Institute for Visualization and Interactive Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit Nr. 74

Feature Based Volumetric Terrain Generation

Michael Becher

Course of Study:	Informatik
Examiner:	Prof. Dr. Thomas Ertl
Supervisor:	Dr. Guido Reina, Dr. Michael Krone
Commenced:	January 18, 2016
Completed:	July 29, 2016
CR-Classification:	I.3.5, I.3.7

Abstract

Two-dimensional heightfields are the most common data structure used for storing and rendering of terrain in offline rendering and especially real-time computer graphics. By its very nature, a 2D heightfield cannot store terrain structures with multiple vertical layers such as overhangs and caves. This restriction is lifted if a volumetric data structure is chosen in place of a 2D heightfield. However, the workflow of manual modelling and editing of volumetric terrain usually involves a large number of minor edits and adjustments and is very time consuming. Therefore, I propose to use three-dimensional curve-based primitives to efficiently model prominent, large scale terrain features and present techniques for volumetric generation of a complete terrain surface from the sparse input data by means of diffusion-based algorithms. By combining an efficient, feature-based toolset with a volumetric terrain representation, the modelling workflow is accelerated and simplified while offering the full artistic freedom of volumetric terrain.

Contents

1 Introduction

The study and depiction of landscapes has a long-standing history in traditional arts such as painting[1] and photography as well as in literature. It comes as no surprise that virtual landscapes have been present in the field of computer graphics since its early days. As early as 1980, animated short *Von Libre* by Loren Carpenter explored a mountain scene depicting fractal terrain [Car80]. In the real-time computer graphics of the 1970s and early 1980s, virtual landscapes were still mostly limited to basic silhouettes drawn by vector graphics or rather coarse 2D raster graphics. But with the rapid increase of processing power, computer graphics algorithms becoming more advanced, and the advent of hardware accelerated 3D computer graphics, the quality of virtual terrain improved likewise. Over the decades, many approaches to rendering landscapes were explored, including bitmap sprites and voxel graphics. Eventually, most real time applications settled on using 3D polygonal meshes to render detailed terrain surfaces. Figure 1.1 contains a personal selection of examples for real-time terrain rendering in games over the last 40 years. Titles were selected to represent a variety of different techniques and styles.

Behind the scenes, heightfields (or heightmaps) were established as a common representation for terrain. A heightfield is formally defined as a map $h : \mathbb{R}^2 \to \mathbb{R}, (x, y)^\top \mapsto h(x, y)$, whereas a heightmap denotes a corresponding 2D texture map that stores terrain elevation data. To this date, heightmaps remain the standard solution for rendering terrain in some of the most popular, commercial game engines including Unreal Engine 4, CryEngine V and Unity [Epia][Cryb][Uni]. However, overhanging terrain structures or caves simply cannot be represented by a heightmap without further extensions. Even terrain without overhangs, but with steep slopes, cliffs or any arbitrary terrain structures that stretch out vertically, can cause issues due to stretched out textures or geometry. These drawbacks have to be compensated for, for example by adding additional geometry to the terrain surface where necessary.

Such restrictions do not apply to a terrain representation in a three-dimensional or volumetric domain. It is possible to represent and store any terrain structure, including in particular overhanging structures, caves or generally terrain with several vertical

[1]Evidence suggests landscape paintings date back to as early as Ancient Greece [Hug]

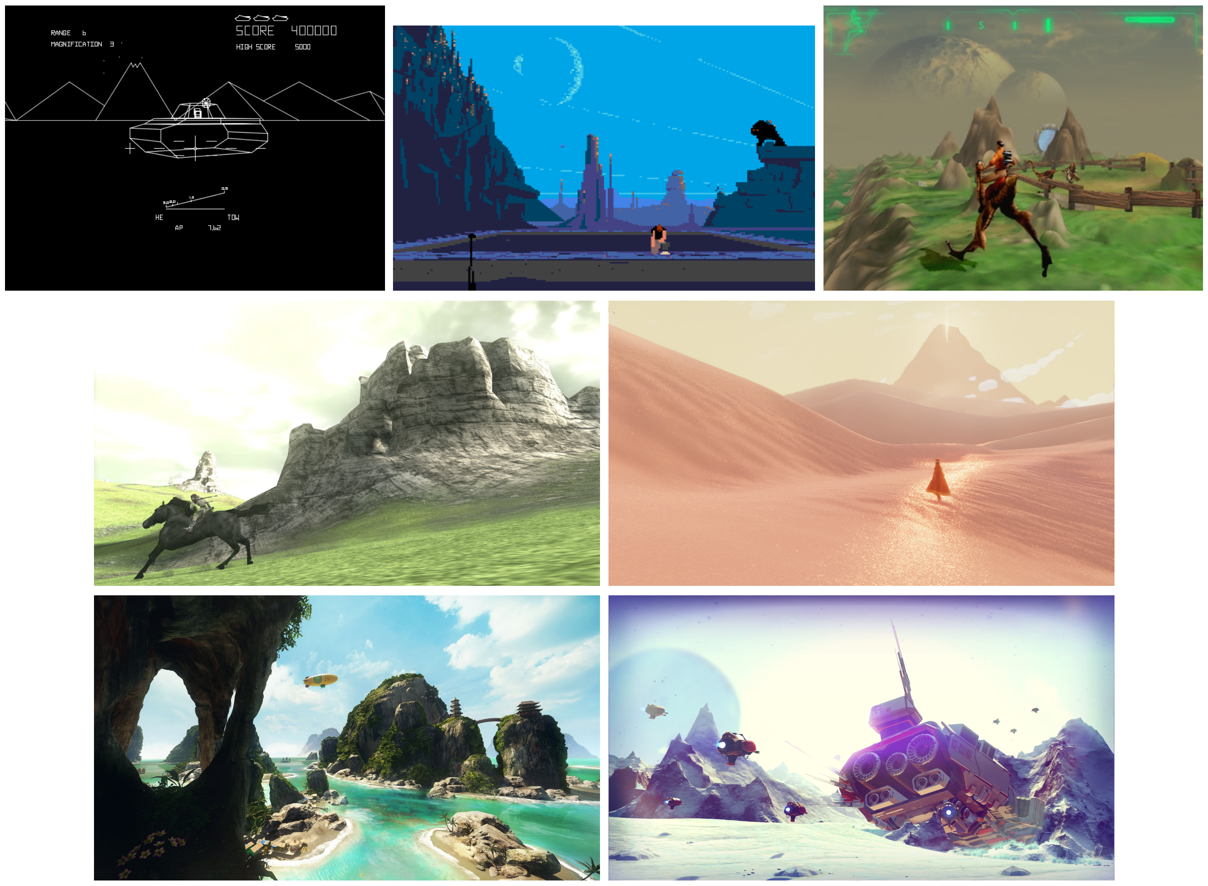

Figure 1.1: A personal selection of terrain rendering in the history of real time computer graphics. From top to bottom and left to right: *Battlezone* (image from Bradley Trainer version) from 1980 uses vector graphics to display a mountain landscape on the horizon (Source: Wikipedia). *Another World* from 1991 uses bitmap images to display landscape in the background (Source: Wikipedia). 1999s *Outcast* featured voxel graphics for terrain rendering (Source: Outcast section of developer website [Fra]). *Shadow of the Colossus'* art direction, including its landscapes, is praised by critics and gamers alike. Published in 2005, the game relies on polygon geometry to render terrain (Source: Giantbomb [Gia]). Released in 2012, *Journey*'s desert landscapes were well received. (Source: "Dynamic Sand Simulation and Rendering in Journey" Siggraph 2012 [Edw]). Crytek's *The Climb* is a 2016 VR exclusive title that represents the current state of the art in real-time graphics. (Source: Official website [Cryc]). Upcoming title *No man's sky* has received much notice for its massive use of procedural asset generation, which includes terrain. (Source: Official website [Hel]).

layers, in a volumetric data structure. Corresponding research, frameworks or extensions to existing graphics engines are indeed available. However, manual creation and editing of such volumetric terrain representations is often a tedious task if every cubic meter has to be manually *painted* with a brush (or rather blob) tool. In many cases it is preferred to generate volumetric terrain data by procedural algorithms. This, on the other hand, limits the control the user can exercise over specific regions of the terrain. As soon as precise and controlled terrain modelling is required, volumetric approaches are not as common as the use of classic 2D heightmaps.

The aim of this thesis is to develop an efficient and intuitive method for controlled generation and editing of volumetric terrains. To that end, I propose to apply concepts and modelling tools that have previously been used to create terrain heigtmaps to volumetric representations. The immediate intention is to let the user model prominent, large-scale terrain features, like ridge lines and cliffs, that define the overall appearance of the landscape. Inspired by the use of curve-based tools for heightmap creation, parametric 3D spline curves are used to model the terrain features and place constraints in the scene. A volumetric terrain representation that matches the input given by the user is automatically generated. A terrain surface is extracted from the volumetric representation and, based on user-controlled constraints, medium to high resolution details are added by simple, procedural means to create a visually pleasing result.

The approach I present in this thesis enables users and level designers to model complex, almost arbitrary terrain structures, including in particular structure that are impossible to represent in heightmaps due to multiple vertical layers. The 3D curve-based modelling tool I propose potentially increases the overall productivity for modelling complex landscapes, as editing of large-scale terrain features is reduced to adjusting a small amount of curve control vertices and properties. This results in faster iterations for terrain modelling workflows both when tweaking existing scenes and when designing large landscapes from scratch. As asset generation and environment modelling, including terrain, is one of the most time consuming and costly aspects of modern video game production, improving workflows is an important factor. Even if terrain features like cave structures and rock formations realised by my method are eventually replaced by custom made, high resolution polygon geometry, it raises the visual bar in the pre-vis stage. Due to the underlying volumetric representation, the creative freedom for level designers is less constraint and the method handles non-realistic, fictional landscapes[2] just as well as more traditional scenes.

[2]Think “floating islands”.

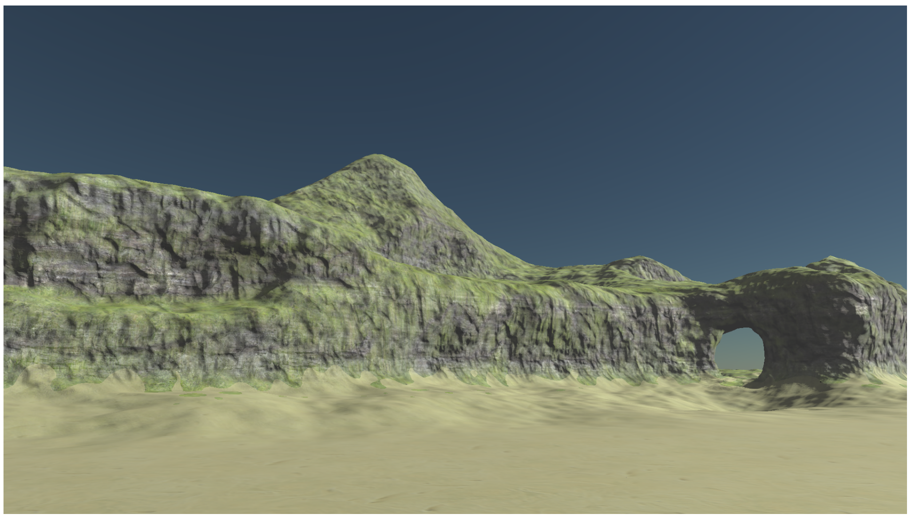

Figure 1.2: Teaser image showcasing a medium scale landscape created with my method. Note the slightly overhanging cliffs and arch.

Structure

Chapter 2 – Related Work: A brief overview of related work in the field of terrain modelling and rendering in the context of computer graphics.

Chapter 3 – Modelling Primitives: Introduction to the modelling primitives I propose to use for modelling complex and overhanging terrain surfaces.

Chapter 4 – Volumetric Terrain Generation Pipeline: After the modelling primitives are established, I present the multistage computational pipeline used to generate arbitrary terrain surfaces from the input modelling primitives. Each stage of this pipeline is discussed in detail.

Chapter 5 – Large Landscapes using Volume Bricking: In this chapter I discuss how subdividing the computational domain into subregions offers greater flexibility and how the pipeline has to be modified to handle distributed computations.

Chapter 6 – Rendering: Before discussing the results of my method, I briefly review the rendering of terrain surfaces in our prototype implementation.

Chapter 7 – Results & Discussion: Analysis of the results generated by my method. I evaluate whether or not the generated terrain surfaces meet basic criteria with regard to quality, measure the performance of the computational pipeline and formulate basic guidelines for working with my toolset.

Chapter 8 – Conclusion: In the final chapter I summarize the results and give an outlook on possible future work.

2 Related Work

This chapter gives a brief overview of related work in the field of terrain modelling and rendering, both in general and in the specialized field of volumetric terrain generation and rendering. An overview of different methods used to model and represent terrain is given in the work of Natali *et al.* [NLP+13]. Different techniques are classified into data oriented scenarios and workflow oriented ones and a comparison of their abilities is made. With regard to the classification of Natali *et al.*, the method I present in this thesis is a workflow oriented, data-free technique.

Researching and developing new techniques for more efficient generation of high quality terrain is common goal. The paper *"Feature based terrain generation using diffusion equation"* by Hnaidi *et al.* presents a framework for modelling terrain based on parametric spline curves [HGA+10]. The *Feature Curve* concept they introduce for modelling terrain features has been a notable inspiration for this thesis. However in contrast to my method, the technique they present is limited to creating heightmaps.

Procedural methods are a popular approach to generating terrain surfaces. A comprehensive overview of general procedural approaches in computer graphics is given in the book *"Texturing and Modeling: A Procedural Approach"* by Ebert *et al.* [EMP+02]. A survey focusing on methods used to procedurally generate terrain is given in *"A Survey of Procedural Methods for Terrain Modelling"* by Smelik *et al.* [SDT+09]. Mark *et al.* present a procedural approach specifically targeting caves in *"Procedural Generation of 3D Caves for Games on the GPU"* [MBMT15]. As it aims to generated terrain with multiple vertical layers, it is noteworthy in the context of this thesis. The creation of caves with my proposed method is briefly discussed in Chapter 7.

Volumetric terrain is a specialized field that often comes with its own methods. A framework with volumetric terrain generation in mind has been presented by Peytavie *et al.* [PGGM09]. They combine material layers stored in a volumetric datastructure with an implicit surface representation and offer high level tools for sculpting the terrain surface. A method for refining terrain described by an implicit function is presented in *"Localised Topology Correction for Hypertextured Terrains"* by Gamito *et al.* [GM08]. The algorithm they present removes components that are disconnected from the main body of the terrain surface.

Alongside the creation and modelling of volumetric terrain, rendering and texturing is also of some interest. In their work *"Level of Detail for Real-Time Volumetric Terrain Rendering"* Scholz *et al.* present a level of detail algorithm for rendering large scale volumetric terrain targeting real-time applications [SBD13].

3 Modelling Primitives

This chapter introduces the modelling tools or rather modelling primitives used to create a volumetric terrain representation and ultimately a terrain surface. Instead of directly shaping and modifying the terrain surface, I propose to place primitives in the scene that describe and constrain the terrain surface in their local vicinity, and from which the terrain surface is propagated into the whole domain. Generally this a known concept and has, at least to some degree, proven its value in real world application. For example, many of today's popular game engines have included spline tools into their terrain editing toolset [Epib][Crya]. Using spline curves, level designers can quickly place roads, rivers and similar structures into their levels and force the terrain surface to match these elements. The novelty of my approach is not the general idea but the application in an otherwise unrestricted three-dimensional computational domain.

Our modelling primitives are grouped into two categories: The first category – our primary modelling tool – encompasses curve-based primitives. These are discussed in detail in section 3.1. The second category is that of simple primitives, which are discussed in section 3.2. I further differentiate between surface modelling primitives and guidance modelling primitives. Surface modelling primitives both define the surface contour in a local area and specify that a surface is actually present at their location. As such they act as a seed location for terrain propagation into the domain. Guidance modelling primitives, on the other hand, only define the potential surface contour in a local area and thus can influence and guide the surfaces in the vicinity but are not necessarily part of or even near the actual terrain surface.

3.1 Feature Curves

Feature Curves are a concept previously used in the context of terrain modelling by Hnaidi *et al.* [HGA+10]. The term is derived from the idea that curve-based modelling primitives are used to describe prominent terrain features such as ridgelines, river beds or cliffs. Due to the shape of many terrain features, like the aforementioned ones, parametric curves are a good fit to design a generic modelling tool for describing such features. My concept of a Feature Curve shares the same basic idea, the exact definition and properties however differ in several aspects from those of Hnaidi *et al.*. Most prominently, I define Feature Curves in 3D space and in doing so I am able to describe almost arbitrary terrain surface structures including, in particular, structures that are impossible to describe with a heightfield. We base our Feature Curves on three-dimensional cubic B-spline curves [DDM+78].

Definition 3.1.1 (B-spline)
*A B-spline **B** of degree n is defined as $\boldsymbol{B}^n(t) = \sum_{i=0}^{m} v_i N_i^n(t)$ where $v_i \in \mathbb{R}^3, 0 \leq i \leq m$ are the control vertices that control the shape of the curve and its position in space, and N_i^n are the B-spline basis functions. Given a knot vector $T = (u_0, \ldots, u_{m+n+1})$ where $(m+1) > n$ the B-spline is defined on the domain $u_n \leq t \leq u_{m+1}$, that is where $(n+1)$ basis functions overlap. By using knot multiplicity, a B-spline curve achieves end-point interpolation, i.e. the spline curve starts and ends in the first and last control vertex v_0 and v_m, respectively.*

To use a B-spline curve as a terrain surface modelling tool, it is further augmented with additional attributes such as left- and right-hand bitangent vectors defining the slope to both sides of the curve and noise parameters for describing high frequency details that cannot be expressed in the smooth curve shape. These attributes are gathered in so called Constraint Points that are placed on and associated with – we also say attached to – a specific Feature Curve.

Definition 3.1.2 (Constraint Point)
A Constraint Point p attached to a Feature Curve $\mathcal{F}$ is defined as a tuple $\left(b_l, b_r, \alpha, \beta, t\right)$ where $b_l, b_r \in \mathbb{R}^3$ are the left- and right-hand bitangent vector, both α and β are scalar values used for noise generation and the scalar value t denotes the Constraint Point's position on $\mathcal{F}$ in the curve's parameter space. The vectors b_l and b_r are perpendicular to the curve's tangent vector at the curve point given by t.

Left- and right-hand bitangent vectors are necessary to describe the terrain surface flow orthogonal to the curve and to furthermore define surface normals. While the bitangent vectors are therefore mandatory for our terrain generation process, noise parameters as a means to procedurally generate high frequency surface details are just one of many possible attributes carried by a Constraint Point to automatically enrich the generated terrain surface. The exact choice of the noise parameters depends on the targeted noise function. We chose to use relatively generic parameters and store amplitude in α and roughness in β. Constraint Points could carry many additional attributes such as material properties or the likelihood of vegetation growing on the surface in the vicinity of a Feature Curve.

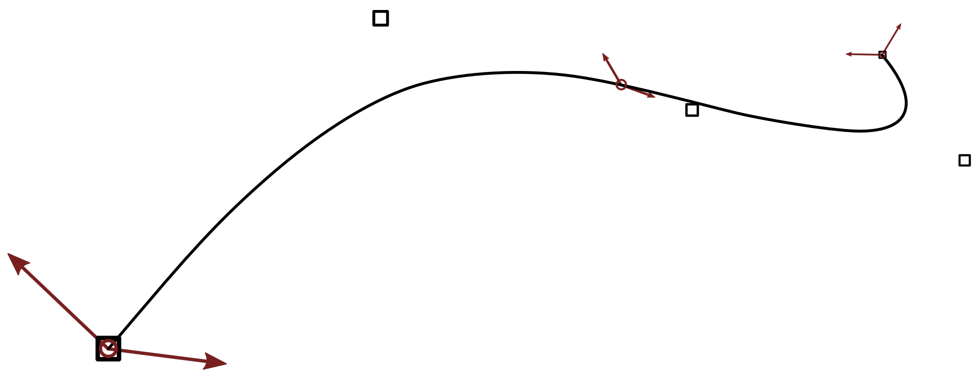

Figure 3.1: A Feature Curve with five control vertices (black squares) and three Constraint Points (red circles). At each Constraint Point, the bitangent vectors are displayed.

Using definition 3.1.1 and 3.1.2, a Feature Curve is defined as follows:

Definition 3.1.3 (Feature Curve)
A Feature Curve $\mathcal{F}$ is defined as $\mathcal{F}(t) = \left(\boldsymbol{B}^3(t), \mathcal{P}(t), s\right)$ where $\boldsymbol{B}^3$ is a cubic B-spline, $\mathcal{P}$ is a parametric function that interpolates between constraint points in a given set P and s is a boolean value that determines whether $\mathcal{F}$ acts as a seed primitive for the terrain surface or simply as a guidance primitive.

