REPRESENTATION, EDITING AND REAL-TIME VISUALIZATION OF COMPLEX 3D TERRAINS

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Çetin Koca

September, 2012

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Uğur Güdükbay (Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. İbrahim Körpeoğlu

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Ahmet Enis Çetin

Approved for the Graduate School of Engineering and Science:

Prof. Dr. Levent Onural
Director of the Graduate School

ABSTRACT

REPRESENTATION, EDITING AND REAL-TIME VISUALIZATION OF COMPLEX 3D TERRAINS

Çetin Koca
M.S. in Computer Engineering
Supervisor: Assoc. Prof. Dr. Uğur Güdükbay
September, 2012

Terrain rendering is a crucial part of many real-time computer graphics applications such as video games and visual simulations. It provides the main frame-of-reference for the observer and constitutes the basis of an imaginary or simulated world that encases the observer. Storing and rendering terrain models in real-time applications usually require a specialized approach due to the sheer magnitude of data available and the level of detail demanded. The easiest way to process and visualize such large amounts of data in real-time is to constrain the terrain model in several ways. This process of regularization decreases the amount of data to be processed and also the amount of processing power needed at the cost of expressivity and the ability to create interesting terrains.

The most popular terrain representation, by far, used by modern real-time graphics applications is a regular 2D grid where the vertices are displaced in a third dimension by a displacement map, conventionally called a height map. It is the simplest and fastest possible terrain representation, but it is not possible to represent complex terrain models that include interesting terrain features such as caves, overhangs, cliffs and arches using a simple 2D grid and a height map. We propose a novel terrain representation combining the voxel and height map approaches that is expressive enough to allow creating complex terrains with caves, overhangs, cliffs and arches, and efficient enough to allow terrain editing, deformations and rendering in real-time. We also explore how to apply lighting, texturing, shadowing and level-of-detail to the proposed terrain representation.

Keywords: Terrain representation, terrain visualization, caves, overhangs, cliffs, voxel terrain, height map terrain.

ÖZET

KARMAŞIK 3B ARAZİLERİN GERÇEK-ZAMANLI OLARAK SİMGELENMESİ, DÜZENLENMESİ VE GÖRSELLEŞTİRİLMESİ

Çetin Koca
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Assoc. Prof. Dr. Uğur Güdükbay
Eylül, 2012

Arazi görselleştirme, bilgisayar oyunları ve görsel benzetimler gibi gerçek-zamanlı bilgisayar grafikleri uygulamalarının çok önemli bir parçasıdır. Arazi görselleştirme, izleyiciye temel bir referans-çerçevesi sağlamasının yanında izleyiciyi saran hayali veya benzetimli dünyanın temelini oluşturur. Gerçek-zamanlı uygulamalarda arazi modellerinin saklanması ve görselleştirmesi genellikle veri boyutu büyüklüğü ve talep edilen detay seviyesi nedeniyle bu iş için özelleşmiş bir yaklaşım gerektirir. Böyle büyük boyutlu verilerin gerçek-zamanlı olarak işlenmesi ve görselleştirilmesi için izlenecek en kolay yol arazi modellerinin birçok yönden kısıtlanmasıdır. Bu kalıba uydurma işlemi, arazi modellerinin ifade edilebilirliğinin ve ilginç arazi modelleri yaratma olanaklarının kısıtlanması pahasına işlenmesi gereken veri boyutunu ve ihtiyaç duyulan işlem gücünü azaltır.

Çağdaş gerçek-zamanlı grafik uygulamaları tarafından en çok kullanılan arazi gösterimi yükseklik haritasının üçüncü boyutta uygulandığı düzenli bir 2B ızgaradır. Bu gösterim, mümkün olan en basit ve hızlı işlem olanağı sağlayan arazi gösterimidir; ancak mağara, asılı kaya, uçurum ve kemer gibi ilginç arazi özelliklerinin temsil edilmesine olanak sunmaz. Biz, hacimsel gösterim ve yükseklik haritası yaklaşımlarını birleştiren, mağara, asılı kaya, uçurum ve kemer gibi arazi özelliklerini içeren karmaşık arazi modellerini temsil edebilecek ifade yeneğine sahip ve gerçek-zamanlı olarak arazi düzenleme, şekil değişikliği ve görselleştirmesine izin verecek kadar verimli yeni bir arazi gösterimi öneriyoruz. Aynı zamanda, önerilen arazi gösterimine ışıklandırma, kaplama, gölgelendirme ve detay seviyesi belirleme işlemlerinin nasıl uygulanabileceğini inceliyoruz.

Anahtar sözcükler: Arazi modeli simgeleme, arazi görselleştirmesi, mağaralar, asılı kayalar, kayalıklar, hacimsel arazi, arazi yükseklik haritası.

