail-switch-factor>8</level-of-detail-switch-factor>
  </rendering-attributes>
</planet>
```

# Bibliography


[1] F. Aurenhammer. Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM Computing Surveys (CSUR)*, vol. 23, no. 3, pp. 345–405, 1991. Cited on page 23.

[2] F. Belhadj and P. Audibert. Modeling landscapes with ridges and rivers. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pp. 151–154, 2005. Cited on page 14.

[3] B. Benes and R. Forsbach. Layered data representation for visual simulation of terrain erosion. In *Proceedings Spring Conference on Computer Graphics*, pp. 80–86. IEEE, 2001. Cited on page 14.

[4] B. Beneš and R. Forsbach. Visual simulation of hydraulic erosion, 2002.

[5] N. Chiba, K. Muraoka, and K. Fujita. An erosion model based on velocity fields for the visual simulation of mountain scenery. *The Journal of Visualization and Computer Animation*, vol. 9, no. 4, pp. 185–194, 1998. Cited on page 14.

[6] G. Cordonnier, J. Braun, M.-P. Cani, B. Benes, E. Galin, A. Peytavie, and E. Guérin. Large scale terrain generation from tectonic uplift and fluvial erosion. In *Computer Graphics Forum*, vol. 35, pp. 165–175. Wiley Online Library, 2016. Cited on page 11.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009. Cited on page 26.

[8] N. Corporation. Cuda c++ programming guide, 2019. Cited on pages xviii, 58, 59, 60, and 61.

[9] D. D'ambrosio, S. Di Gregorio, S. Gabriele, and R. Gaudio. A cellular automata model for soil erosion by water. *Physics and Chemistry of the Earth, Part B: Hydrology, Oceans and Atmosphere*, vol. 26, no. 1, pp. 33–39, 2001. Cited on page 14.

[10] C. De Boor, C. De Boor, E.-U. Mathématicien, C. De Boor, and C. De Boor. *A practical guide to splines*, vol. 27. springer-verlag New York, 1978. Cited on page 27.

[11] E. Derzapf, B. Ganster, M. Guthe, and R. Klein. River networks for instant procedural planets. *Computer Graphics Forum*, vol. 30, no. 7, pp. 2031 – 2040, 2011. doi:10.1111/j.1467-8659.2011.02052.x. Cited on pages xvii, 6, 12, 15, and 19.

[12] R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, vol. 4, no. 1, pp. 1–9, 1974. Cited on page 21.

[13] A. Fournier, D. Fussell, and L. Carpenter. Computer rendering of stochastic models. *Commun. ACM*, vol. 25, no. 6, p. 371–384, 1982. doi:10.1145/358523.358553. Cited on page 12.

[14] A. Fournier, D. Fussell, and L. Carpenter. Computer rendering of stochastic models. *Commun. ACM*, vol. 25, no. 6, p. 371–384, 1982. doi:10.1145/358523.358553. Cited on page 24.

[15] J. Gain, P. Marais, and W. Straßer. Terrain sketching. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pp. 31–38. ACM, 2009. Cited on page 11.

[16] E. Galin, E. Guérin, A. Peytavie, G. Cordonnier, M.-P. Cani, B. Benes, and J. Gain. A review of digital terrain modeling. *Computer Graphics Forum*, vol. 38, no. 2, pp. 553–577, 2019. doi:10.1111/cgf.13657. Cited on page 9.

[17] E. Guérin, J. Digne, E. Galin, and A. Peytavie. Sparse representation of terrains for procedural modeling. In *Computer Graphics Forum*, vol. 35, pp. 177–187. Wiley Online Library, 2016. Cited on page 11.

[18] É. Guérin, J. Digne, E. Galin, A. Peytavie, C. Wolf, B. Benes, and B. Martinez. Interactive example-based terrain authoring with conditional generative adversarial networks. *Acm Transactions on Graphics (TOG)*, vol. 36, no. 6, p. 228, 2017. Cited on page 11.

[19] A. Lagae, S. Lefebvre, R. Cook, T. DeRose, G. Drettakis, D. Ebert, J. Lewis, K. Perlin, and M. Zwicker. A survey of procedural noise functions. *Computer Graphics Forum*, vol. 29, no. 8, pp. 2579–2600, 2010. doi:10.1111/j.1467-8659.2010.01827.x. Cited on page 12.

[20] D. Luebke, M. Reddy, J. D. Cohen, A. Varshney, B. Watson, and R. Huebner. *Level of detail for 3D graphics*. Morgan Kaufmann, 2003. Cited on page 15.