Thus, a point on a Feature Curve is a tuple of a position in 3D space (given by the evaluation of a B-spline), a tuple of interpolated constraint attributes and the seed property of the Feature Curve. Figure 3.1 shows a Feature Curve with five control vertices and three Constraint Points.

All Feature Curves have a minimum of two Constraint Points attached, placed at the curve's endpoints. An arbitrary number of additional Constraint Points can be added in between. The attributes given by the Constraint Points are linearly interpolated along the curve, i.e. interpolation coefficients are determined by distance in parameter space. However, the interpolation of bitangent vectors between Constraint Points is non-trivial due to the curve's arbitrary shape and orientation in 3D space. A correct and consistent interpolation is performed as follows.

For a point on the Feature Curve given by the curve parameter t, let p_0 and p_1 be the surrounding Constraint Points. To interpolate a bitangent vector b at t given bitangents by p_0 and p_1, I first compute the curve's tangent vectors $v(t_{p_0})$, $v(t_{p_1})$ and $v(t)$ at the locations of p_0, p_1 and t using a finite difference operator as

$$v(t) = \frac{\mathbf{B}(t+\epsilon) - \mathbf{B}(t-\epsilon)}{|\mathbf{B}(t+\epsilon) - \mathbf{B}(t-\epsilon)|}$$

where ϵ is a small offset in parameter space. Using the angle between the tangent vectors given by $acos(v(t_{p0})^\top \cdot v(t))$ and $acos(v(t_{p_1})^\top \cdot v(t))$ as well as the cross-products $v(t_{p_0}) \times v(t)$ and $v(t_{p_1}) \times v(t)$ it is possible to obtain rotational transformation matrices M_0 and M_1 that map $v(t_{p_0})$ and $v(t_{p_1})$ to $v(t)$. The same transformation matrices can be applied to the bitagent vectors $b(t_{p_0})$ and $b(t_{p_1})$ given by the Constraint Points. The resulting vectors correspond to the bitangents if they were to simply follow the curve's shape from the respective constraint point to the curve point given by t with no interpolation taking place. I then simply interpolate linearly between the two transformed bitagents:

$$b(t) = \alpha \cdot M_0 \cdot b(t_{p_0}) + (1 - \alpha) \cdot M_1 \cdot b(t_{p_1})$$

where α is the linear interpolation factor between p_0 and p_1 at t in parameter space. To guarantee perpendicularity of the interpolated bitagent vector and the curve tangent vector we additionally project the interpolated vector onto the plane defined by the tangent vector:

$$b(t)' = b(t) - (v(t)^\top \cdot b(t)) \cdot v(t)$$

Both the left- and right-hand bitangent vector are interpolated this way in order to evaluate $\mathcal{P}(t)$. Special care is needed if the angle between a tangent vector pair is equal or close to 180° as the rotational transformation is not unique in that case. To avoid undesired behaviour, the reference point is moved away from the respective Constraint Point towards t to a point where bitangent interpolation is possible without ambiguity and from which the rotational transformation to t is unique. Alternatively, additional Constraint Points hidden from the user could be automatically inserted once the angle between tangent vectors at two successive Constraint Points gets close to 180°.

With the ability to interpolate the bitangent vectors along the curve, I can offer a visual intuition of our Feature Curves. Consider a line segment of fixed length. One endpoint of the segment is attached to the Feature Curve while the other one is given by the bitangent vector as interpolated at the attachment point to the Feature Curve and the length of the line. If swept along the Feature Curve, this line segment traces a surface in the shape of a ribbon that is at one edge attached to the Feature Curve. Considering both the left- and right-hand bitangent vector, a Feature Curve can be visualised as a pair of ribbons joined at one edge as illustrated in Figure 3.2. Conceptually this is similar to a Frenet ribbon but instead of the normal of a Frenet-Serret frame, the interpolated Constraint Point bitangent vectors are used.

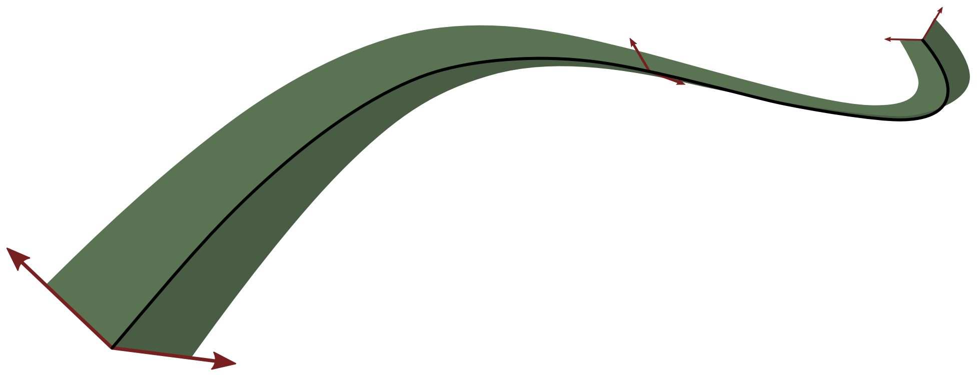

Figure 3.2: A visual representation of a Feature Curve with the left- and right-hand bitangent vectors tracing two ribbons along the Feature Curve.

3.2 Simple shapes

Feature Curves are to some degree specialized to describe elongated features that are widely present in landscapes. In some cases, a number of additional modelling primitives can be useful to supplement the Feature Curves. Especially if the user does not want to explicitly describe a terrain feature but simply wants to locally enforce a normal direction and bitangent orientation, simple geometric shapes are often sufficient. Two basic shapes we propose to use are a quad composed of two triangles and a circle composed of a triangle fan. While the quad is the most atomic modelling primitive, a circle comes in handy to close off tunnel sections, even out plateaus or to simply impose noise constraints to larger, flat areas.

3.3 Proxy Geometry

With respect to the terrain generation process presented in the upcoming chapter, it is advantageous to find a common representation for all modelling primitives. Especially for Feature Curves a more practicable representation is needed than the continuous definition as a parametric curve. Having all primitives share a common representation allows us to process them using a single routine.

Since all modelling primitives are entities placed and arranged in the scene, proxy geometry is needed to give visual feedback of their placement and shape at any rate. It stands to reason that some form of triangle mesh is best suited for this purpose. Such a graphical representation is also exactly what is required by the terrain generation pipeline. Recall the visual intuition of a Feature Curve as a pair of connected ribbons. A proxy triangle mesh for Feature Curves is created in this image. To that end, Feature

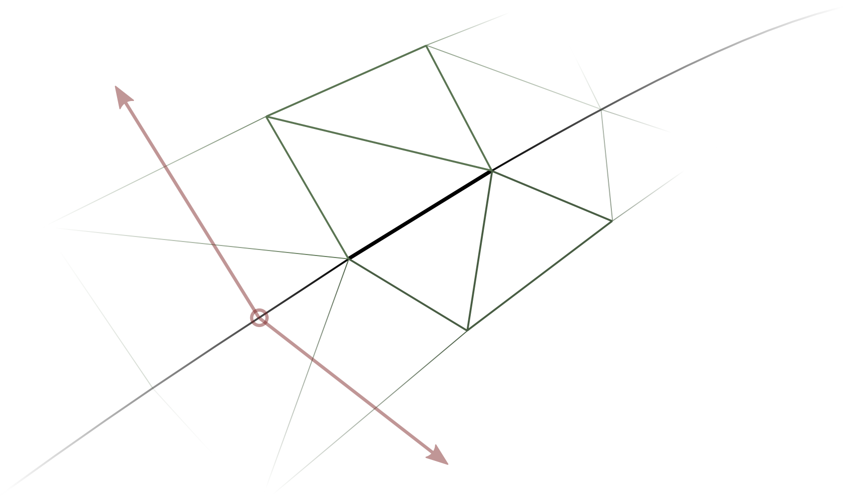

Figure 3.3: Close up of a single line segment and the triangle arrangement used to construct the proxy mesh.

Curves are first discretized into piecewise linear line segments. The number of line segments the Feature Curve is subdivided into is either fixed or based on the length of the curve. For each line segment, the left- and right-hand bitangent vector is interpolated at the endpoints and used to create two distorted rectangles that extend from the line segment some way into direction of one of the bitangents, respectively. Each rectangle is simply build from two triangles. Figure 3.3 illustrates the mesh layout for a line segment. The final result is a triangle mesh in the shape of two ribbons. Curve position and shape as well as bitangent vector information is implicitly contained in the vertex positions of the proxy mesh. Additional constraint properties such as noise generation parameters are stored as additional vertex attributes alongside a per vertex normal vector which is derived from the curve tangent at the segment's endpoints and the interpolated bitangent vectors. A complete proxy mesh of a Feature Curve is shown in Figure 3.4.

As simple shapes are already constructed as triangle meshes, no additional work is needed for this category of modelling primitives.

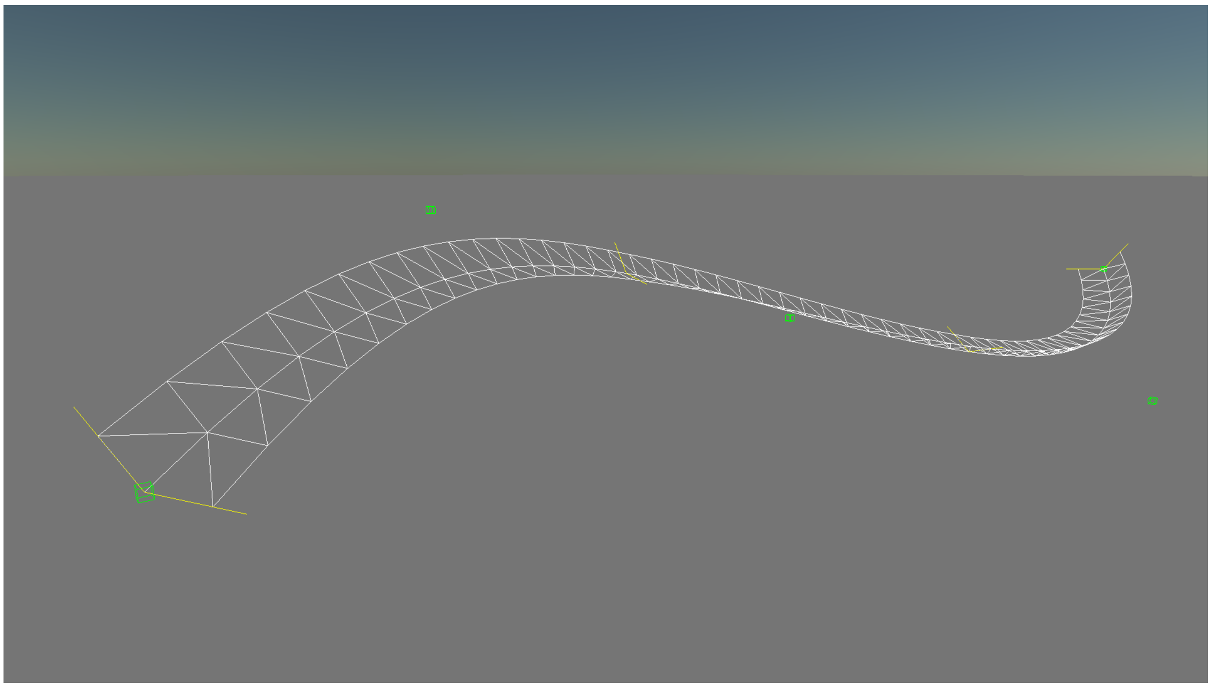

Figure 3.4: A screenshot of a Feature Curves as displayed in the editor interface of my prototype implementation. Green boxes represent control vertices and Constraint Point bitangent vectors are indicated by yellow lines. The proxy mesh of the Feature Curve is display as a white wireframe to not unnecessarily obstruct the view.

4 Volumetric Terrain Generation Pipeline

At the heart of my framework for feature based volumetric terrain generation lies a computational pipeline that creates a terrain surface mesh from a given set of input modelling primitives. This chapter discusses the algorithms and techniques implemented within this pipeline. Section 4.1 gives an overview of all stages of the pipeline and defines the domain in which computations are performed. Sections 4.2 to 4.7 detail the individual stages of the pipeline.

4.1 Overview

The input modelling primitives implicitly define a surface, which in the following is referred to as the target terrain surface. To retrieve an explicit representation of this surface, I make the basic assumption that the target surface is predominantly smooth, that is that the second derivative of its normal vectors equals zero. Furthermore, recall that by definition of the modelling primitives, the target terrain surface passes through the location of surface modelling primitives and in the vicinity of both surface modelling primtives and guidance ones, the surface complies the the normal vectors given by modelling primitives. From these properties, it follows that one can compute a dense normal vector field using the normal vectors given by the modelling primitives as initial values for a diffusion process. This vector field can be computed for the whole area of interest regardless of whether the terrain surface will be present in a region or not. As this vector field conforms to the target surface properties (i.e. at the location of modelling primitives it is equal to the normal vectors given by the primitives and is smooth everywhere else), it is also an implicit and most notably a dense description for any potential terrain target surfaces. Combined with points that are initially known to be located on the surface, both an implicit and an explicit representation of the target surface is derived from the normal vector field.

The area in which the terrain surface is generated is restricted to a limited domain that is defined by a discrete, three-dimensional, uniform, regular grid. All stages of the pipeline operate on discrete vector- and scalar fields aligned to that grid.

Definition 4.1.1 (Scalar field)
A map $\boldsymbol{I} : \{0, \ldots, w-1\} \times \{0, \ldots, h-1\} \times \{0, \ldots, d-1\} \subset \mathbb{N}^3 \to \mathbb{R}$ *is called a discrete three-dimensional scalar field with width* w*, height* h *and depth* d*. It maps a position* (i, j, k) *to a scalar value* $I(i, j, k)$*.*

Definition 4.1.2 (Vector field)
A map $\boldsymbol{V} : \{0, \ldots, w-1\} \times \{0, \ldots, h-1\} \times \{0, \ldots, d-1\} \subset \mathbb{N}^3 \to \mathbb{R}^n$ *is called a discrete three-dimensional vector field with width* w*, height* h *and depth* d*. It maps a position* (i, j, k) *to a n-dimensional vector* $V(i, j, k)$*.*

Within the domain, the following fields are defined to store information computed during the pipeline execution:

- $\mathcal{N}$: A vector field containing surface normal vectors
- $\mathcal{B}$: A vector field containing surface bitangent vectors
- $\mathcal{S}$: A scalar field storing terrain surface information
- $\mathcal{R}$: A vector field containing 2D vectors that store amplitude and roughness parameters for a noise generator

The normal vector field $\mathcal{N}$ stores surface normal vectors for the whole domain and can be computed from sparse input data without additional knowledge. On its own it gives no information about the location of a surface, but if a surface point is known it is possible to compute the complete surface using $\mathcal{N}$. The bitangent vector field $\mathcal{B}$ stores surface tangent vectors, that is vectors that are perpendicular to the corresponding entries of $\mathcal{N}$ and tangential to the surface. Field $\mathcal{B}$ is used to describes the surface flow between modelling primitives. Together, $\mathcal{N}$ and $\mathcal{B}$ are referred to as the guidance fields because they are used to guide the computation of the target terrain surface in $\mathcal{S}$. Field $\mathcal{S}$ contains an implicit description of the target terrain surface as a signed distance function. The resolution of the fields $\mathcal{N}$, $\mathcal{B}$, $\mathcal{R}$ and $\mathcal{S}$ equals the resolution of the domain grid. Consequently, each value of a field – in 3D fields usually referred to as a voxel – corresponds to grid cell. Voxels are located at the center of the corresponding cell. In contrast to voxels, grid cells actually occupy a certain space in the domain and operations such as geometric intersections are therefore expressed with regard to the grid cells.

Before the pipeline is executed, all fields are set to specific initial values. All values of the normal and bitangent fields $\mathcal{N}$ and $\mathcal{B}$ are set to zero vectors, as are the values of the noise parameter field $\mathcal{R}$. The surface field $\mathcal{S}$ is set to contain the maximum possible distance within the field (given in voxels), i.e. $\sqrt{width^2 + height^2 + depth^2}$.

During the discussion of the individual pipeline stages, the following consistent notation is used:

- $\mathbf{x} \in \mathbb{R}^3$ denotes an arbitrary position in the continuous domain
- (i, j, k) with $i, j, k \in \mathbb{N}$ denotes an indexed position in a discrete, three-dimensional grid or field
- $c_{i,j,k}$ is the grid cell at position (i, j, k)
- $\mathcal{N}_{i,j,k}$ is a single value of field $\mathcal{N}$
- $\mathcal{N}(\mathbf{x})$ is an interpolated value of $\mathcal{N}$ at position $\mathbf{x}$
- Notation regarding $\mathcal{B}$, $\mathcal{R}$ and $\mathcal{S}$ is analogue to $\mathcal{N}$

My approach to generating a renderable terrain surface representation is designed as a computational pipeline that consists of the five main stages (plus one optional stage) listed below.

1. **Voxelization** – Takes the modelling primitives as input and converts them to the computational domain.
2. **Guidance field diffusion** – Computes a dense normal vector field and bitangent one using diffusion algorithms.
3. **Noise field diffusion** – Computes a dense field of noise parameters using a diffusion algorithm.
4. **Feature Curve expansion** – Optional. Automatically expands the user-defined Feature Curves along the guidance fields.
5. **Surface propagation** – Propagates surface field values along the guidance field.
6. **Surface mesh reconstruction** – Takes the final surface field and extracts an iso-surface as a triangle mesh.

4.2 Voxelization

The initial step in the terrain generation process is to convert the input modelling primitives from their vector-based definition in continuous 3D space to the discrete computational domain. As explained at the end of Chapter 3, proxy meshes are created for all modelling primitives. The input of the terrain generation pipeline can therefore be generalised to triangle meshes. Even generic meshes, for example a pre-modelled asset containing a complete section of terrain, can be used as input for the pipeline.

Hence, the first task of the pipeline is to process input geometry. The proxy meshes of all relevant modelling primitives have to be represented in the computational domain, that is they have to be discretized in a three-dimensional, regular, uniform grid. This process is often referred to as voxelization. Several methods are known to effectively voxelize triangle geometry at interactive frame times by utilizing the GPUs parallel processing power. Crassin and Green propose a method that takes advantage of the GPUs rasterization capabilities [CG12]. For this method, triangles are first transformed into the space of the voxel grid and then rasterized along the most suited coordinate axis using orthogonal projection and a viewport matching the resolution of the corresponding dimensions of the voxel grid. Each rasterized fragment is then mapped to one or several voxels and data is written to a 3D texture accordingly. Note that although the rendering pipeline is used, no conventional 2D render target is bound during rendering as the 3D texture has to be accessed via image store in the fragment shader.

However, I decide to use a more generic approach that is easily portable to different frameworks and computing devices. For each triangle in a proxy mesh, all cells in the uniform grid that intersect the triangle have to be determined and the properties of the triangle have to be stored in the respective voxels of the fields $\mathcal{N}$, $\mathcal{B}$, $\mathcal{R}$ and $\mathcal{S}$. First, each triangle is transformed to the so called voxel space, that is the space where the domain is aligned with the coordinate axes and covers the space from $(0,0,0)^\top$ to $(1,1,1)^\top$. The bounding box of the triangle is then computed to limit the grid cells that are checked for intersection with the triangle[1] to only those that intersect the bounding box. Finding these cells simplifies to a floor and ceil operation in voxel space. Once a cell intersecting the triangle has been found, the data stored in the triangle's vertices is written to the data buffers.

Since a grid cell is potentially intersected by more than a single triangle and each triangle potentially has a different normal and bitangent vector as well as different noise generation parameters, the results stored in the corresponding voxels during voxelization have to be averaged over all intersecting triangles. Therefore voxelization is split into two steps: Gathering and averaging. First, I gather the normal, left-hand and right-hand bitangent vector as well as the noise parameters of all triangles that intersect a cell in an intermediate storage buffer. Once all triangles are processed, the values gathered per cell are averaged and written to $\mathcal{N}$, $\mathcal{B}$, and $\mathcal{R}$ respectively. Algorithm 4.1 describes the complete process.