Acknowledgement

First and foremost, I want to thank my dear family for providing me an environment in which I could follow my passion and develop my skills. This thesis could not have been possible without the love, care and support of my mother and father. I am grateful to my dear brother for always being one of my best friends, and to my beloved sister for making life more fun and enjoyable with her jokes and smile.

I want to thank my friends Süleyman Fatih İşler and Fatih Karakuş for always being there for me, sharing the happiest and the most stressful moments of my life, and lending me a hand and a mind whenever I was in need.

I feel very lucky for having a chance to work with my advisor Uğur Güdükbay as I learned so much from him within his lectures and during my thesis studies. I am grateful to him for his support and encouragement through the course of my M.Sc. studies, and for his sincere patience and understanding even when I was not able to devote the time deserved by my studies. He always valued my ideas and endeavors, sometimes even more than me, and shed light on my path with his vision. I also want to thank the members of the thesis jury, İbrahim Körpeoğlu and Ahmet Enis Çetin for evaluating the thesis and providing their invaluable feedback.

I want to express my sincere gratitude to ASELSAN Inc., the company that I am proud to be a part of, especially to my team leader, Fikri Dikmen and each and every one of my teammates for their support and understanding during my studies.

I want to thank David Arkenstone, Ludovico Einaudi, Debbie Wiseman, Thomas Newman, James Horner, Hans Zimmer, Clint Mansell, Iron Maiden, Nightwish, Korpiklaani, and many other great musicians and composers for always providing inspiration to me, refreshing my soul, and making the world a much better and delightful place to live with their unique tunes.

Finally, I would like to thank TÜBİTAK (The Scientific and Technological Research Council of Turkey) for valuing my ideas and financially supporting my M.Sc. studies through their BİDEB scholarship.

Dedicated to my loving parents...

Happy birthday mom, I love you!

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introduction

Terrain rendering has been one of the most popular topics of computer graphics for a long time. It usually provides a canvas to the visualization on which other details are added, such as vegetation, ponds, artificial structures, animated characters, and vehicles. Terrain is, therefore, probably the most important and influential element of such visualizations that require outdoor rendering of rural areas.

Terrain rendering in real-time is a yet more interesting topic, and a unique experience for the user as it allows roaming or flying around the terrain freely at interactive speeds. Real-time terrain rendering is one of the most important elements of virtual worlds. It has a broad application area covering education, mapping, navigation, military strategic planning, simulation training, motion picture, and last, but definitely not least, video games.

1.1 Overview

Real-time terrain rendering is a very challenging task. A real-time rendering application must be able to execute its per-frame processing and the rendering pipeline at a rate of at least 30 frames per second. This means that the algorithms that run during the simulation have at most 30 milliseconds to perform their task for each frame. Furthermore, the entire processing power is usually not reserved solely for terrain rendering purposes since a real-time simulation, e.g., a video game, have much more processing to do per-frame, such as collision detection, artificial intelligence, networking, and managing and rendering elements of the virtual world other than the terrain. Consequently, the algorithms that operate on and render terrain data in real-time are expected to do a huge amount of work, with limited processing power, and within a very strictly limited time frame.

The terrain data, even a moderately large and detailed one, is almost always too large to be processed by simple brute-force algorithms in real-time. Representing and rendering terrain models in real-time applications usually require a specialized set of algorithms due to the sheer magnitude of data available and the level of detail demanded. It is a common approach to significantly constraint the terrain representation so that it can be stored in a memory-efficient way, and very simple algorithms can be used to operate on data, yielding high performance. The more constrained the terrain representation is the simpler the data structures and the algorithms are, and the easier it is to process and render it. This benefit comes at a cost, though, as it decreases the expressive power of the terrain representation.

The most popular terrain representation used for real-time rendering is based on heightmaps. Heightmap-based approaches usually define the terrain surface as a regular, uniform grid of vertices on a 2D plane. Each vertex on the plane is then displaced along the height-axis according to the height value retrieved from the heightmap for that vertex. There are variations of these approaches where a non-uniform grid of vertices is used to approximate the terrain surface. These approaches, however, still use heightmaps to represent the terrain surface. This representation basically samples the terrain from a top-down view and, consequently, is not able to represent volumetric terrain features such as caves, overhangs and arches. Even truly vertical cliffs cannot be represented with heightmaps since two vertices cannot be at the same position when viewed from the top. In spite of these limitations, heightmap-based approaches are still the most popular ones used in the industry, e.g., in almost all video games. This is mainly due to the lack of different terrain representations that can relax the constraints of the heightmap-based approaches, and of different algorithms that can operate on these representations and render them efficiently in real-time.