[21] B. B. Mandelbrot. Stochastic models for the earth's relief, the shape and the fractal dimension of the coastlines, and the number-area rule for islands. *Proceedings of the National Academy of Sciences*, vol. 72, no. 10, pp. 3825–3828, 1975. doi:10.1073/pnas.72.10.3825. Cited on page 11.

[22] B. B. Mandelbrot. *The Fractal Geometry of Nature*, vol. 173. W. H. Freeman and Co., New York, 1983. Cited on page 11.

[23] B. B. Mandelbrot and J. W. Van Ness. Fractional brownian motions, fractional noises and applications. *SIAM Review*, vol. 10, no. 4, pp. 422–437, 1968. doi:10.1137/1010093. Cited on page 11.

[24] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. Van Gool. Procedural modeling of buildings. In *ACM SIGGRAPH 2006 Papers*, pp. 614–623, 2006. Cited on pages 5 and 9.

[25] F. K. Musgrave, C. E. Kolb, and R. S. Mace. The synthesis and rendering of eroded fractal terrains. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '89, pp. 41–50. ACM, New York, NY, USA, 1989. ISBN 0-89791-312-4. doi:10.1145/74333.74337. Cited on page 14.

[26] F. K. Musgrave, D. Peachey, K. Perlin, S. Worley, and D. S. Ebert. *Texturing and Modeling: A Procedural Approach*. Academic Press Professional, Inc., USA, 1994. ISBN 0122287606. Cited on pages 11 and 35.

[27] K. Nagashima. Computer generation of eroded valley and mountain terrains. *The Visual Computer*, vol. 9, no. 13, pp. 456–464, 1997. Cited on page 14.

[28] K. Nicholson and A. Naicker. Gpu based algorithms for terrain texturing, 2008. Cited on page 67.

[29] J. Olsen. Realtime procedural terrain generation, 2004. Cited on page 14.

[30] Y. I. Parish and P. Müller. Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pp. 301–308, 2001. Cited on pages xvii, 5, 9, and 10.

[31] K. Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, vol. 19, no. 3, p. 287–296, 1985. doi:10.1145/325165.325247. Cited on page 11.

[32] P. Prusinkiewicz and M. Hammel. A fractal model of mountains and rivers. In *Graphics Interface*, vol. 93, pp. 174–180. Canadian Information Processing Society, 1993. Cited on page 14.

[33] P. Prusinkiewicz and A. Lindenmayer. *The algorithmic beauty of plants*. Springer Science & Business Media, 2012. Cited on pages 5 and 9.

[34] G. Rozenberg and A. Salomaa. *The mathematical theory of L systems*. Academic press, 1980. Cited on page 4.

[35] M. Schwarz and P. Müller. Advanced procedural modeling of architecture. *ACM Transactions on Graphics*, vol. 34, no. 4 (Proceedings of SIGGRAPH 2015), pp. 107:1–107:12, 2015. Cited on page 9.

[36] D. Shreiner, G. Sellers, J. Kessenich, and B. Licea-Kane. *OpenGL programming guide: The Official guide to learning OpenGL, version 4.3*. Addison-Wesley, 2013. Cited on pages xix, 65, and 66.

[37] M. Steinberger, M. Kenzel, B. Kainz, J. Müller, W. Peter, and D. Schmalstieg. Parallel generation of architecture on the gpu. In *Computer graphics forum*, vol. 33, pp. 73–82. Wiley Online Library, 2014. Cited on page 9.

[38] M. Steinberger, M. Kenzel, B. Kainz, P. Wonka, and D. Schmalstieg. On-the-fly generation and rendering of infinite cities on the gpu. In *Computer graphics forum*, vol. 33, pp. 105–114. Wiley Online Library, 2014. Cited on page 9.

[39] G. N. Stiny. Pictorial and formal aspects of shape and shape grammars and aesthetic systems., 1975. Cited on page 9.

[40] F. Strugar. Continuous distance-dependent level of detail for rendering heightmaps. *Journal of graphics, GPU, and game tools*, vol. 14, no. 4, pp. 57–74, 2009. Cited on page 91.

[41] S. T. Teoh. Riverland: An efficient procedural modeling system for creating realistic-looking terrains. In *International Symposium on Visual Computing*, pp. 468–479. Springer, 2009. Cited on page 11.

[42] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky. Instant architecture. *ACM Transactions on Graphics (TOG)*, vol. 22, no. 3, pp. 669–677, 2003. Cited on page 9.

[43] H. Zhou, J. Sun, G. Turk, and J. M. Rehg. Terrain synthesis from digital elevation models. *IEEE transactions on visualization and computer graphics*, vol. 13, no. 4, pp. 834–848, 2007. Cited on page 11.