[1]A comprehensive overview of useful intersection routines is given at http://www.realtimerendering.com/intersections.html

Algorithm 4.1 Voxelization of Feature Curves

```
Input: $\mathcal{P} =$ set of Feature Curves proxymeshes
$M_{VS}$ = transformation matrix from world to voxel space
$\mathcal{S}$ = field that stores surface information
 1: for all $p \in \mathcal{P}$ do
 2:     for all triangles $t \in p$ do
 3:         $v_0, v_1, v_2 \leftarrow$ vertices of $t$
 4:         $v_0 \leftarrow M_{VS} \cdot v_0$
 5:         $v_1 \leftarrow M_{VS} \cdot v_1$
 6:         $v_2 \leftarrow M_{VS} \cdot v_2$
 7:         $\mathcal{B} \leftarrow$ bounding box of $v_0, v_1, v_2$
 8:
 9:         for all grid cells $c_{i,j,k}$ intersecting $\mathcal{B}$ do
10:             if $\neg(t$ intersects $c_{i,j,k})$ then
11:                 skip loop iteration
12:             end if
13:
14:             Add normal of $t$ to intermediate buffer for $c_{i,j,k}$
15:             Add bitangent of $t$ to intermediate buffer for $c_{i,j,k}$
16:             $\mathcal{S}_{i,j,k} \leftarrow 0.0$
17:         end for
18:     end for
19: end for
20:
21: for all grid cells $c_{i,j,k}$ do
22:     if intermediate buffer of $c_{i,j,k}$ is not empty then
23:         $b_l \leftarrow$ average of all left-hand bitangents in intermediate buffer
24:         $b_r \leftarrow$ average of all right-hand bitangents in intermediate buffer
25:         $\mathcal{B} \leftarrow$ average of $b_l$ and $b_r$
26:         $\mathcal{N} \leftarrow$ average of all normals in intermediate buffer
27:     end if
28: end for
```

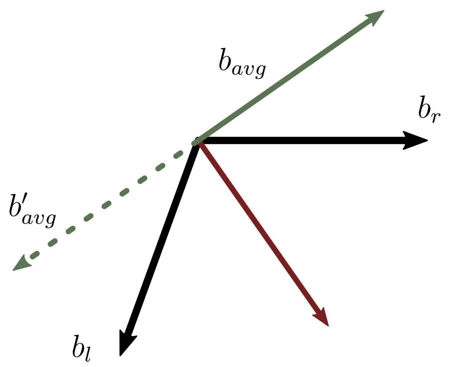

Figure 4.1: A 2D illustration of left-hand bitangent vector and the right-hand one. The red vector shows the result of computing the mean of both vectors. The green vectors show the two possible results of my averaging operation.

4.2.1 Normal and bitangent field

For the upcoming sections, let $\Omega_N \subset \mathcal{N}$ and $\Omega_B \subset \mathcal{B}$ be the subset of voxels that have normal and bitangent vector information stored during the voxelization. The normal vectors gathered in the first step of the voxelization are simply summed up and normalized to obtain the final value. In cells intersected by both ribbons of the Feature Curve, the final normal vector is averaged over normals derived from both the left- and right-hand bitangent vector. The normal field is therefore smoothed along the spine of the Feature Curve.

Special care is required to average the bitangent vectors as desired. Given two bintangent vectors b_l and b_r, I want to compute the vector b_{avg} which maximizes $|b_l^\top \cdot b_{avg}| + |b_r^\top \cdot b_{avg}|$. Figure 4.1 illustrates two bitangent vectors, the vector obtained by simple averaging and the vectors that satisfy our criterion. Instead of computing the mean value of all gathered bitangent vectors, I first average left-hand and right-hand bitangent vectors separately. If the angle between the averaged left- and right-hand vector is larger than $90°$, the vectors are subtracted to obtain the desired vector, otherwise added up. Observe that in the first case the resulting vector is not well defined due to subtraction being anticommutative. Both possible results are valid bitangent vectors and in a sense are equal with regard to our criteria for bitangent vectors.

4.2.2 Noise field

For the upcoming sections, let $\Omega_R \subset \mathcal{R}$ be the subset of voxels that have noise parameters stored during the voxelization. Just like normal and bitangent vectors, the noise parameter values are averaged over all triangles intersecting the corresponding cell of that voxel. This is simply computed as the mean value of all values stored during gathering.

4.2.3 Surface field

For the upcoming sections, let $\Omega_S \subset \mathcal{S}$ be the subset of voxels corresponding to cells that are intersected by at least a single triangle. If the voxelized Feature Curve acts as a seed primitive for the terrain surface all voxels in Ω_S are simply set to a value of 0 since $\mathcal{S}$ is used as a signed distance function (see Section 4.6 for details). As the exact distance between a voxel and an intersecting triangle is therefore ignored, no synchronisation between triangles is required and I directly write the value to $\mathcal{S}$ during the gathering step of the voxelization. Feature Curves only acting as guidance primitives do not modify $\mathcal{S}$ during the voxelization.

4.2.4 Implementation

Voxelization is performed on the GPU using OpenGL 4.3 compute shaders and is therefore highly parallelized even though the implementations does not take advantage of the fast rasterization capabilities of the GPU. Gathering and averaging step of the voxelization are realised by two separate compute shaders.

For each modelling primitive, I dispatch as many executions of the gathering shader as there are triangles in the primitve's proxy mesh. Each compute shader execution is equivalent to a single iteration of the loop beginning at line 2 of Algorithm 4.1. The implementations makes extensive use of OpenGL 4.3 Shader Storage Buffer Objects (SSBOs). Both the input modelling primitive proxy meshes and the output intermediate buffer are stored in SSBOs for convenient read-write access to GPU memory. Normal vectors, bitangent ones and noise values are stored in a per-voxel linked list. GPU based linked lists have been researched and used in the context of order independent transparency and the principle directly applies to my use case [YHGT10].

The averaging shader is dispatched once per grid cell and traverses the corresponding linked-list to perform the averaging operations as described above. The shader then directly writes the results to the fields $\mathcal{N}$, $\mathcal{B}$ and $\mathcal{R}$ which are are implemented using 3D textures. For $\mathcal{N}$ and $\mathcal{B}$, single precision floating point textures are used, while for $\mathcal{S}$ and $\mathcal{R}$ half precision is sufficient. Further memory optimizations are noted for future work.

4.3 Guidance Field Diffusion

After the voxelization the fields $\mathcal{N}$ and $\mathcal{B}$ are only sparsely populated. To compute a closed surface from the sparse input data, one first has to compute a dense vector field for both normal and bitangent field.

This problem of computing a dense vector field from sparse input shares similarities to *gradient vector flow*, a method for computing a dense gradient vector field for an image by minimizing an energy functional [XP97]. The energy functional consists of a smoothness term that dominates in regions with small image gradient magnitude and a second term that ensures the result to match the computed image gradients in regions where the gradient magnitude is large. By replacing the image gradient with either $\mathcal{N}$ or $\mathcal{B}$, it is theoretically possible to apply gradient vector flow to my problem. However, after the voxelization the vectors stored in $\mathcal{N}$ and $\mathcal{B}$ either have unit length if they are in Ω_N or Ω_B or zero length otherwise. The weighting of data and smoothness term becomes binary. It follows that voxels in Ω_N or Ω_B simply retain their values while for all others voxels only the smoothness term remains.

4.3.1 Normal field

The most straightforward method to obtain a dense normal vector field that satisfies a smoothness term is to solve Laplace's equation on $\mathcal{N}$ [Eva98]:

$$\Delta\mathcal{N} = (\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2})\mathcal{N} = 0$$

By definition the solution of Laplace's equation yields a smooth result. Although this is generally favourable, there are many cases where a hard *edge* on the terrain surface, and therefore in the normal field, is desired. To retain edges on the surface to at least some extent, instead of using Laplace's equation I opt for using anisotropic diffusion on the normal field $\mathcal{N}$:

$$\frac{\partial\mathcal{N}}{\partial t} = div\,(c(x,y,z,t)\nabla\mathcal{N}) = c(x,y,z,t)\Delta\mathcal{N} + \nabla c \cdot \nabla\mathcal{N}$$

The following discretization as presented by Perona and Malik is used to solve the equation on the discrete field using an iterative computation [PM90].

$$\begin{aligned}\mathcal{N}^{t+1}_{i,j,k} = \mathcal{N}^{t}_{i,j,k} + \frac{1}{6}\Bigg(& c_E \cdot \left[\frac{\partial \mathcal{N}^t}{\partial x}\right]^{+}_{i,j,k} + c_W \cdot \left[\frac{\partial \mathcal{N}^t}{\partial x}\right]^{-}_{i,j,k} \\ & + c_U \cdot \left[\frac{\partial \mathcal{N}^t}{\partial y}\right]^{+}_{i,j,k} + c_D \cdot \left[\frac{\partial \mathcal{N}^t}{\partial y}\right]^{-}_{i,j,k} \\ & + c_N \cdot \left[\frac{\partial \mathcal{N}^t}{\partial z}\right]^{+}_{i,j,k} + c_S \cdot \left[\frac{\partial \mathcal{N}^t}{\partial z}\right]^{-}_{i,j,k} \Bigg)\end{aligned}$$

Where the expression $\left[\frac{\partial}{\partial x}\right]^{+}_{i,j,k}$ denotes the finite difference operator using a forward difference in x-direction at the location (i, j, k), while $\left[\frac{\partial}{\partial x}\right]^{-}_{i,j,k}$ denotes the backward difference operator. At the boundaries of the domain, the finite difference operators will access values outside of domain and thus outside of $\mathcal{N}$. To evaluate the finite difference operators, boundary conditions for the domain are needed. Homogeneous Neumann boundary conditions are used for field $\mathcal{N}$:

$$\begin{aligned}\mathcal{N}_{-1,j,k} &= \mathcal{N}_{0,j,k} \\ \mathcal{N}_{w+1,j,k} &= \mathcal{N}_{w,j,k} \\ \mathcal{N}_{i,-1,k} &= \mathcal{N}_{i,0,k} \\ \mathcal{N}_{i,h+1,k} &= \mathcal{N}_{i,h,k} \\ \mathcal{N}_{i,j,-1} &= \mathcal{N}_{i,j,0} \\ \mathcal{N}_{i,j,d+1} &= \mathcal{N}_{i,j,d}\end{aligned}$$

In the context of anisotropic and edge enhancing diffusion, edges are usually identified using the magnitude of the gradient of the image. In case of a vector valued, discrete function, I found that simply using the dot product between adjacent vectors gives a reasonable result. I define the coefficients c_E, c_W, c_U, c_D, c_N and c_S as a function g of the negative dot product of adjacent voxels in $\mathcal{N}$ mapped to $[0,1]$.

$$c_E = g\left(\frac{1}{2} - \left(\frac{(\mathcal{N}^t_{i,j,k})^\top \cdot \mathcal{N}^t_{i+1,j,k}}{2}\right)\right)$$
$$c_W = g\left(\frac{1}{2} - \left(\frac{(\mathcal{N}^t_{i,j,k})^\top \cdot \mathcal{N}^t_{i-1,j,k}}{2}\right)\right)$$
$$c_U = g\left(\frac{1}{2} - \left(\frac{(\mathcal{N}^t_{i,j,k})^\top \cdot \mathcal{N}^t_{i,j+1,k}}{2}\right)\right)$$
$$c_D = g\left(\frac{1}{2} - \left(\frac{(\mathcal{N}^t_{i,j,k})^\top \cdot \mathcal{N}^t_{i,j-1,k}}{2}\right)\right)$$
$$c_N = g\left(\frac{1}{2} - \left(\frac{(\mathcal{N}^t_{i,j,k})^\top \cdot \mathcal{N}^t_{i,j,k+1}}{2}\right)\right)$$
$$c_S = g\left(\frac{1}{2} - \left(\frac{(\mathcal{N}^t_{i,j,k})^\top \cdot \mathcal{N}^t_{i,j,k-1}}{2}\right)\right)$$

Where g is chosen as a sub-quadratic function well known from the literature [PM90]:

$$g(x) = \frac{1}{1 + (\frac{x}{k})^2}$$

The choice of parameter k depends on the desired amount of smoothing across *edges*. In my current implementation, it is set to a fixed value of 0.01, which I found to produce good results, but adding it to the parameter set of Constraint Points would give additional control to the user. The voxels in $\Omega_N \subset \mathcal{N}$ set during voxelization act as initial conditions for solving the equation. Note that for all voxels that have not yet been set to a meaningful value, i.e. $\mathcal{N}_{i,j,k} = \vec{0}$, the function g will evaluate to 1 due to the dot product being 0. In this case, the anisotropic diffusion equation simplifies to Laplace's equation. I deliberately use this property in order to propagate normal vectors to regions of the field where no meaningful coefficient for the anisotropic diffusion can be computed yet.

The number of iterations required depends on the resolution of the computational grid. To ensure that each voxel is set to a value other then the zero vector, at least $\sqrt{width^2 + height^2 + depth^2}$ iterations should be performed.

4.3.2 Bitangent field

Diffusing the bitangent vector field $\mathcal{B}$ in the same manner as the normal field $\mathcal{N}$ will not yield the desired result. Unlike $\mathcal{N}$, $\mathcal{B}$ only contains one of many vectors that are perpendicular to the normal vector and therefore tangential to the target surface. The orientation of the bitangent field is of much greater interest than the actual direction of the individual bitangent vectors. For the bitangent vectors, I consider the vector field smooth if the dot product of adjacent vectors is either close to one or close to minus one, meaning the sign is ignored. For this reasons, a diffusion process as used for $\mathcal{N}$, which basically computes the mean value of a local neighbourhood, is not applicable. This also becomes apparent from Figure 4.1, which shows a cross section. The red vector obtained as the mean value of b_l and b_r is certainly not perpendicular to mean of the normal vectors corresponding to b_l and b_r.

It is possible to formulate the computation of the most suitable bitangent vector for any given voxel based on a local neighbourhood as an optimisation problem:

$$\mathcal{B}_{i,j,k}^{t+1} = \arg\max_{f} \left(\sum_{x \in \mathcal{NB}_{i,j,k}} \left(\mathcal{B}_x^{t\top} \cdot f \right)^2 \right) \quad \text{s.t. } \|f\| = 1$$

Where $\mathcal{NB}_{i,j,k}$ is the set of indexed positions of grid cells directly adjacent to $c_{i,j,k}$. Examples of neighbourhood vectors are easily found for which no unique solution to this problem exits. A solution to this problem might possibly be found by reformulating it as a total least-squares problem, but for an efficient implementation within a diffusion process, a more straightforward function to compute averaged bitangent vectors is needed. Recall from section 4.2 that I define the average of two bitangent vectors as

$$avg(v, w) = \begin{cases} \frac{v+w}{\|v+w\|} & \text{if } \left((v)^\top \cdot w \right) > 0 \\ \frac{v-w}{\|v-w\|} & \text{else} \end{cases}$$

However the local neighbourhood of a voxel contains six bitangent vectors and the equation is only applicable to two vectors. Nevertheless this function is used to obtain a temporary approximation, as a proper method for averaging all six bitangents of the local neighbourhood could not be developed in time. First, the vector pairs of the three main axes are averaged, obtaining a single vector per axis: b_x, b_y and b_z. From these another average is computed for the xz-plane that is then averaged together with b_y to obtain the final result:

$$\begin{aligned} b_x &= avg(\mathcal{B}_{i+1,j,k}^t, \mathcal{B}_{i-1,j,k}^t) \\ b_y &= avg(\mathcal{B}_{i,j,k+1}^t, \mathcal{B}_{i,j,k-1}^t) \\ b_z &= avg(\mathcal{B}_{i,j+1,k}^t, \mathcal{B}_{i,j-1,k}^t) \\ \mathcal{B}_{i,j,k}^{t+1} &= avg\left(avg\left(b_x, b_y\right), b_z\right) \end{aligned}$$

The function avg is neither commutative nor associativity and changing the order in which vectors are averaged potentially changes the result. Therefore, the resulting bitangent vector is biased.

As another alternative to computing an analytic solution, I also propose a promising idea for computing a suitable bitangent vectorfield that I discuss in theory in the following but could not implement and test in the scope of this thesis. Ultimately, I wish for the bitangent vector field to represent a smooth flow between the modelling primitives. Assume that a dense bitangent vector field is computed by solving Laplace's equation and consider the resulting vector field as a flow field of some virtual fluid. In this view, all modelling primitives technically are sources for outflow and it becomes obvious that the resulting vector field does not connect the modelling primitives. Inspired by this observation, I propose to introduce the following categories to modelling primitives (in particular to Feature Curves), to assist a stable bitangent field computation: Source, sink and pass-through.

For Feature Curves, the category controls the direction of the left- and right-hand bitangent vector.

- **Source** (or outflow) Feature Curves are equivalent to the definition so far. The bitangent vectors are pointing away from the spine of the curve.
- **Sink** (or inflow) Feature Curves invert the direction of the bitangent vectors. They are pointing towards the spine of the curve.
- Finally, **pass-through** Feature Curves invert of either the left-hand bitangent vector or the right-hand one, but not both. The virtual fluid is allowed to *pass through* the curve, possibly getting redirected in the process.

Using these categories, the virtual flow should smoothly move from sources to sinks resulting in an amiable bitangent vector field. The assignment of categories has to be controlled by the user on a case to case basis. Generally, a source primitive should be surrounded by either sink primitives or pass-through primitives, at least if they should connect to a surface.

4.3.3 Implementation

A GPU implementation is used to accelerate the computation of field $\mathcal{N}$ and $\mathcal{B}$. Each iteration of the diffusion equations that are used to compute the guidance fields is performed on the GPU within a compute shader. Thus, a single iteration is highly parallelized.

Iterative algorithms on the GPU, using data stored in textures, require backbuffer textures as write targets when values are read from the primary data textures as source. Else, values are overwritten before it is guaranteed that all parallel executions have read all necessary values before they are altered by other executions. Synchronisation on this scale is hard to achieve if at all possible, making *ping-pong* buffering the preferred approach. Every other iteration swaps the target and source slot between the primary texture and the backbuffer one. Memory barriers have to be used as required to guarantee each iteration finishes writing to the target texture before it is used as the source for the next iteration. Field $\mathcal{N}$ and $\mathcal{B}$ are currently computed separately and require a backbuffer texture each.

4.4 Noise Field Computations

Apart from the essential properties, Features Curves also store parameters for noise generation. I deliberately store only noise generation parameters and not a final 3D noise signal since a lot of the higher frequency details contained in a suitable noise signal are lost unless the resolution of $\mathcal{R}$ is noticeably higher than that of the surface field $\mathcal{S}$. The actual generation of a noise signal is performed after completion of the terrain generation pipeline. During rendering it is used to add additional medium to small scale details to the final surface mesh. Like the values of the guidance fields, the noise parameters have to be propagated into the whole domain. A dense noise field is computed by simply solving Laplace's equation on $\mathcal{R}$ using the voxels in Ω_R as initial values. The implementation of this stage is very similar to the guidance field computations. It uses a compute shader to solve Laplace's equation on the GPU in an iterative computation.

4.5 Feature Curve Expansion

As an additional, optional stage it is possible to expand the input Feature Curves along the guidance fields. This decreases the open space not covered by any user-defined Feature Curves and potentially reduces the ambiguity of the landscape's surface in some areas. After the Feature Curves are expanded, both voxelization and guidance field computation have to be repeated to update $\mathcal{N}$, $\mathcal{B}$ and $\mathcal{R}$. To expand a Feature Curve I trace a virtual curve similar to a streamline through the guidance fields, adding new control vertices to the Feature Curve at given intervals. As soon as a virtual curve closes in on another Feature Curve the process is terminated. A thorough explanation of how a curve is traced through the guidance field is given in 4.6.1. The implementation of the Feature Curve expansion is purely CPU-based and simply uses the standard Feature Curve functionality.

4.6 Surface Propagation

Once the complete guidance fields are successfully computed, all conditions to compute the surface field $\mathcal{S}$ are met. There are multiple ways to construct $\mathcal{S}$ based on the guidance fields and depending on the strategy used, different properties can be of interest.

The two approaches presented in this chapter both share a common idea: To propagate the surface information within $\mathcal{S}$ starting from the values that are stored in Ω_S during the voxelization stage. This is motivated by the possibility to easily parallelize the computations if the information can be independently propagated at various locations at the same time. I classify our approaches as either *push* or *pull*, based on whether voxels push new values to a certain subset of other voxels or pull values from a local neighbourhood to compute a new value to assign to themselves.

4.6.1 Streamline Propagation

The first approach I propose for propagating the surface information into the domain works by tracing points on a potential target surface. If the origin of the tracing process is a known surface point, I deduce that the traced points also are part of this surface.

To that end, let $c(s) \in \mathbb{R}^3$ be a parametric curve defined in our computational domain by

$$\left(\frac{\partial c(s)}{\partial s}\right)^{\top} \cdot \mathcal{N}(c(s)) = 0$$

where $\mathcal{N}(c(s)) \in \mathbb{R}^3$ denotes the normal vector at position $c(s)$ obtained by linear interpolation in $\mathcal{N}$. Such a curve's tangent is in every curve point perpendicular the previously computed normal vector field. Note that the target terrain surface is in theory already well defined by $\mathcal{N}$ and any number of points that are known to be part of the surface. By limiting our interest to curves $c(s)$ that intersect at least one cell that corresponds to a voxel in Ω_S, i.e. a voxel which is known to be part of the surface, we obtain a set of curves that lie on the target terrain surface. This leads to the following approach to computing the surface field $\mathcal{S}$:

Starting in all voxels $\mathcal{S}_{i,j,k} \in \Omega_S$, spawn and trace curves along the guidance field until either a certain distance has been covered or the computational domain is left. For each cell of the domain that is intersected by any of these curves we mark the corresponding entry of $\mathcal{S}$ as being part of the surface and add it to Ω_S. This process is theoretically repeated until no more voxel are added to $\mathcal{S}$ and Ω_S contains the complete target surface. In practise it is repeated only once with a varying strategy for choosing curves to trace.

Since the definition and intuition of $c(s)$ is similar to that of a streamline we refer to this method as *streamline propagation*.

To actually trace a curve $c(s)$, it is approximated by piecewise linear segments. Given a location $\mathbf{x}$ in the domain as the start point of a curve segment, find the normalized direction vector $v(\mathbf{x})$ to compute the linear segment from $\mathbf{x}$ to $\mathbf{x}' = \mathbf{x} + l \cdot v(\mathbf{x})$ where l denotes the length of the segment. The endpoint $\mathbf{x}'$ becomes the start point of the next segment. Feasible directions for v are only constraint to being perpendicular to $\mathcal{N}(\mathbf{x})$ which still leaves an infinite number of possible vectors. This ambiguity is solved by introducing the direction vector v' from the previous segment and choosing the new direction v so that $v(\mathbf{x})^\top \cdot \mathcal{N}(\mathbf{x}) = 0$ and $v^\top \cdot v'$ is maximized. Using this strategy the resulting direction for the next segment is perpendicular to the normal vector field while changes in the curve direction are penalised. Obviously, the first curve segment has no predecessor. An initial direction has to be decided for all curve starting points. A simple consistent strategy is to chose a vector v perpendicular to $\mathcal{N}$ that lies in one of the coordinate planes. Depending on the local normal vector at the curve spawn point, choose a coordinate plane that is not parallel to the plane defined by the normal vector. Once an initial direction is decided, the curves try to keep that direction. If curves are only spawned once at the initial Ω_S, this approach cannot cover the complete target surface since all curves are restricted to a single coordinate plane. To fill any gaps, a second wave of curves is spawned at each voxel in the updated Ω_S. The initial direction at the spawn points however is now set to lie in a coordinate plane orthogonal to the one used in the first iteration.

It is possible to improve the efficiency and result of the streamline propagation by taking the bitangent vector field $\mathcal{B}$ into consideration. Unlike the normal vector field the bitangent vector field gives a single feasible vector to follow if we want to move along the surface. Because the bitangent vectors are perpendicular to the tangent of the Feature Curve they originated from, it is also the optimal direction to move away from Ω_S and propagate information into yet unexplored regions of the domain. In fact the bitangent vector field was computed for exactly this purpose. With regard to the bitangent vector field the definition of $c(s)$ could be formulated as an actual streamline:

$$\frac{\partial c(s)}{\partial s} \times \mathcal{B}(c(s)) = 0$$

Nevertheless, the original definition also still holds true if $\mathcal{N}$ and $\mathcal{B}$ match and are well defined. The ambiguity of having to chose an initial direction for the curves is now obsolete, as is the need to stabilize the curve by penalising deviation from the previous curve direction. During the first iteration, $v(\mathbf{x})$ is simply set to $\mathcal{B}(\mathbf{x})$. The bitangent vectors are directly used to construct the piecewise linear curve just like in a usual streamline computation. Depending on the structure of the bitangent field it might be necessary to invert the sign of the vectors read from $\mathcal{B}$ while tracing the curve. Again the direction vector v' from the previous segment is used in order to keep the general direction of the curve. If the sign of the dot product of v and v' is negative we switch the sign of v to keep the computation from stalling. During the second iteration, the direction vector is again chosen perpendicular to the directions used in the first iteration and the normal field. The cross product of $\mathcal{N}$ and $\mathcal{B}$ delivers the desired vector. An overview of the algorithm using both $\mathcal{N}$ and $\mathcal{B}$ is shown in Algorithm 4.2.

Algorithm 4.2 Streamline Propagation

```
Input: $\mathcal{N}$, $\mathcal{B}$, $\mathcal{S}$, $\Omega_S \subset \mathcal{S}$
Output: $\mathcal{S}$
 1: for $i \in \{0,1\}$ do                                          // two iterations
 2:     for all $\mathcal{S}_{i,j,k} \in \Omega_S$ do
 3:         $\mathbf{x} \leftarrow (i,j,k)$                          // set the origin of the curve tracing
 4:
 5:         if $i = 0$ then                  // set initial direction based on current iteration
 6:             $v \leftarrow \mathcal{B}(\mathbf{x})$
 7:         else
 8:             $v \leftarrow \mathcal{B}(\mathbf{x}) \times \mathcal{N}(\mathbf{x})$
 9:         end if
10:
11:         $v' \leftarrow v$
12:
13:         while $x$ in domain do                                  // loop for tracing curve
14:             $\mathbf{x} \leftarrow \mathbf{x} + l \cdot v$
15:             $\mathcal{S}_{\text{floor}(\mathbf{x})} \leftarrow 0$      // add cell corresponding to current location to surface
16:             Add $\mathcal{S}_{\text{floor}(\mathbf{x})}$ to $\Omega_S$
17:
18:             if $i = 0$ then                                     // get new direction
19:                 $v \leftarrow \mathcal{B}(\mathbf{x})$
20:             else
21:                 $v \leftarrow \mathcal{B}(\mathbf{x}) \times \mathcal{N}(\mathbf{x})$
22:             end if
23:
24:             $v \leftarrow \text{sign}((v)^\top \cdot v') \cdot v$     // avoid deadlocks from bitangent vectors facing
    each other
25:             $v' \leftarrow v$
26:         end while
27:     end for
28: end for
```

The streamline propagation method generally has a few issues. First of all it is difficult to guarantee that every cell that is theoretically intersecting the target surface is also intersected by at least one curve. It is possible to solve this issue by switching the algorithm from a push to a pull approach. Instead of spawning curves in every voxel in Ω_S, curves are spawned in every voxel in $\mathcal{S} \setminus \Omega_S$. If a curve intersects at least one voxel in Ω_S we deduce that the voxel that the curve originated from is also intersecting the target surface and is set to 0.0 and added to Ω_S. However, this approach introduces false positives and increases the running time of the algorithm since usually $|\Omega_S| \ll |\mathcal{S}|$. If the vector fields $\mathcal{N}$ and $\mathcal{B}$ exhibit singular points of node or focus type, the pull variation of the algorithm also tends to generate surfaces that are several voxels thick or even multiple surfaces originating from a single feature curve. Another issue of the streamline propagation in its current stage is that it tends to lead to a double surface when using an iso-surface computation to extract the terrain surface from $\mathcal{S}$. While in the upcoming distance field propagation method a sign change in the surface field is a clear indication to the terrain surface being present, no differentiation between voxels above and below the surface is made in the streamline propagation and all values in $\mathcal{S}$ are greater or equal to 0. Consequently, in practice the iso value chosen for the surface extraction should not be exactly 0 and value in $\mathcal{S}$ slightly larger than 0 occur twice around a surface. Additionally, the streamline propagation method relies on the gradient field $\mathcal{B}$ to generate the best results, but the quality of $\mathcal{B}$ itself currently does not live up to my expectations.

4.6.2 Distance Field Propagation

As the streamline propagation method suffers from several issues and, in its improved version, relies on the still biased bitangent field, I pursue a second approach to propagating the surface information. The second method is realised as an incremental pull approach. Unlike in the previous method, only the local neighbourhood of voxels is considered in each step of the incremental computation.

A classic way to implicitly describe the surface of 3D objects using a 3D scalar field is by means of a signed distance function (SDF). In general, each function value of a SDF is set to the smallest distance between the value's position in the domain and the described object's surface. In our case the sign of the distance values also determines whether a point is located above or below the terrain surface.

Definition 4.6.1 (Signed Distance Function)
Let f be a scalar field and s be a surface inside the domain of f. Furthermore, let $\boldsymbol{x}_s$ be the coordinate of a surface point on s that is closest to a point in f with coordinate $\boldsymbol{x}$ and $N(\boldsymbol{x}_s)$ the normal vector of the surface at $\boldsymbol{x}_s$. Then the value of f at $\boldsymbol{x}$ is set to

$$f(\boldsymbol{x}) = \begin{cases} \|\boldsymbol{x} - \boldsymbol{x}_s\| & \textit{if } \left((\boldsymbol{x} - \boldsymbol{x}_s)^\top \cdot N(x_s)\right) > 0 \\ -\|\boldsymbol{x} - \boldsymbol{x}_s\| & \textit{else} \end{cases}$$

The distance field propagation methods aims for $\mathcal{S}$ to be computed as a signed distance function with regard to our target terrain surface. Obviously, at this point I do not have a surface as input to compute $\mathcal{S}$ but rather I want to compute $\mathcal{S}$ by different means in order to reconstruct the target surface from it. To compute $\mathcal{S}$ based only on the initial values set during the voxelization and the guidance field and I define a function that approximates a voxel's distance value based on it's local neighbourhood.

Let $\mathcal{NB}_{i,j,k}$ be the set of coordinates adjacent to the location (i,j,k) in the domain, i.e. $(i \pm 1, j, k)$, $(i, j \pm 1, k)$ and $(i, j, k \pm 1)$. Given a cell c_x with $x \in \mathcal{NB}_{i,j,k}$, assume that a target surface intersecting c_x is locally planar and its orientation is given by $\mathcal{N}_x$. Then I can model this surface as a plane defined by the midpoint of c_x and the normal vector $\mathcal{N}_x$. The signed distance d between the midpoint of cell $c_{i,j,k}$ and this plane is obtained by a simple point-plane distance calculation. This is illustrated in Figure 4.2. In case the target terrain surfaces actually intersect c_x, i.e. $\mathcal{S}_x = 0$, the obtained distance value d is assigned to $\mathcal{S}_{i,j,k}$ unless a smaller distance is computed for one of the other neighbour cells. If on the other hand c_x is not intersected by the target surface but the distance to the nearest surface point is already known and stored in $\mathcal{S}_x$ I simply assign the sum $\mathcal{S}_x + d$ to $\mathcal{S}_{i,j,k}$. If c_x is neither intersected by the surface nor a valid distance is known yet, no value based on c_x is assigned to $\mathcal{S}_{i,j,k}$ at this point. Due to the construction of $\mathcal{N}$ this is a feasible approximation to the distance value at location (i,j,k). The surface normal indicates the direction to the nearest surface and is therefore a reasonable argument for determining whether a voxel is closer or further away from a surface than the reference voxel. The complete computation of $\mathcal{N}$ is performed as follows:

Recall that $\Omega_S \subset \mathcal{S}$ is the subset of $\mathcal{S}$ that is set during the voxelization and let $\partial\Omega_S \subset \mathcal{S} \setminus \Omega_S$ be the boundary of Ω_S and $\mathcal{S} \setminus \Omega_S$. Set every value $\mathcal{S}_{i,j,k} \in \partial\Omega$ according to the following function and insert it into Ω_S.

$$\mathcal{S}_{i,j,k} = \min_{x \in \mathcal{NB}_{i,j,k}} \left(\mathcal{S}_x + ((i,j,k) - x) \cdot \frac{\mathcal{N}_{i,j,k} + \mathcal{N}_x}{2} \right)$$

Once $\Omega_S = \mathcal{S}$, a signed distance field for our target surface is successfully estimated. Instead of the normal vector at the neighbour location x, the actual calculation uses an interpolated normal located between (i,j,k) and x in order to reduce the error introduced by assuming the surface to be locally planar.

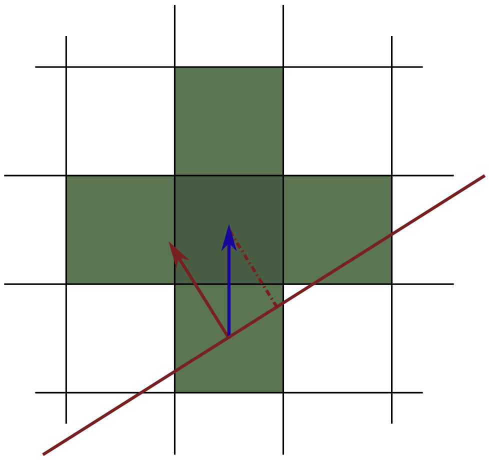

Figure 4.2: A 2D cross section with a cell $\mathcal{S}_{i,j,k}$ in the center in dark green and its direct neighbours in lighter green. The blue vector is the direction vector from a neighbour cell midpoint and the red one is the corresponding normal vector. The dotted red line shows the distance between the cell midpoint and the plane defined by the normal, which is the projection of the vector onto the red one.

Note that each value $\mathcal{S}_{i,j,k}$ is set only once. An iterative computation that tries to improve the distance approximation over time proved to be unstable. Modelling the surface contour as locally planar introduces an error that can lead to two neighbouring voxels continually de- or increasing each other's value. This might be improved by further research into a more elaborate model of the local surface contour and resulting distance approximation.

Due to the nature of the volumetric terrain representation, it is possible to have more than a single connected terrain surface and each surface may or may not be closed. Therefore it is very difficult to define a clear concept of in- and outside of the terrain even when considering the surface normal given at every location in the domain by $\mathcal{N}$. It might often occur that two neighbouring voxels of our surface field $\mathcal{S}$ have different signs simply because the voxel's values originate from two separate surfaces. When extracting an iso surface using the iso value 0 such a sign change in the field can lead to an undesired surface flow. Examples of this will be presented in Section 7.2.

4.6.3 Smoothing

The use of anisotropic diffusion often leads to local discontinuities in field $\mathcal{N}$ to allow harder edges in the terrain. Furthermore, due to the discrete nature of $\mathcal{N}$ and $\mathcal{B}$, neighbouring entries of these field are never exactly identical unless the target terrain surface is perfectly flat. As a result the surface propagation – in particular the signed distance field method – tends to suffer from noise as it assumes the target surface to be locally flat. To prepare the surface field $\mathcal{S}$ for the final stage of the pipeline, the noise has to be suppressed to a certain degree. This is achieved by simply applying a Gaussian filter on $\mathcal{S}$.

4.6.4 Implementation

Like previous pipeline stages, the surface propagation is implemented using compute shaders. The signed distance field method is implemented as an iterative computation, although in each iteration a different subset of voxel is processed. Each iteration dispatches as many compute shader executions as there are grid cells. The shader decides whether or not it was called for a cell in $\partial\Omega_S$ and continues accordingly. As $\partial\Omega_S$ is in most cases only a small subset of $\mathcal{S}$, this approach is somewhat inefficient. A fixed number if iterations is performed, chosen empirical as $1.5\times$ the highest domain resolution.

The streamline propagation method on the other hand uses only two iterations as mentioned before, again dispatching as many executions of a compute shader as there are grid cells. Each shader execution itself is computationally much more expensive as it traces a curve through the guidance fields if it corresponds to a cell in Ω_S.

Both methods require a backbuffer texture for $\mathcal{S}$ to store the results of an iteration in. In each iteration, source and target texture are switched between the original texture for $\mathcal{S}$ and its backbuffer. Finally, smoothing is performed by an additional compute shader that implements a separated guassian filter.

4.7 Surface Mesh Extraction

The final task of the pipeline is to create a surface representation for efficient rendering. I choose a polygon mesh as the final terrain surface representation as it fits best into the existing rendering infrastructure. Triangle meshes are efficient to render on GPUs and compared to the data fields, the mesh is expected to require less memory. Furthermore, concerning lighting and texturing, many known and well documented techniques can be directly applied to a terrain surface mesh. Alternatively, an efficient rays casting renderer, optimized to working with the fields $\mathcal{S}$, $\mathcal{N}$ and $\mathcal{R}$, could be developed.

Creating the terrain surface mesh is equivalent to computing an iso-surface in $\mathcal{S}$, which is done using the Marching Cubes algorithm. If $\mathcal{S}$ is a signed distance field, the iso-value is set to zero. If $\mathcal{S}$ is unsigned however, the iso-value should be slightly larger than zero. Since the Marching Cubes algorithm is extensively covered in the literature, I will only briefly review it [LC87]. The algorithm processes a scalar field by dividing it into *cubes* that are formed by each group of eight directly adjacent voxels. Given an iso-value, for each cube it is independently checked whether or not a part of the isosurface is located within the cube. If it is, the exact shape of the isosurface within the cube is determined and triangles that model the isosurface are generated accordingly. Given the eight scalar values at a cube's vertices, the total number of existing triangle configurations is limited to 256. Therefore it is easy to just classify the individual cubes and then read the number of required triangles and their configuration from a lookup table. All triangle vertices are located on the cubes edges and their exact position is computed by linear interpolation on the edges based on the iso-value. The final output of the algorithm is a triangle mesh that covers the iso-surface in the whole domain.

4.7.1 Implementation

I choose to implement a variation of Marching Cubes that uses Histogram Pyramids [DZTS08][SEL11]. This method is fast and can implemented memory efficient, which is noteworthy as the GPU memory footprint of the pipeline is already quite large due to the use of various volumetric fields. Because it computes the exact amount of triangles generated by the Marching Cube algorithm before the actual mesh data is generated, it is well suited for the terrain generation pipeline that stores the terrain surface mesh rather than computing it dynamically during rendering.

The computation is split into three steps, each of which is again performed on the GPU using compute shaders. The first step is the initial classification of all cubes. It stores the cube's configuration and number of generated triangles in 3D textures. Next, the Histogram Pyramid needs to be constructed with the texture containing the triangle count as the base level. Building the Histogram Pyramid requires a compute shader dispatch per additional level. Finally, using the total amount of triangles (which is stored in the final level of the pyramid) an empty triangle mesh of fitting size is created. A final compute shader is then executed once per triangle. By traversing the Histogram Pyramid, each shader execution locates its corresponding cube and reads the previously stored configuration in order to generate the triangles for this cube. Both index and vertex buffer of the mesh are bound as SSBOs, so that the mesh data can be written from within the compute shader. The vertices of the terrain mesh contain position, normal vectors and noise parameters. To keep memory `vec4` aligned, position and normal are each written to a `vec4` and the noise parameters are split and written to the fourth entry, respectively. This is also illustrated in Figure 4.3.

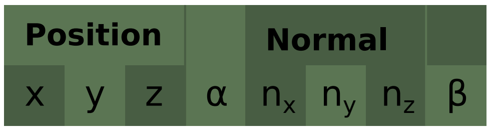

Figure 4.3: Vertex layout of the generated terrain surfaces mesh.

Because this techniques uses a pyramid data structure, the resolution of the Marching Cubes grid needs to be a power of two in each dimension for it to work properly. By extension, the input field of the Marching Cubes computations needs be larger by one in each dimension. The computational grid is therefore restricted to resolutions of $(2^x + 1) \times (2^y + 1) \times (2^z + 1)$.

5 Large Landscapes using Volume Bricking

So far the computational domain is limited to a single connected grid of rectangular shape. This limitation allows for clear definitions and facilitates the understanding of the terrain generation pipeline's core structure. However, due to the properties of the computational grid, all areas in the domain are treated uniformly. In practise this leads to a potential waste of processing power in areas where the terrain surface is not present at all, or which are far away from the virtual camera and thus do not need to be computed at full detail. Especially by taking advantage of adaptive resolution for distant regions, overall larger terrain becomes feasible.

Hence, to increase the flexibility of my terrain generation approach, the domain is subdivided into sub-regions called bricks. Just like the whole domain before, the individual bricks are rectangular, uniform grids. However, the resolution of individual bricks can vary and the resolution of the whole domain is therefore no longer uniform.

The computations performed by the terrain generation pipeline are now distributed over multiple bricks and each brick comes with its own set of the fields $\mathcal{N}$, $\mathcal{B}$, $\mathcal{R}$ and $\mathcal{S}$. The first pipeline stage (the voxelization) remains largely unchanged and is simply performed for all bricks individually. In the following stages however, global solutions (*w.r.t* the whole domain) for the guidance field diffusion, the noise parameter diffusion and the surface propagation are required. If these stages were to be executed for all bricks individually, the influence of a modelling primitive would be limited to only those regions of the domain that are covered by bricks it intersects. To compute global solutions, the bricks have to exchange values with neighbouring bricks at the shared boundaries, meaning that the boundary conditions of computations have to be modified to access values from neighbouring bricks. Furthermore, the exchanged values have to be updated after each iteration, requiring some sort of synchronisation between bricks. As the modified computations are still using shared memory, expansive data transfers are avoided.

One more issue remains and is found in the method used to reconstruct the actual surface mesh. If the mesh reconstruction is simply performed for all bricks individually, a separate terrain surface mesh is constructed for each brick. Recall from the final section

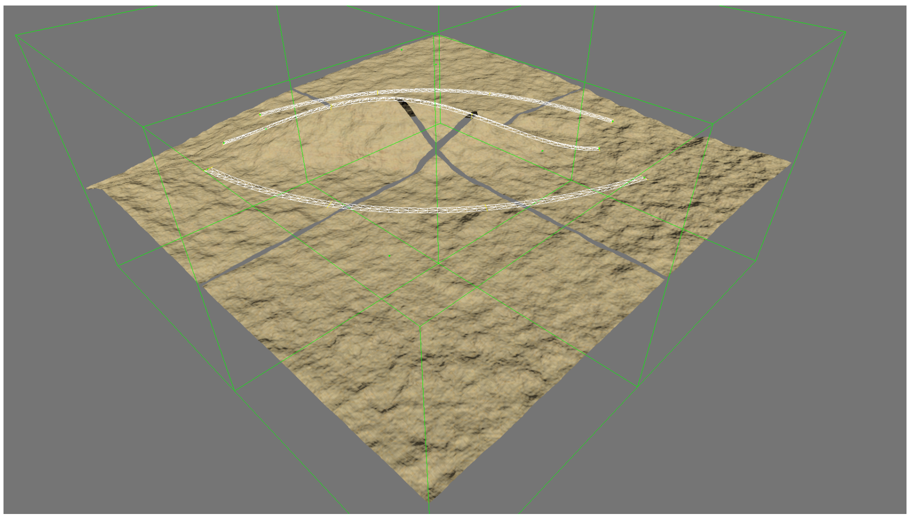

Figure 5.1: A single ridge line spanning over four bricks. Clearly visible gaps remain between the surface meshes of the individual bricks if not filled in explicitly.

of the previous chapter that the marching cube algorithms operates only in between the voxels of $\mathcal{S}$, and therefore a small, empty gap remains at the boundary of the domain. This also applies to the surface reconstruction performed on individual bricks and gaps remains in between neighbouring bricks. As seen in Figure 5.1 the resulting overall landscape is visually subdivided into per brick patches.

These gaps have to be closed. This is possible by inserting additional geometry in between bricks where applicable. To avoid visual artefacts and gaps between the brick surface meshes, the resolution of the inserted geometry has to match the resolution of the brick with the higher resolution. Generating the filler geometry is done by additional Marching Cube computations that are limited to the boundary voxels of the respective bricks.

Implementation

Implementation wise, two major changes are necessary for the distributed computations of several bricks. First of all, all bricks have to take turns in computing the next iteration. As the implementation was done with bricking in mind, each iterations is already designed as a separate compute shader dispatch. Basically, within the iteration loop, I now additionally loop over all bricks. Secondly, the boundary conditions have to be modified. While texture access outside of the domain was previously either clamped manually or taken care of by OpenGL's texture wrapping, I now explicitly set boundary condition textures for each brick boundary. If any texture coordinate falls outside the range of the current brick, the texture of the corresponding neighbour brick is accessed. Because texture access coordinates are normalized, they can be used directly to access neighbouring bricks even if the resolution of the bricks does not match. The wrapping mode of all textures used as boundary condition has to be set to `GL_REPEAT` so that any texture coordinate greater one accesses the beginning of the neighbour texture while texture coordinates below zero access the end of the texture. If a brick has no neighbour in a specific direction, the boundary condition textures for this direction are simply set to the brick's texture themselves. Note that the wrapping mode of the textures of the current brick should be set to `GL_CLAMP_TO_EDGE` for this to work.

Note that to simplify the implementations, each brick currently performs two iterations in a row. This is due to the ping-pong buffering that stores the result of every second iteration in a backbuffer. Neighbouring bricks however can currently only access the primary textures of a brick. Consequently, the values used as boundary conditions are not updated uniformly, introducing a small additional error. I postponed the implementation of proper backbuffer handling in the context of a distributed computation in order to save time.

6 Rendering

The focus of this thesis is the feature based generation of terrain surfaces in a 3D domain and the details of rendering the resulting surface mesh are only a secondary concern. Nevertheless, I will briefly discuss the techniques used for rendering and address possible implications of the terrain generation pipeline on rendering.

Terrain rendering relies on GPU hardware tessellation to increase the geometry density of the final terrain surface mesh. Vector displacement is applied to the tessellated geometry to add medium to small scale surface details that could not be directly expressed by a modelling primitive and the basic surface mesh. Recall that noise generation parameters are embedded in the vertex data of the terrain mesh. Within the tessellation evaluation shader a 3D noise signal is generated ad hoc from these parameters that can vary over the surface. In my implementation, 3D fractal noise is generated by combining several perlin noise functions of different scales [Per85]. The vertices' world position multiplied by the roughness parameter is used as input to the noise function and the computed scalar noise value is then multiplied by the amplitude parameter. The final noise value is used as the magnitude of the vector displacement applied to every vertex along its normal vector. Since the surface normal is obviously modified by displacing vertices, it must be updated accordingly. Note that applying the displacement along the vertex normal is equivalent to applying it along the up-axis in the tangent space of the mesh. In this space the normal vectors of the displaced surface are easily obtained by computing the gradient of the noise function. The obtained normal vectors are conceptually equal to those stored in a tangent space normal map. Thus, by the multiplying it with the inverse tangent space transformation matrix, the world space normal vector of the displaced surface is retrieved. Finally, the tangent space transformation matrix itself is updated before passing it on to the fragment shader where it is required for texture based normal mapping to add the final, material dependent high frequency details to the terrain surface.

The framework that my prototype implementation is integrated in uses physically-based lighting implemented in a deferred rendering pipeline. Surface shading of opaque scene objects is based on the Cook-Torrance specular BRDF combined with a diffuse lambertian BRDF and generally follows the description in [NP]. As such, a diffuse albedo map, a specular color map, a roughness map[1] and a normal map are expected by the surface shader. The textures used in images throughout this thesis are in the public domain[2]. Where necessary, textures were slightly tweaked to provide a better match to my physically based shading model implementation.

An important implication on texturing is that the result of the generation process is a dynamically created mesh of arbitrary shape. Unless texture coordinates are also created dynamically and combined with e.g. a 3D painting tool, one has to rely on either procedural or more generic texturing solutions. Since the output of my terrain generation pipeline is from a rendering point of view mostly the same as that of other volumetric terrain implementations, I can easily apply known texturing techniques to the generated mesh. A simple and popular technique for applying textures to a dynamically created landscape mesh without proper uv-maps is tri-planar projection [Ngu07]. The basic idea is that a total of three textures are associated with the three main coordinate axes and projected along the respective axis onto the object. For each surface point, the axis is chosen that results in the least distortion of the texture. In practise, the texture value for a single surface point is computed as a weighted sum of the textures with weights based on the surface normal. This technique is used for all texture maps required for my shading model implementation.

The final surface textures are composited from two layers, each of which is composed of the full set of surface albedo, specular color, roughness and normal map. Blending between the two layers is based on the global y-coordinate, i.e. the altitude, of a surface point and the slope angle. Such blending functions are often used in heightmap based terrain rendering as a means to model altitude climate zones or to simply apply a second layer on top of the basic textures, e.g a snow layer. However, when applied to volumetric terrain including caves and overhangs this simple blending function leads to awkward texturing such as snow on the ceiling of a cave whereas the cave's walls are clear of snow. One possible remedy to this is to check the surface normal vector against either an effective force vector[3] or simply the world up vector. The basic tri-planar texturing implementation used on several screenshots within this thesis already offers a quite compelling visual result although limited in terms of material variation. Additional texture layers and more complex compositing functions including texture masks for different materials can increase variation while still relying on tri-planar projection but exceeded the scope of this thesis. Raising the visual quality of the landscape by using

[1]Although I hear it confuses artists [Kar], I use an actual roughness map, i.e. a higher value results in a rougher, more diffuse surface.

[2]The textures are from *Stunt Rally 2.3* and have been dedicated to the public domain. (Source: http://opengameart.org/content/terrain-textures-pack-from-stunt-rally-23

[3]For example the gravity of a planet.

more advanced texturing techniques in general, and specifically ones that benefit from the feature based approach, are noted for future work.

Finally, the rendering pipeline is supplemented with a volume rendering pass which in context of terrain rendering is primarily used for debugging purposes. Since fields $\mathcal{N}$, $\mathcal{B}$, $\mathcal{R}$ and $\mathcal{S}$ are stored as 3D textures, we can directly display them as volumetric scene objects, allowing us to debug single stages of the terrain generation pipeline more easily.

7 Results & Disscussion

The results are analysed with regard to the quality of the terrain surface and the performance of the terrain generation pipeline in section 7.2 and 7.3. Quality is first evaluated with a number of isolated, synthetic examples of essential terrain features and subsequently by modelling more complex landscapes. Finally in section 7.4 I share some observation on how to use my 3D modelling tools effectively.

Unless stated otherwise, the images in this chapter are made using the distance field propagation method (see 4.6.2) and a grid resolution of $64 \times 32 \times 64$. I found this method to be more stable and robust, generating reasonable result even from somewhat unreasonable Feature Curve arrangements.

A prototype of the terrain generation method presented in this thesis has been implemented as a proof of concept. It is embedded in my custom built graphics engine framework which handles rendering, scene management and controls. The application is written in C++ and relies on OpenGL (4.3+) for graphics and general purpose computing on the GPU (GPGPU).

7.1 User Interface & Tools

My prototype implementation only features a very basic interface and only the most fundamental toolset. To edit and create Feature Curves and control vertices, the user can freely move the camera in the scene. Figure 7.1 shows the two core tools to interact with Feature Curve modelling primitives: The move gizmo, a tool well known from 3D level editors and modelling software, and the bitangent gizmo. The bitangent gizmo has the shape of a halo and always lies in the plane defined by the tangent vector of the curve at the corresponding Constraint Point, also referred to as the bitangent plane. It shows up once a bitangent constraint (rendered as yellow lines) is selected. By clicking on the gizmo and then dragging the mouse, a new bitangent direction is computed as the vector from the Constraint Point on the curve to the cursor position projected into the bitangent plane. The move gizmo shows up once a a control vertex is selected and allows to move the it on the main coordinate axes. As shown in Figure 7.1, selected entities are highlighted in orange.

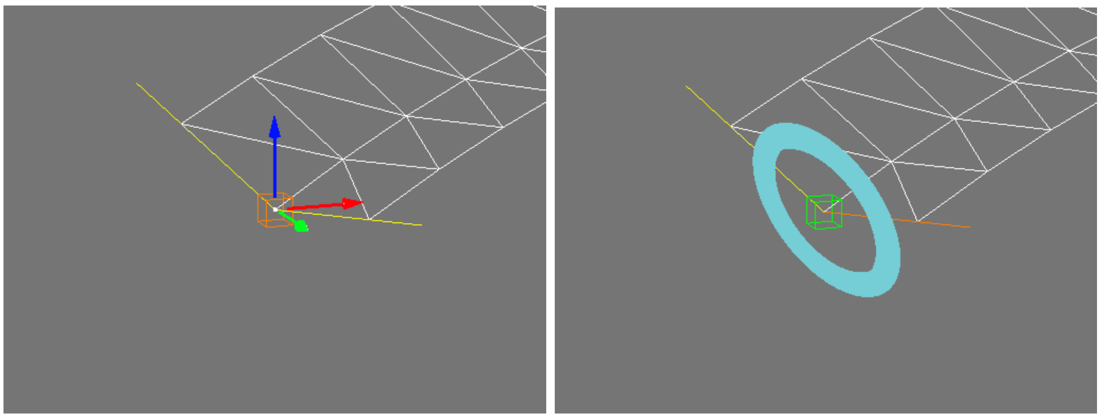

Figure 7.1: The core tools used to edit Feature Curves: A move tool and a tool to edit Constraint Point bitangent vectors.

By interacting with these two tools, the shape of a Feature Curve and its bitangent constraints can be freely manipulated. The tools are supplemented by a number of keyboard shortcuts that allow insertion of additional control vertices at the position of the mouse cursor as well as adjusting the noise parameters stored at constraint points.

The responsiveness of the modelling tools depends primarily on the resolution of the computation grid used in the terrain generation pipeline. In any case, the shape of a Feature Curves is always updated right away. This way the modelling process feels responsive even if the terrain surface update takes a longer time. If the computational grid is small enough, interactive updates of the terrain surface are possible, resulting in a rewarding modelling experience. With interactive updates of the terrain surface, the best control during modelling is achieved as the user is given immediate feedback of his manipulation of a modelling primitive. If interactive updates of the terrain surface are not possible due to a larger computational grid size, I advise to trigger updates of the terrain surface only by explicitly using a keyboard shortcut. Else, the updated terrain will pop up some time after the corresponding modelling input was given, thereby slightly interrupting the user in his current modelling input which most likely no longer directly conforms to the surfaces that was just generated. At the resolution used to create the results presented in this chapter ($64 \times 32 \times 64$), an update of the terrain surface is expected in about one second even on a mobile system. Even though just barely, in practise this is still within the limit of what is perceived as interactive. Exact performance values can be found in Table 7.1 and Table 7.2.

Generally, creation and modelling of Feature Curve is fluid and the creation of landscapes by modelling terrain features feels intuitive. Small landscapes containing several features can be created in a matter of a few minutes. Like Hnaidi *et al.* remark in their work, the basic layout of a terrain can be done in a matter of minutes, but adding additional details can take considerably longer. This observation translates well to my experience of working with my three-dimensional Feature Curves. Modelling of a complex scene with a dozen or more Feature Curves, like the one shown in Figure 7.10 and Figure 7.11, can take from 60 to 90 minutes, depending on the level of experience[1] with the toolset and how much of the scene was decided on beforehand.

7.2 Capabilities & Terrain Quality

I focus the discussion on capabilities of the my method and terrain quality on the generated surface representation, that is the generated mesh data. Consequently, the specific rendering techniques, in particular concerning texturing and lighting, and their impact on terrain quality are mostly ignored. The exception to that is the noise generation as it directly relates to the generated surface mesh.

In order to model diverse landscapes of a high quality, I expect my tools and generation pipeline to allow the modelling of a number of essential terrain features. This includes cliffs (overhanging), ridge lines, river beds, hills, arches and caves. Additional I evaluate the ability to create serpentine roads. For each of these features I assess whether or not my tools and algorithms can successfully recreate them. For the following examples the colouring scheme is set to a uniform base colour with elevation contour lines to better convey the shape of the landscape. Furthermore, noise generation is disabled to concentrate on the generated base geometry.

The first example shown in Figure 7.2 is a simple cliff. As my technique has no concept of a ground level, the cliff continues until it reaches the end of the domain. By tilting the gradient vectors slightly inwards the cliff becomes overhanging. A second curve is necessary to *catch* the cliff face at the desired height and to create a ground plane. Even though the angle between the gradient vectors at each Constraint Point of the Feature Curve in the images is $90°$ and less, the edge of the cliff is still smoothed to a certain degree. This is an inherent property of the generated landscapes due to the diffusion algorithm used to create the guidance fields. Although anisotropic diffusion is employed to counter this effect it still remains a minor issue at the current stage of development and is noticeable throughout most images in this chapter. The perceived smoothing lessens if the resolution of the grid is increased.

The next set of images (Figure 7.3) show a single ridgeline as well as multiple converging ridgelines. Again, additional curves are needed at ground level since otherwise the slope would simply continue downward until the end of the domain. This example also

[1]I could not play around with the final prototype quite as much as I would have liked to.

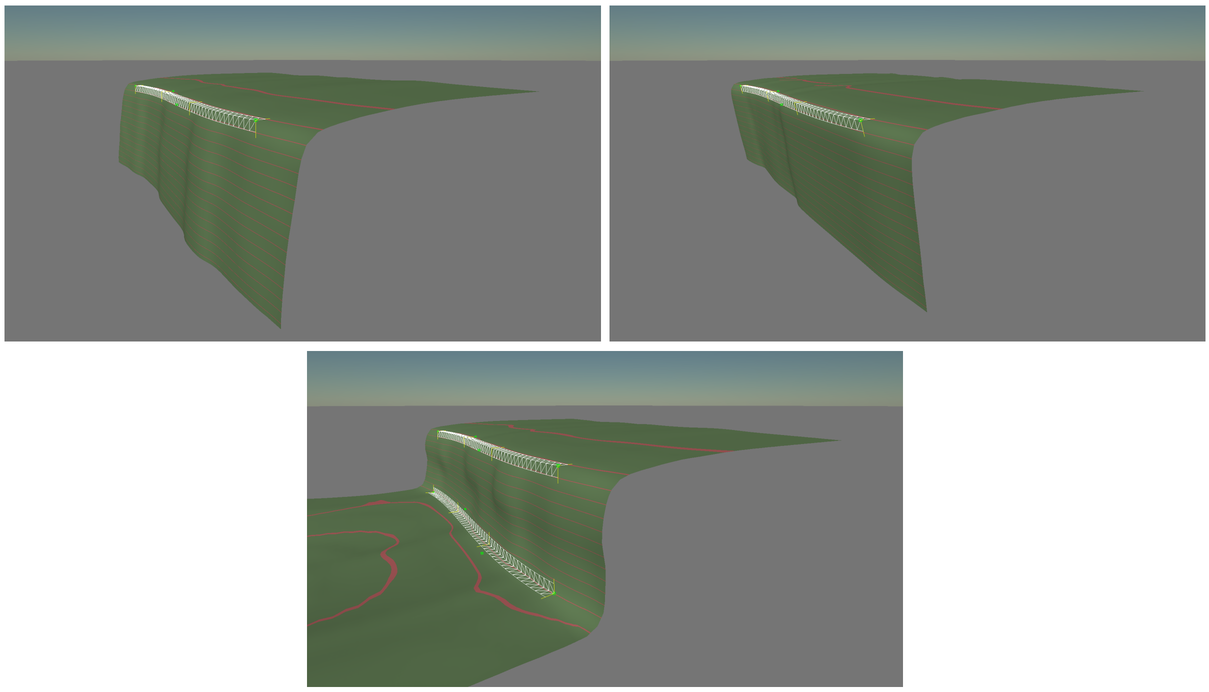

Figure 7.2: A cliff with slight curvature. Upper right image introduces overhang by slightly tilting down facing bitangents. Lower image shows a cliff with ground level achieved by adding a second Feature Curve.

demonstrates how the space between Feature Curves is filled with organic shapes that fit well within the context of smooth landscapes.

A riverbed as seen in Figure 7.4 is modelled similar to a single ridge line using three Feature Curves. Two curves define the river's shoreline and ground level while the third controls the river's depth.

To create a simple hill a single Feature Curves is used to circumscribe the hill's boundary with up-facing gradient vectors on the inside and horizontal gradients on the outside to define the ground level. Depending on the anisotropic smoothness parameter, the hilltop becomes pointed. A second Feature Curve can be used to the alter the shape of the hilltop as desired.

Arches are curved structures that can form in nature through erosion of rock formations. They cannot be represented by a heightfield. Figure 7.6 shows how this kind of terrain feature is achieved with my method by placing three Feature Curves: The first curve defines the overall shape and extent of the arch. A second curve limits the lower extent where the arch spans over ground. As usual, a third curve is required to define the ground level, especially in the area directly below the arch. Like in the previous examples, I observe that my method produces a very smooth mesh.

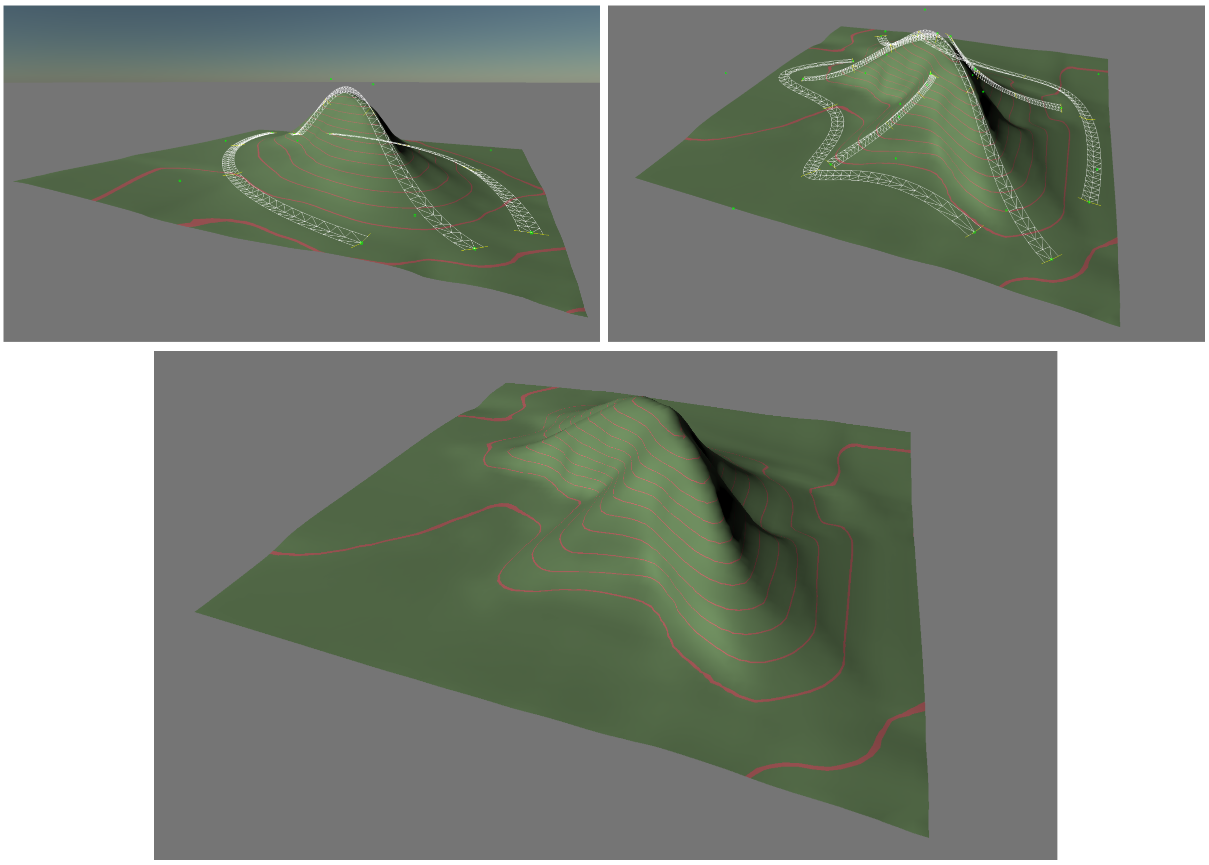

Figure 7.3: Upper left image: A ridge line modelled by a single Curve. The surrounding Feature Curves define the ground level. Upper right image: Multiple converging ridge lines using multiple Feature Curves. Lower image: Same as upper left but with hidden interface.

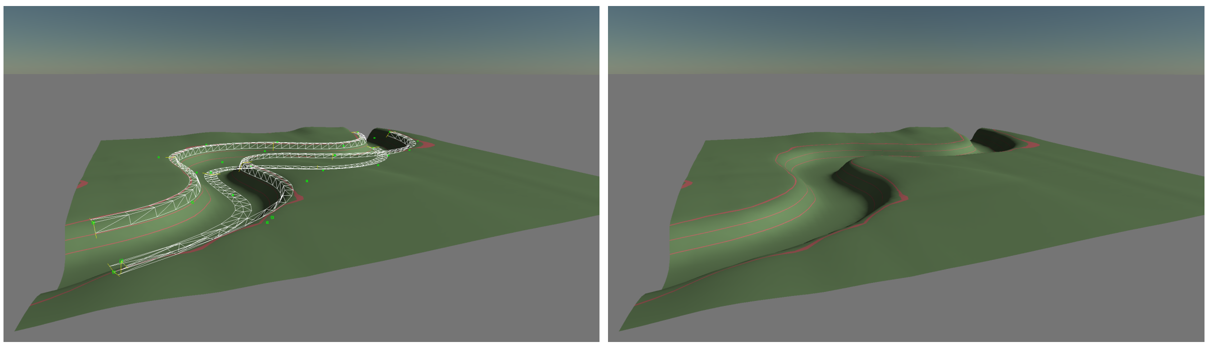

Figure 7.4: A riverbed modelled by three Feature Curves: Two curves define the shoreline, a third the ground of the river.

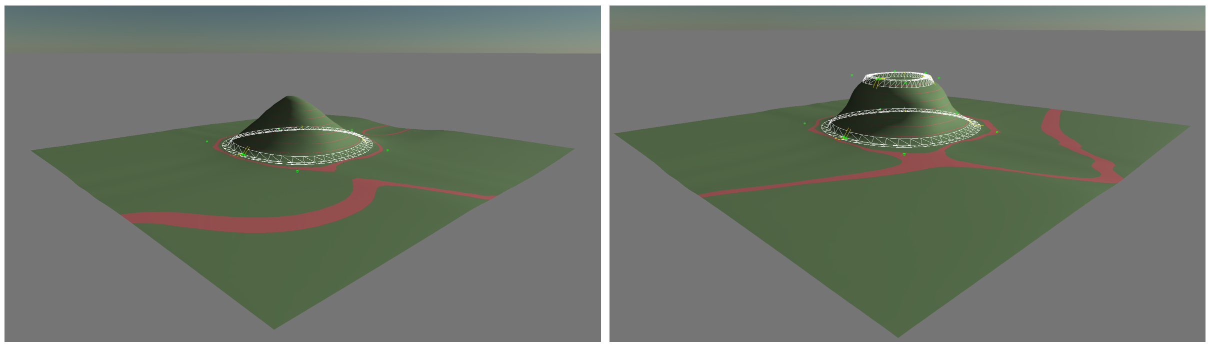

Figure 7.5: A simple hill can created by a single curve. The modify the hill's shape, a second curve is used.

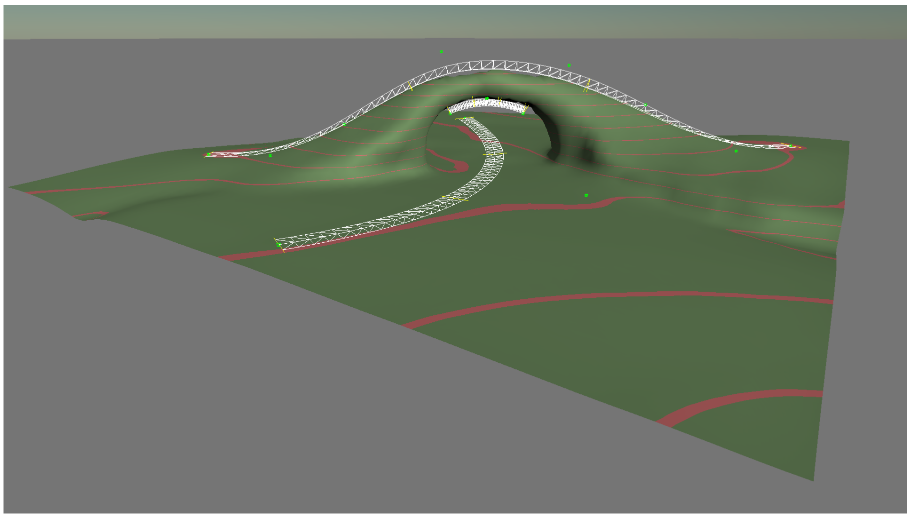

Figure 7.6: An arch, usually formed through erosion of rock formations, is successfully created by my method with only three Feature Curves.

One of the key goals of my technique is to generate complex overhanging terrain and cave structure from very little user input. Figure 7.7 shows a scene with a single, curved cave with a an exit/entry in a rock face as well as a scene with a branching cave. A single curve is required to model the cave entry in the rock face and I used four additional curves to define the course of a cave tunnel. This could be reduced to three curves, but having a dedicated curve for floor, ceiling and both walls feels more intuitive. This arrangement shows that a more complex modelling primitive for common terrain features that require several Feature Curves, just like this tunnel, would be useful. In case of a tunnel, such a modelling primitive could internally place four Feature Curves, as illustrated in the example, and offer additional properties like tunnel width and height to the user.

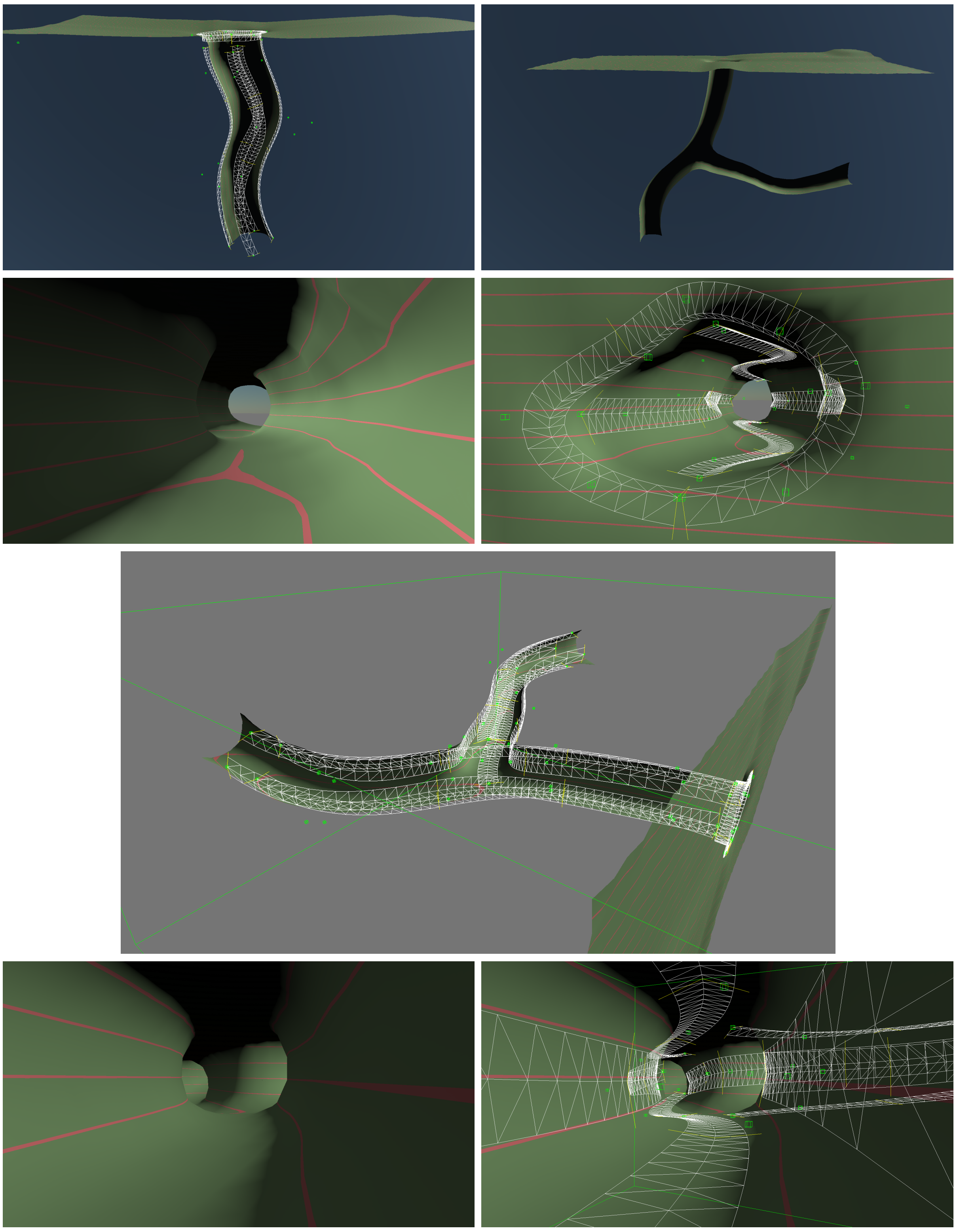

Figure 7.7: Top row: A single and a branching tunnel viewed from below. Second row: Shot and reverse shot of the inside of a single tunnel. Third row: Overview of the branching tunnel. Eight Feature Curves are required to model this scene. Bottom row: Inside of the branching tunnel.

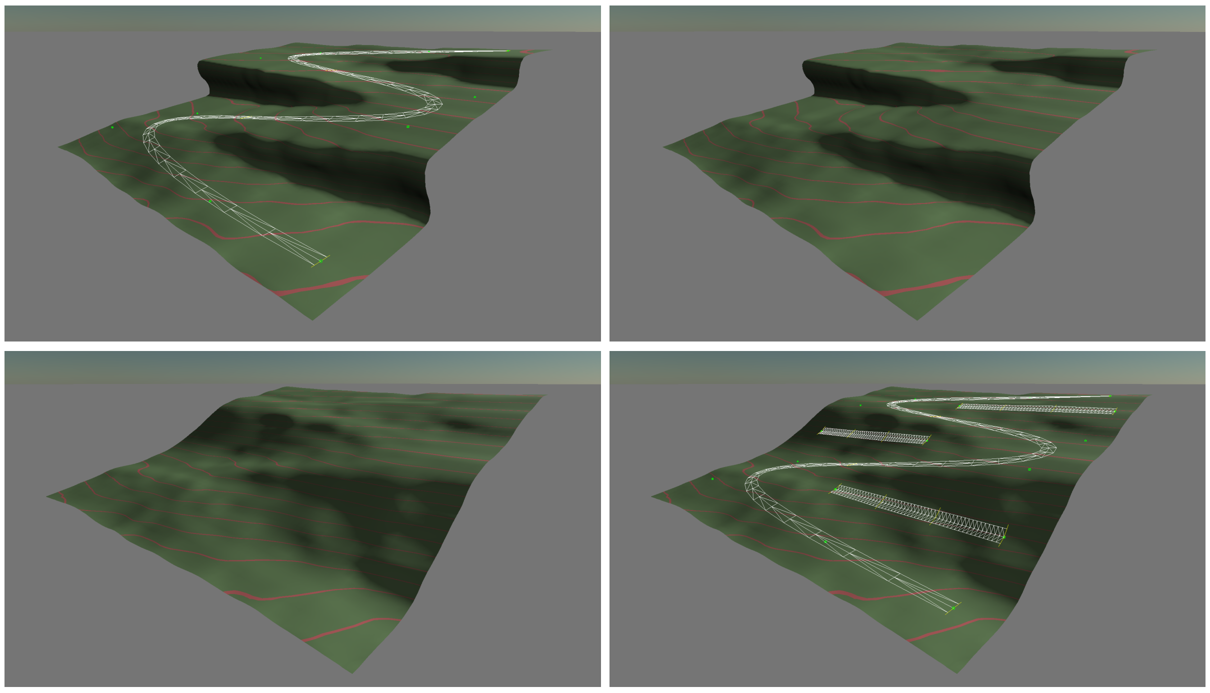

Figure 7.8: Example of a serpentine road modelled by a single Feature Curve. Additional curves can be used to define the space between the road segments.

The final terrain feature I demonstrate is a serpentine road illustrated in Figure 7.8. Spline curves are a natural fit to this kind of man-made structure. If a single spline curve is used to outline the road, the space between adjacent road segments is automatically bridged by a small (overhanging) cliff. By inserting additional Feature Curves, a smoother transition can be achieved.

One can observe that, with noise and texturing disabled, the results presented so far look very smooth and artificial as they lack medium and small scale surface variations. As discussed in chapter 6 noise is applied to the surface to get a more realistic and visually compelling result. Figures 7.9 shows noise being applied to flat base geometry. The noise parameters are given by the Feature Curve spanning across the plane and vary in the left image from $(1.0, 2.0)$ at one endpoint to $(1.0, 15.0)$ at the other where the first value denotes amplitude and the second the roughness value. In the second image, the values vary from (0.0,5.0) to (2.0,5.0). The exact noise parameter values that give plausible results depend on the noise implementation and unit scale. In my case, a distance of one unit theoretically equals a meter in real world measurements[2] but the example landscapes shown here are not always to scale.

[2]Lighting calculations are based on this scale.

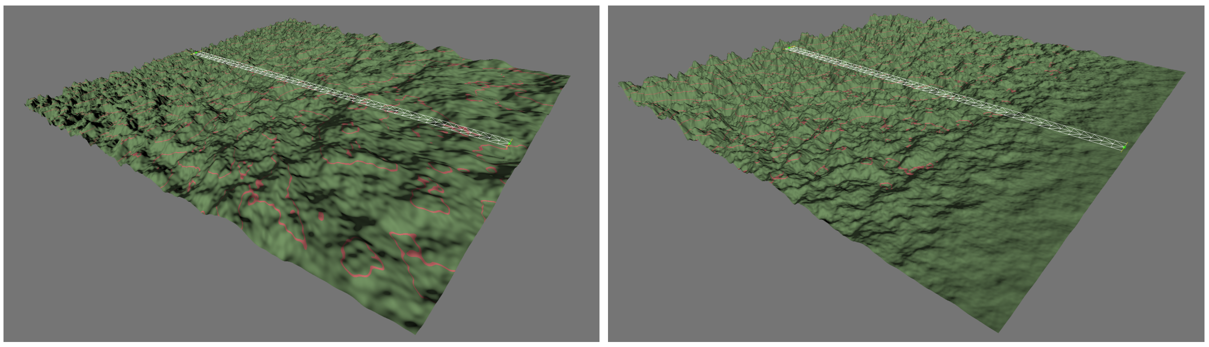

Figure 7.9: Left: Varying roughness at constant amplitude. Right: Varying amplitude at constant roughness.

The full potential of the terrain generation pipeline is unleashed once surface noise and texturing is enabled and thereby missing details are added to the terrain surface. Figures 7.10 and 7.11 show more complex scenes that combine several different terrain features. In Figure 7.10 a large canyon scene is shown that includes features such as an arch, caves, small cliffs and a partial ceiling supported by a pillar. The scene contains nineteen Feature Curves and is computed at a grid resolution of $128 \times 64 \times 128$. A version of the image with Feature Curves displayed is shown in Figure B.1 in the appendix. Figure 7.11 shows a large coastline scene, also generated with a grid resolution of $128 \times 64 \times 128$. Twelve Feature Curves are used in this scene. For a large version of the image without displayed Feature Curves refer to Figure B.3 and Figure B.4 in the appendix.

The technique also has to be analysed with regard to its robustness. For this purpose, the left image in Figure 7.12 shows a scenario where two Feature Curves with partially conflicting constraints overlap. My technique generates a plausible results even though the normal direction defined by the Feature Curves is conflicting. If two modelling primitives with the same normal orientation are placed above one another (*w.r.t* normal direction), an additional, undesired surface will be generated between the primitives due to a sign change in the surface field $\mathcal{S}$. This should be mostly fixed by a small addition to the surface reconstruction implementation that checks if the absolute distance values are actually close to zero.

In areas not directly covered by any modelling primitives, the terrain generation sometimes exhibits unexpected behaviour and produces distorted terrain surfaces. In most cases, this is easily fixed by rearranging the Feature Curves or activating the automatic Feature Curve expansion. Because the latter increases the running time of the pipeline and relies on the biased bitangent field, I currently advise to rearrange and expand curves manually.

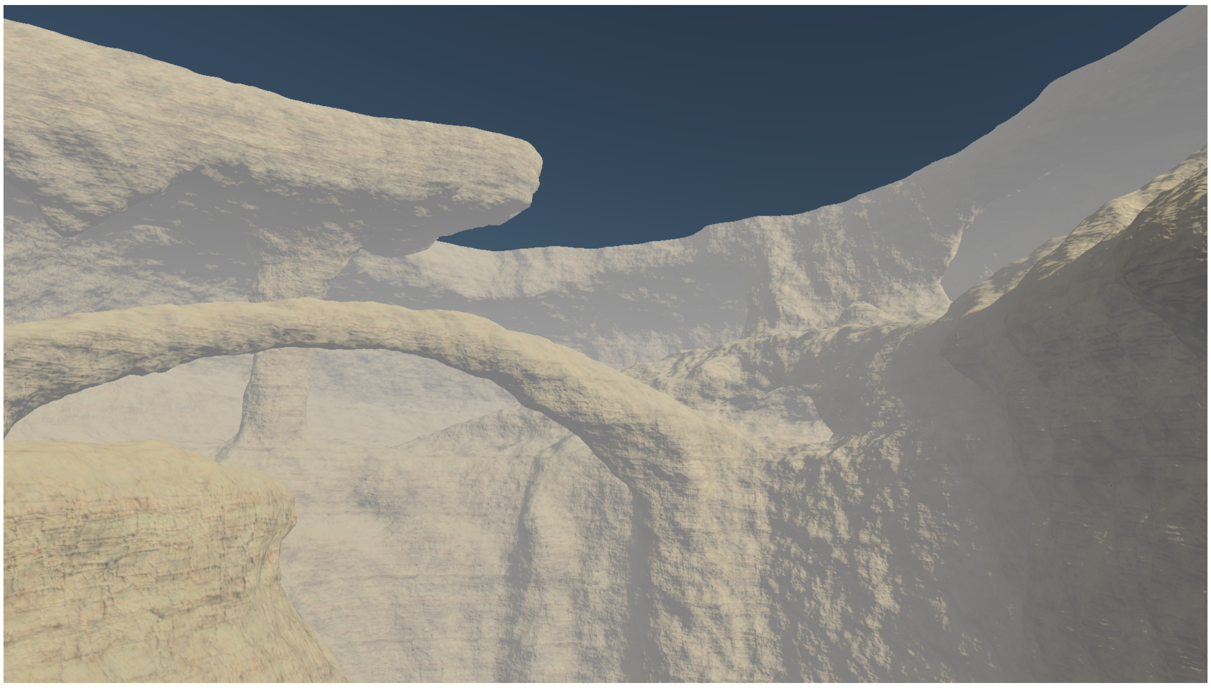

Figure 7.10: A large canyon made from nineteen Feature Curves. Simple ground fog adds an extra level of depth to the scene.

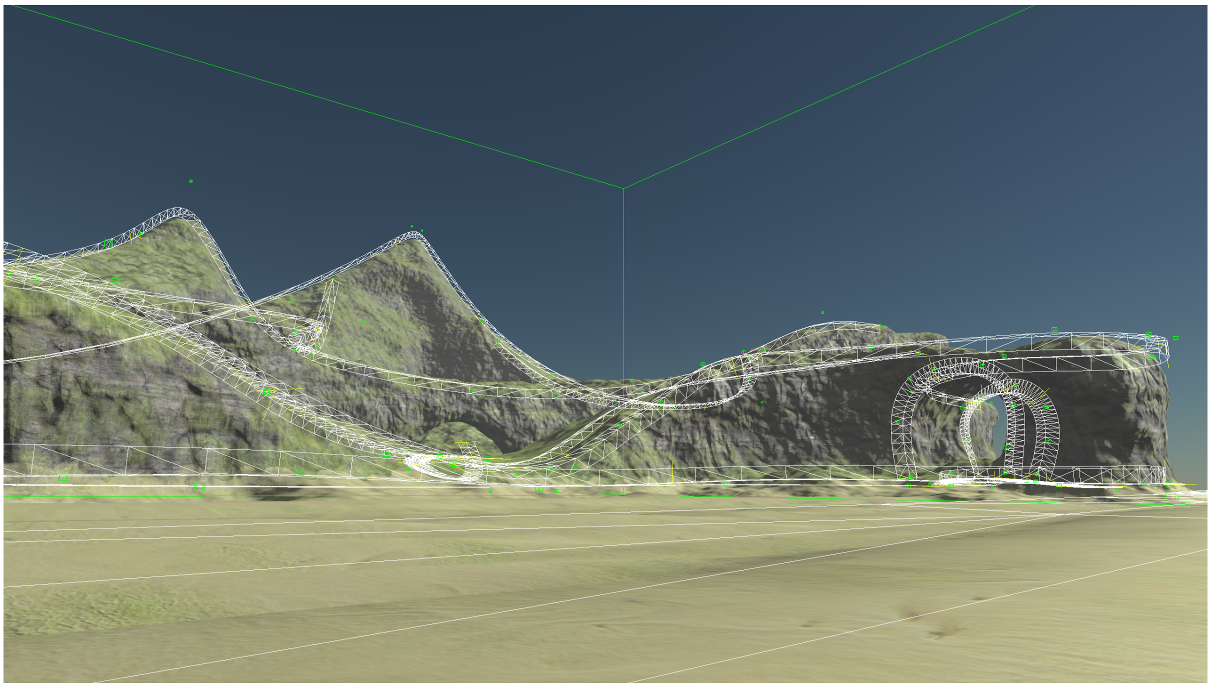

Figure 7.11: A large coastline scene showcasing arches, cliffs and regular mountains. Different texture layers are used for the beach and the inland, as well as varying noise parameters.

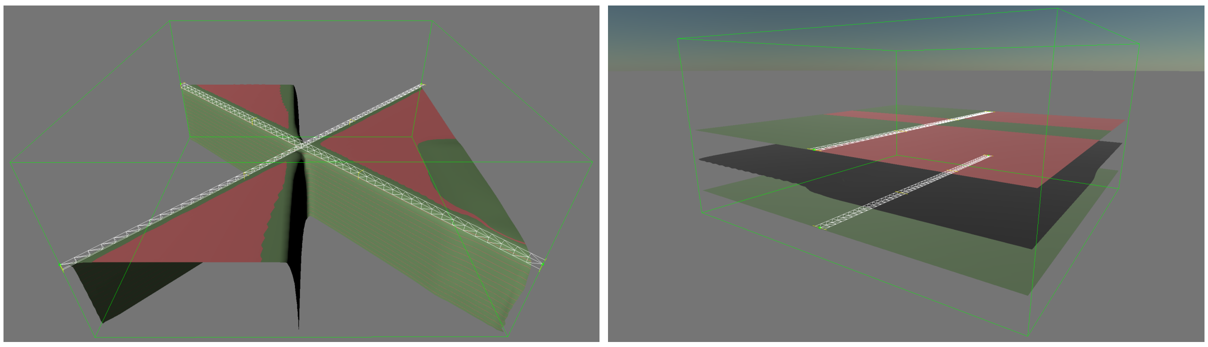

Figure 7.12: Left: A simple example of intersecting Feature Curve with conflicting normal orientation. Right: Undesired surface between Feature Curves.

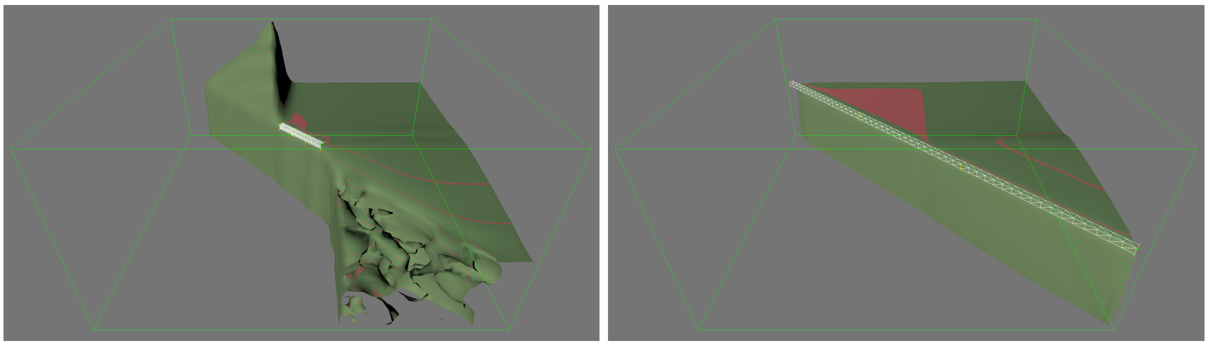

Figure 7.13: Left: A worst case scenario for the method. The Feature Curve is oriented at a $45°$ angle to the computation grid and only covers a small area of the expected terrain feature (a cliff). Right: Such behaviour is easily fixed by increasing the area covered by the Feature Curve to the complete, desired terrain feature.

7.3 Performance

Performance is measured on two different system configurations: A mobile setup with an Intel Core i7 4500U and a Nvidia Geforce GT840M GPU with 2GB dedicated DDR3 VRAM, and a desktop setup consisting of an AMD Phenom II X6 1045T and an AMD HD7870 GPU with 2GB GDDR5 VRAM. Both systems are running Windows 7 64bit.

Naturally, the performance depends to some degree on the modelled scene. Voxelization in particular depends on the number of Feature Curves in the scene and the size of the generated terrain surface has a slight impact on the performance of the surface propagation and generation stage. But in general, the size of the computational grid dominates the performance significantly. Table 7.1 lists the time required for a complete pipeline execution using the the signed distance field propagation method for terrain propagation and without automatic Feature Curve Expansion. Performance measurements are made

	$32 \times 16 \times 32$	$64 \times 32 \times 64$	$128 \times 64 \times 128$
Complete pipeline execution	77.26 ms 28.44 ms	1176.31 ms 355.13 ms	16864.78 ms 5821.66 ms

Table 7.1: Computation time required for a complete execution of the terrain generation pipeline at different grid resolutions. Upper values are the results from the mobile system, lower values are measurements from the desktop system.

	$32 \times 16 \times 32$	$64 \times 32 \times 64$	$128 \times 64 \times 128$
Reset	0.16 ms 0.21 ms	1.13 ms 0.69 ms	8.34 ms 3.9 ms
Voxelization	0.42 ms 0.2 ms	1.83 ms 0.58 ms	12.35 ms 3.1 ms
Guidance field diffusion	30.71 ms 11.54 ms	587.85 ms 163.36 ms	8859.55 ms 2526.69 ms
Noise diffusion	16.06 ms 6.11 ms	279.891 ms 87.79 ms	4235.62 ms 1353.18 ms
Surface propagation	10.9332 ms 4.42 ms	165.64 ms 61.5 ms	2665.57 ms 1627.66 ms
Surface smoothing	1.08 ms 0.37 ms	8.97 ms 2.52 ms	72.3057 ms 19.41 ms
Surface reconstruction	17.9 ms 5.59 ms	131.01 ms 38.68 ms	1011.05 ms 287.72 ms

Table 7.2: Computation times of the individual pipeline stages at different grid resolutions. Upper values are the results from the mobile system, lower values are measurements from the desktop system.

using a scene with a single ridgeline made from three Feature Curves, similar to Figure 7.3. Although not counted as a pipeline stage, resetting the fields $\mathcal{N}$, $\mathcal{B}$, $\mathcal{R}$ and $\mathcal{S}$ to suitable initial values is required before the pipeline execution and is therefore listed in Table 7.2. Smoothing of the surface field $\mathcal{S}$ is also listed separately due to the structure of the implementation.

The increase in execution time with increasing grid resolution matches theoretical exceptions. If the grid resolution is doubled in all dimensions, i.e. increases by factor 8, the number of iterations performed for guidance field diffusion and surface propagation doubles as well. Accordingly, the execution time increases by a factor of 16. From this

	$32 \times 16 \times 32$	$64 \times 32 \times 64$	$128 \times 128 \times 128$
$\mathcal{N}$	0.19 MB	1.57 MB	12.58 MB
$\mathcal{B}$	0.19 MB	1.57 MB	12.58 MB
$\mathcal{R}$	0.07 MB	0.52 MB	4.19 MB
$\mathcal{S}$	0.035 MB	0.26 MB	2.097 MB
Overall	0.59 MB	4.72 MB	37.74 MB

Table 7.3: Memory requirements of the individual fields at different grid resolutions. Not accounting for overhead or backbuffer textures.

observation alone it becomes obvious that larger grid resolutions require optimized algorithms and implementations to allow interactive editing. Although out prototype implementation benefits from the parallelism and raw compute power of modern GPUs, the code is largely unoptimized and performance improvements are expected in future iterations. However, smaller grid sizes up to 64^3 can already be edited interactively even on mobile systems and older desktop hardware. Concerning the surface reconstruction, the result suggests an undiscovered bottleneck in my implementation of Marching Cubes, considering the performance results given in [SEL11] are much faster for comparable grid sizes.

Besides the execution time of the pipeline, the memory consumption of the technique is of interest. My approach stores a 3D texture in GPU memory for each of the fields $\mathcal{N}$, $\mathcal{B}$, $\mathcal{R}$ and $\mathcal{S}$ as well as several backbuffer 3D textures and additional memory for mesh data. Table 7.3 lists the memory required to store the data of each field (not accounting for any overhead). Depending on whether backbuffer textures are only created temporarily as required in the individual stages or have a longer lifetime, the memory consumption for the fields alone might be up to two times as much as given in the table. Technically, all 3D texture used during the pipeline execution can be cleared from memory once the final terrain surfaces mesh is constructed.

7.4 Modelling Guidelines

While working with only a few modelling primitives in a small domain is fairly intuitive, combining many modelling primitives to achieve larger, more appealing landscapes with more complex terrain features may become increasingly challenging. Apart from the shape and placement of individual modelling primitives, the quality of the result and also the overall productivity are influenced by the amount of modelling primitives assembled in the scene. Naturally, if the domain is underconstrained, undesired surface flow is likely to turn up in some areas. If on the other hand the domain is overconstrained,

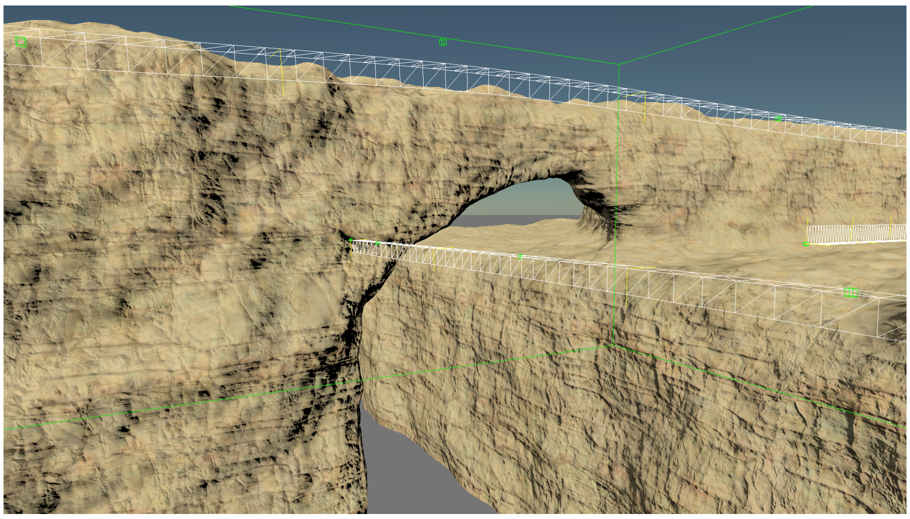

Figure 7.14: Two Feature Curves, each describing a cliff, crossing at a right angle but at different elevation. The transition between the curves is automatically generated in a visually pleasing way.

one is also likely to experience artefacts due to the generation process trying to meet conflicting constraints. Consequently, the key to modelling feature rich, artefact free terrain surfaces with the given tools is to balance the modelling primitive placement.

The first step of modelling a landscape is to identify the major terrain features and be aware of which part exactly of these features is most relevant with regard to the available modelling tools. Sometimes it is fastest to let the generation process automatically fill in gaps instead of trying to force a solution by additional constraints. An example of this is illustrated in Figure 7.14. A visually pleasing transition between the Feature Curves is not directly obvious (from curve placing alone) and placing a Feature Curve in between to explicitly model it might result in a lengthy trial and error workflow.

Keeping the balance between surface modelling primitive and guidance modelling primitives (see beginning of Chapter **??**) requires some experimentation. In general, the influence of guidance modelling primitives is subtler and they yield smoother results. Obviously, at least a single surface modelling primitive should be present in the scene.

Although mentioned before, I would also like to point out one more time, that my method has no concept of a ground level. If a ground level is desired in the level, the user has to explicitly place one or more Feature Curves to specify and outline the location and orientation of the ground plane.

One of the most important guidelines for working with the proposed tools is to pay attention to the distance between modelling primitives – especially between surface modelling primitives. Due to the iterative nature of the guidance field diffusion and surface propagation, information propagates from the modelling primitives like a wave-front through the domain. Therefore interaction between different modelling primitives does not take place at the theoretical intersections of the individual surfaces defined by individual modelling primitives but at locations with equal distance to the primitives. This is effect is strengthened by the anisotropic diffusion of $\mathcal{N}$. Figure 7.15 illustrates the interaction between two surface modelling primitives with perpendicular normal orientation. If the theoretical intersection of the two surfaces falls into a region with equal distance to both curves, the surface merges naturally. Otherwise some sort of crevice will form or the surfaces remain completely separated.

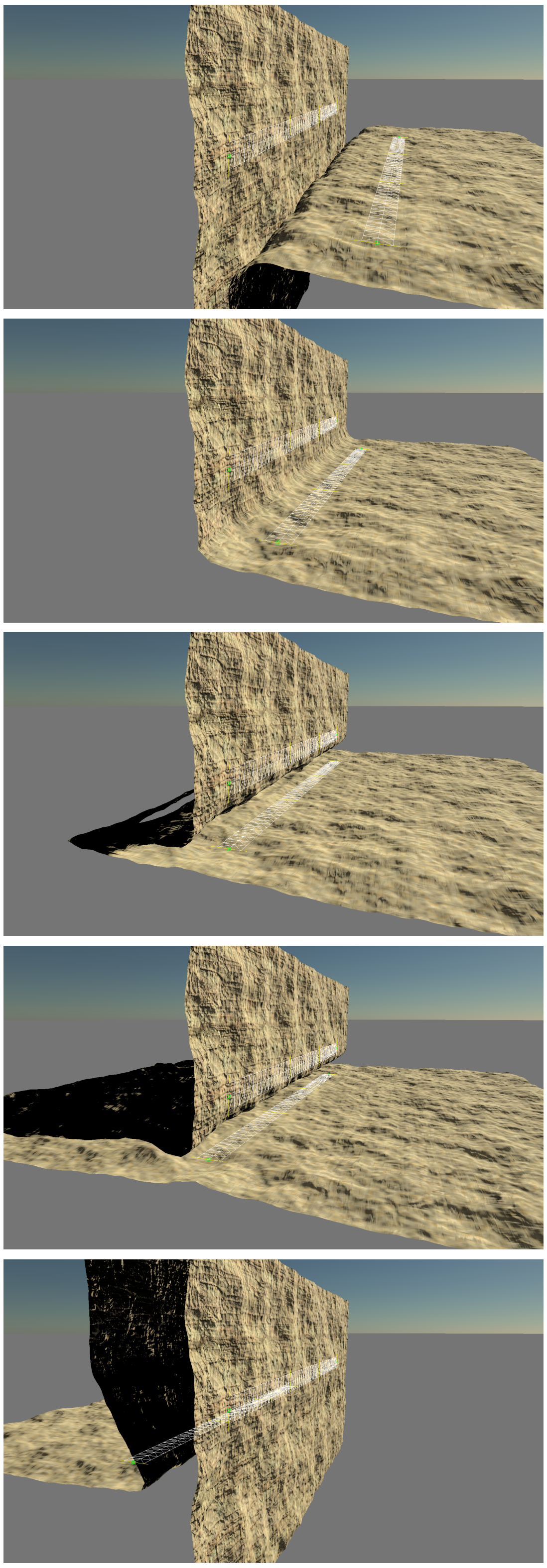

Figure 7.15: Two straight curves with perpendicular normal orientation. From top to bottom, the lower curve is moved below and past the upper curve. Depending on their arrangement, the curves either spawn separate surfaces or share a single surface.

8 Conclusion

In this thesis I proposed a unified approach to modelling and generating almost arbitrary terrain surfaces from sparse user input data, specifically aiming to enable the creation of terrain with multiple vertical levels. The method I developed combines the advantages of a volumetric terrain representation with a primary modelling tool based on three-dimensional, parametric curves that allows fast and efficient editing of large scale terrain features.

I explained how parametric curves are outfitted with the necessary information for defining terrain surfaces. Although parametric curve have previously been used in terrain modelling, I provided a novelty by applying them to a three-dimensional domain. Next, I described a computational pipeline that operates on a three-dimensional domain and combines voxelization, diffusion and iso-surface extraction algorithms to generate a terrain surface mesh from the initial input of Feature Curves. I then explained how the basic pipeline is extended to handle distributed computation in a subdivided domain for the purpose of improved flexibility and ultimately larger computational domains.

The results showed how my method successfully recreates both major terrain features possible to represent with classic heightmaps, as well as complex, vertical terrain features, including overhangs, arches and caves. Additionally, I confirmed that curve-based modelling tools are a good match to terrain features of both natural and man-made origin. Besides the recreation of essential terrain features in isolated scenarios, I proved that my method is capable of generating larger landscapes containing numerous terrain features of different kind. I furthermore showed how medium to small scale detail, which the generated terrain surfaces lacks in its basic form, are added to surface via procedural noise and basic texturing to achieve visually compelling landscape scenes. Finally, even the unoptimized prototype implementation allows interactive editing of at least small regions, making it a suitable method for modelling terrain in real time applications such as game engine level editors.

Future Work

The possibilities for future work are extensive. In order of topics discussed in this thesis they include the following.

Additional terrain constraint properties could be added to the Feature Curves, including for example surface material and vegetation or more complex noise parameters, possibly aimed at geologically motivated noise models. This would reinforce the Features Curves qualification as a unified, multi-purpose terrain modelling tool.

Developing a robust bitangent vector field diffusion technique will require further basic research. Likewise, the presented techniques for anisotropic normal field diffusion and surface propagation will benefit from further research into more accurate models. My implementation of volume bricking could be extended to achieve dynamic level of detail for a moving camera, adjusting the brick resolution as necessary and only generating the terrain surface in a limited area around the camera. As far as rendering is concerned, the generated terrain surface would certainly benefit from advanced texturing, especially in combination with corresponding terrain constraint given by the modelling primitives.

Because the nature of most computationally intense tasks in the terrain generation pipeline is similar or equivalent to solving a differential equation on a discrete domain, many optimization techniques known from research fields such as scientific computing and numerical simulation are applicable to our problem. For example, using a multigrid method to speed up the guidance field and noise field diffusion and possibly the surface propagation seems very promising. Besides the potential performance gain, the coarse to fine approach perfectly fits to the adaptive resolution aimed for in future volume bricking implementations. A multigrid method was not tested in the scope of this thesis, partly due to a lack of time but also since the performance of the current implementation already reaches a level of interactivity that makes optimizations a lower priority. Nevertheless, reasonable optimizations will unlock the gates to larger landscapes and a smoother editing experience.

A Deutsche Zusammenfassung

Zweidimensionale Höhenfelder sind die häufig verwendetste Datenstruktur zum Speichern und Rendern von Landschaften im Bereich des Offline-Renderings und insbesondere der Echtzeit-Computergrafik. Aufgrund seiner Beschaffenheit kann ein 2D Höhenfeld keine Landschaftsstrukturen speichern die meherere vertikale Ebenen enthalten, wie beispielweise Überhänge oder Höhlensysteme. Diese Einschränkung wird aufgehoben, wenn anstelle des Höhenfeldes eine volumetrische Datenstruktur verwendet wird. Jedoch behinhaltet der übliche Arbeitsablauf beim manuellen Modellierien und Bearbeiten von volumetrischen Landschaften vielzählige kleine Bearbeitungsschritte und ist entsprechend zeitaufwändig. In Hinblick darauf schlage ich vor dreidimensionale, kurvenbasierte Primitive zur effizienten Modellierung wesentlicher, großräumiger Landschaftsmerkmale zu verwenden und stelle passende Verfahren vor um eine vollständige Landschaftsoberfläche aus den begrenzten Informationen zu generieren. Durch die Kombination effizienter, merkmalsbasierter Werkzeuge mit einer volumetrischen Landschaftsrepräsentation wird der Arbeitsablauf beschleunigt und vereinfacht, ohne dabei die künstlerische Freiheit, die durch die volumetrische Darstellung gegeben ist, einzuschränken.

B Additional Landscape Images

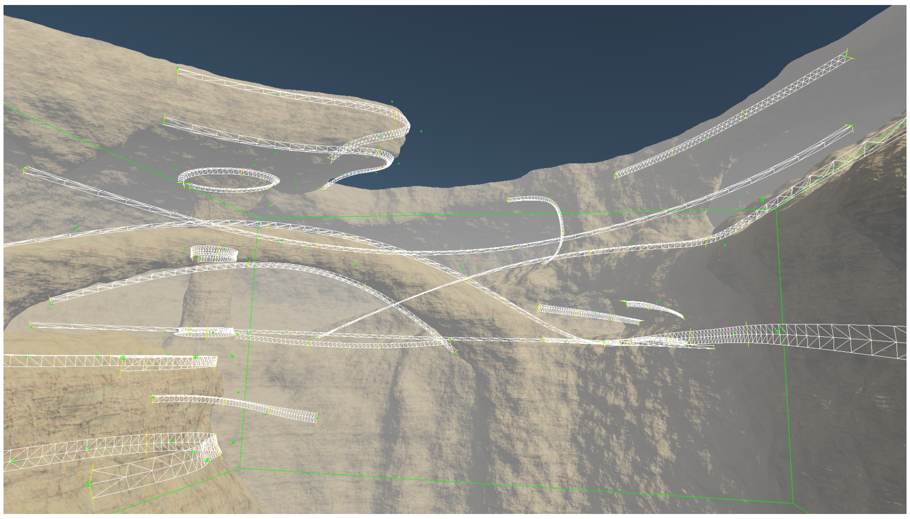

Figure B.1: Canyon scene shown Chapter 7 with displayed Feature Curves.

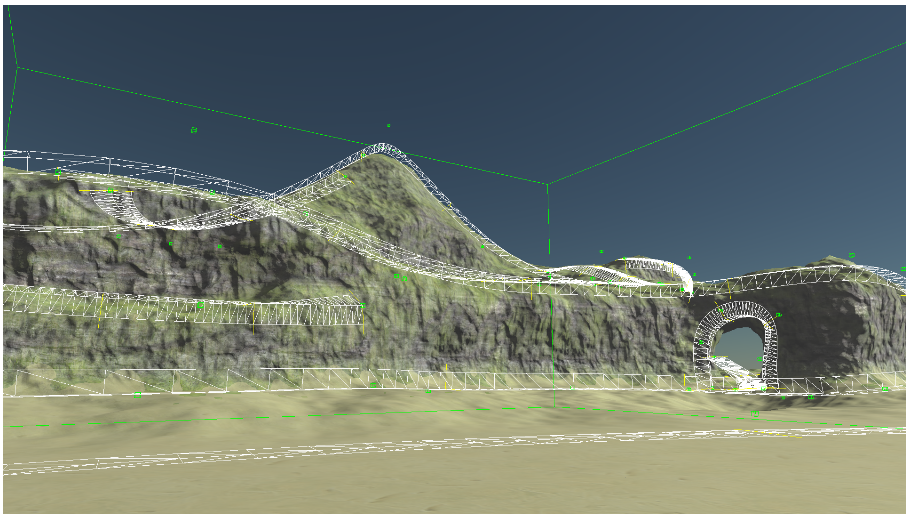

Figure B.2: Coastline scene from the teaser image in the introduction with displayed Feature Curves.

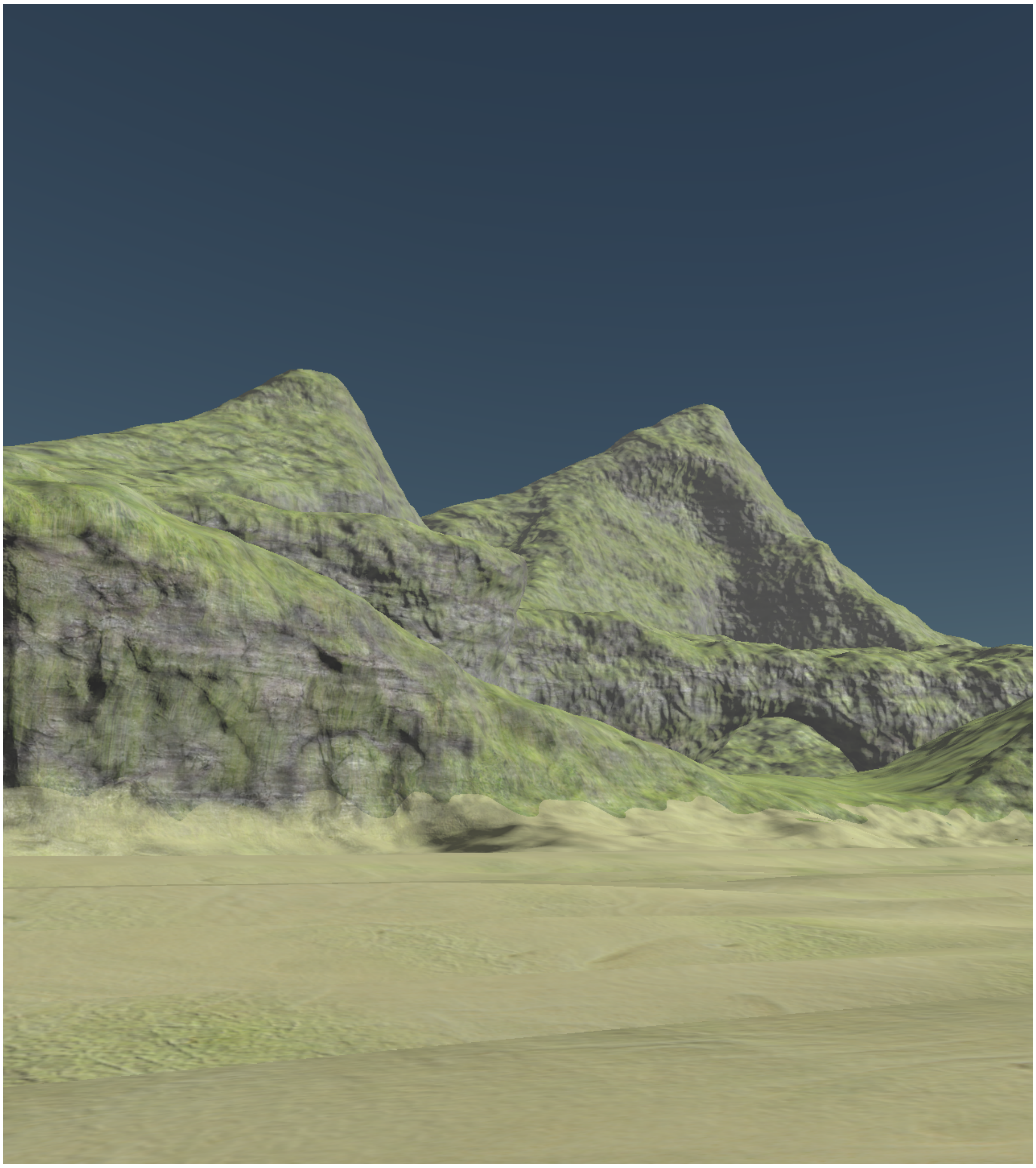

Figure B.3: Large coastline scene from Chapter 7.

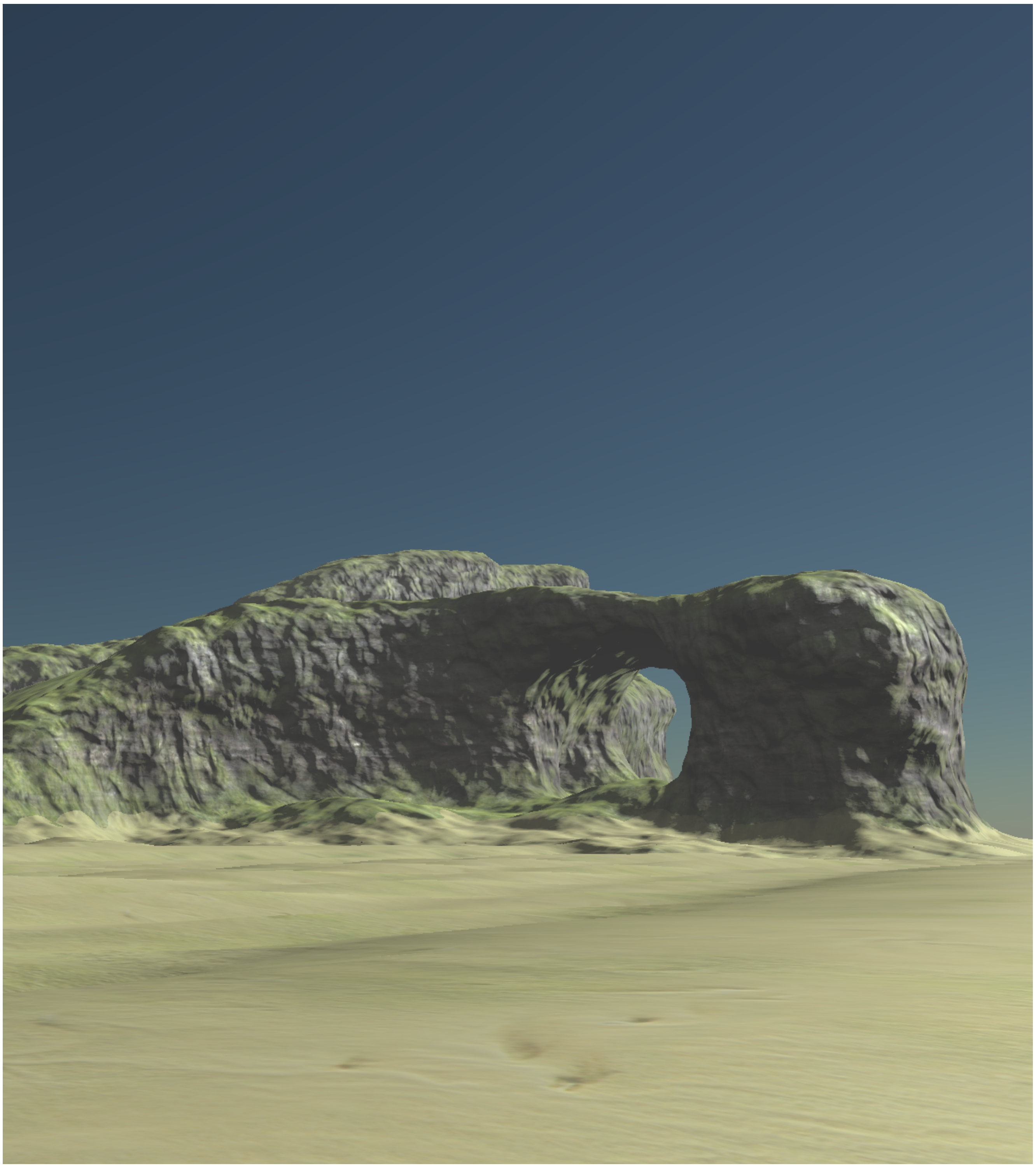

Figure B.4

Bibliography

[Car80] L. C. Carpenter. "Computer Rendering of Fractal Curves and Surfaces." In: *SIGGRAPH Comput. Graph.* 14.3 (July 1980), pp. 109–. URL: http://doi.acm.org/10.1145/965105.807478 (cit. on p. 7).

[CG12] C. Crassin, S. Green. In: *OpenGL Insights*. CRC Press, Patrick Cozzi and Christophe Riccio, July 1, 2012. URL: http://www.seas.upenn.edu/~pcozzi/OpenGLInsights/OpenGLInsights-SparseVoxelization.pdf,%20Chapter%20PDF (cit. on p. 24).

[Crya] Crytek. *CryEngine Road Tool*. URL: http://docs.cryengine.com/display/SDKDOC2/The+Road+Tool (visited on 07/15/2016) (cit. on p. 13).

[Cryb] Crytek. *CryEngine Terrain Editor*. URL: http://docs.cryengine.com/display/CEMANUAL/Terrain+Editor (visited on 07/19/2016) (cit. on p. 7).

[Cryc] Crytek. *The Climb Official Website*. URL: http://www.theclimbgame.com/ (cit. on p. 8).

[DDM+78] C. De Boor, C. De Boor, E.-U. Mathématicien, C. De Boor, C. De Boor. *A practical guide to splines*. Vol. 27. Springer-Verlag New York, 1978 (cit. on p. 14).

[DZTS08] C. Dyken, G. Ziegler, C. Theobalt, H.-P. Seidel. "High-speed Marching Cubes using HistoPyramids." In: *Computer Graphics Forum* 27.8 (2008), pp. 2028–2039 (cit. on p. 41).

[Edw] J. Edwards. *Dynamic Sand Simulation and Rendering in Journey*. SIGGRAPH 2012 Advances in Real-Time Rendering in Games course. URL: http://advances.realtimerendering.com/s2012/ (cit. on p. 8).

[EMP+02] D. S. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, S. Worley. *Texturing and Modeling: A Procedural Approach*. 3rd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002 (cit. on p. 11).

[Epia] Epic Games, Inc. *Unreal Engine 4 Landscape Outdoor Terrain*. URL: https://docs.unrealengine.com/latest/INT/Engine/Landscape/index.html (visited on 07/19/2016) (cit. on p. 7).

[Epib] Epic Games, Inc. *Unreal Engine 4 Landscape Splines*. URL: https://docs.unrealengine.com/latest/INT/Engine/Landscape/Editing/Splines/ (visited on 07/15/2016) (cit. on p. 13).

[Eva98] L. C. Evans. *Partial Differential Equations*. American Mathematical Society, 1998, pp. 20– (cit. on p. 28).

[Fra] Franck Sauer. *Website of an Outcast developer.* URL: http://francksauer.com/index.php/games/test/15-games/published-games/47-outcast-pc (cit. on p. 8).

[Gia] Giantbomb. *Shadow of the Colossus Image Gallery*. URL: http://www.giantbomb.com/shadow-of-the-colossus/3030-6522/images/ (cit. on p. 8).

[GM08] M. N. Gamito, S. C. Maddock. "Localised Topology Correction for Hypertextured Terrains." In: *TPCG*. Citeseer. 2008, pp. 91–98 (cit. on p. 11).

[Hel] Hello Games. *No Man's Sky - Press Image Gallery*. URL: http://no-mans-sky.com/press/sheet.php?p=no_man%27s_sky#images (cit. on p. 8).

[HGA+10] H. Hnaidi, E. Guérin, S. Akkouche, A. Peytavie, E. Galin. "Feature based terrain generation using diffusion equation." en. In: *Computer Graphics Forum (Proceedings of Pacific Graphics)* 29.7 (2010), pp. 2179–2186 (cit. on pp. 11, 13).

[Hug] Hugh Honour, John Fleming. *A World History of Art*. Laurence King Publishing (cit. on p. 7).

[Kar] B. Karis. *Real Shading in Unreal Engine 4*. SIGGRAPH 2013 Course: Physically Based Shading in Theory and Practise. URL: http://blog.selfshadow.com/publications/s2013-shading-course/ (cit. on p. 48).

[LC87] W. E. Lorensen, H. E. Cline. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm." In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '87. New York, NY, USA: ACM, 1987, pp. 163–169. URL: http://doi.acm.org/10.1145/37401.37422 (cit. on p. 41).

[MBMT15] B. Mark, T. Berechet, T. Mahlmann, J. Togelius. "Procedural Generation of 3D Caves for Games on the GPU." In: *Foundations of Digital Games*. 2015 (cit. on p. 11).

[Ngu07] H. Nguyen. *GPU Gems 3*. Addison-Wesley Professional, 2007 (cit. on p. 48).

[NLP+13] M. Natali, E. M. Lidal, J. Parulek, I. Viola, D. Patel. "Modeling terrains and subsurface geology." In: *Proceedings of EuroGraphics 2013 State of the Art Reports (STARs)* (2013), pp. 155–173 (cit. on p. 11).

[NP] D. Neubelt, M. Pattineo. *Crafting a Next-Gen Material Pipeline for The ORder: 1886*. SIGGRAPH 2013 Course: Physically Based Shading in Theory and Practise. URL: http://blog.selfshadow.com/publications/s2013-shading-course/ (cit. on p. 47).

[Per85] K. Perlin. “An image synthesizer.” In: *ACM Siggraph Computer Graphics* 19.3 (1985), pp. 287–296 (cit. on p. 47).

[PGGM09] A. Peytavie, E. Galin, J. Grosjean, S. Merillou. “Arches: a Framework for Modeling Complex Terrains.” In: *Computer Graphics Forum* 28.2 (2009), pp. 457–467. URL: http://dx.doi.org/10.1111/j.1467-8659.2009.01385.x (cit. on p. 11).

[PM90] P. Perona, J. Malik. “Scale-Space and Edge Detection Using Anisotropic Diffusion.” In: *IEEE Trans. Pattern Anal. Mach. Intell.* 12.7 (July 1990), pp. 629–639. URL: http://dx.doi.org/10.1109/34.56205 (cit. on pp. 29, 30).

[SBD13] M. Scholz, J. Bender, C. Dachsbacher. “Level of Detail for Real-Time Volumetric Terrain Rendering.” In: *VMV*. Citeseer. 2013, pp. 211–218 (cit. on p. 12).

[SDT+09] R. M. Smelik, K. J. De Kraker, T. Tutenel, R. Bidarra, S. A. Groenewegen. “A survey of procedural methods for terrain modelling.” In: *Proceedings of the CASA Workshop on 3D Advanced Media In Gaming And Simulation (3AMIGAS)*. 2009, pp. 25–34 (cit. on p. 11).

[SEL11] E. Smistad, A. C. Elster, F. Lindseth. “Fast surface extraction and visualization of medical images using OpenCL and GPUs.” In: *The Joint Workshop on High Performance and Distributed Computing for Medical Imaging* 2011 (2011) (cit. on pp. 41, 63).

[Uni] Unity Technologies. *Unity Manual - Terrain Engine*. URL: https://docs.unity3d.com/Manual/script-Terrain.html (visited on 07/19/2016) (cit. on p. 7).

[XP97] C. Xu, J. L. Prince. “Gradient vector flow: A new external force for snakes.” In: *Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on*. IEEE. 1997, pp. 66–71 (cit. on p. 28).

[YHGT10] J. C. Yang, J. Hensley, H. Grün, N. Thibieroz. “Real-Time Concurrent Linked List Construction on the GPU.” In: *Computer Graphics Forum*. Vol. 29. 4. Wiley Online Library. 2010, pp. 1297–1304 (cit. on p. 27).

All links were last followed on July 28, 2016.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

---

place, date, signature
Ort, Datum, Unterschrift