Voxel-based volumetric representations are commonly used for offline rendering, where performance and efficiency is only of secondary importance. Volumetric terrain representations are inherently able to represent all kinds of volumetric features of a terrain since they sample a true 3D space. Volumetric representations have their own set of problems, though. Rendering methods that can directly render volumetric data, such as ray tracing, are still very slow for real-time high resolution rendering of large and detailed terrains. Modern GPUs are designed to render polygonal surfaces efficiently. Many real-time rendering applications that use volumetric representations extract a polygonal surface of the volumetric representation for rendering purposes to be able to use hardware-accelerated rendering. This incurs an extra overhead to the approaches that are based on volumetric representations. Heightmaps sample the space in 2D, whereas voxels sample the space in 3D, meaning that the size of the voxel representation of a terrain is several orders-of-magnitude larger than that of a heightmap representation. Volumetric representations usually need to be compressed for memory efficiency which incurs yet another penalty on performance due to decompression of terrain data in real-time. It is also difficult to use smoothly varying level-of-detail techniques with voxel representations. Due to these difficulties of working with volumetric terrain models in real-time, they are very rarely adopted.

1.2 Motivation

Even though real-time terrain rendering has been a hot topic for a long time, the majority of the existing approaches are designed to render elevation data, e.g., heightmaps. Volumetric features, such as caves, overhangs and arches are not supported by these approaches. In many real-time terrain rendering applications these volumetric features are modeled as separate 3D meshes, due to technical constraints, despite the fact that they are actually parts of the terrain model. These 3D meshes are then located on the relevant sections of the terrain, just like any other object in the world. These 3D meshes that are modeled separately do not seamlessly blend with the terrain, though. Several tricks are used to conceal the artifacts, such as placing rocks at the entrance of a cave to conceal the artifacts that occur in the regions where the actual terrain model and the 3D cave mesh meet. Such hackish approaches also constraint the size and the detail level of the volumetric features of the terrain as they now must be built as separate 3D meshes and cannot use optimizations specific to terrain models.

Dedicated graphics hardware have become much more common and powerful recently, and the quality of real-time renderings have greatly increased. Even though the surface of the terrains can now be rendered in a much higher level of detail now than ever, the complexity of the terrain models have not shown the same sort of improvement due to the limitations of the traditional approaches. We wish to propose a new terrain representation for real-time rendering that can represent not only elevation data but also volumetric terrain features in a unified way. We aspire that new terrain representations can be used in real-time applications to create and visualize more interesting terrains than ever for a unique experience in demanding applications of today, such as virtual worlds in massively multi-player online games.

1.3 Challenges

Designing a new terrain representation for real-time rendering that supports volumetric features is a challenging task. Some form of volumetric representation must be adopted to be able to represent volumetric features. High quality real-time rendering, on the other hand, is currently only possible with the use of hardware-accelerated rendering techniques which are not designed for volumetric rendering and can only efficiently work with polygonal surfaces. Hence, the volumetric representation must be converted to a polygonal surface for rendering.

A terrain representation usually needs to be bundled with a whole set of algorithms for real-time rendering. There are many simple and efficient algorithms designed to operate on heightmaps. These are, however, not directly usable with any other representation. Therefore, a new terrain representation must either be designed to benefit from the existing algorithms or must come up with a whole new set of algorithms to solve problems of real-time rendering.

The memory-usage and performance characteristics of the terrain representation and the algorithms designed to operate on that are also extremely important. A large terrain with sufficient detail requires high amounts of data to represent the terrain. Furthermore, unlike most other 3D objects, a terrain is usually rendered very close to the surface as the observer walks on the terrain, and also the sections of the terrain that are very far away are still visible to the observer. This requires some kind of level-of-detail approach that can be used with the terrain representation for efficient rendering without significantly decreasing the visual quality. Visualization approaches that are usable with the terrain representation must also be developed, such as approaches for lighting, texturing, and shadowing.

1.4 Research Goals

The design of the terrain representation, the algorithms that operate on it and the rendering pipeline used to render the terrain were influenced by the following design goals:

- The terrain representation must be able to represent volumetric terrain features such as caves, overhangs, arches, and vertical cliffs.
- Rendering performance of the terrain must be sufficient for real-time rendering including each and every step required to render a complete virtual world, such as lighting, textures, and shadows.
- The entire terrain surface must be represented in terms of smaller logical parts. This enables the algorithms to be more efficient as they are able to work at a level higher than that of primitives such as vertices and triangles. The advantage is that each part can be rendered independently, the data can be processed in chunks, culling algorithms can be used to discard redundant parts in a view-dependent way and level-of-detail management can be performed per chunks of primitives, rather than independently for each primitive.
- The terrain representation must allow editing and deformation of the terrain in real-time. Editing the terrain surface must only cause local changes. This is desired for easy creation and manipulation of the terrain in real-time, for more efficient algorithms to re-create or update only the relevant parts of the terrain, and the visual quality of the rendering, where further parts of the terrain are not affected by deformations to some arbitrary part.

- Visual artifacts must not occur during the extraction of the terrain surface, rendering of the terrain, or at the level-of-detail boundaries where the resolution of the terrain is changed abruptly to accommodate for the difference in surface proximity.

- The terrain representation must be suitable with visualization elements such as lighting, texturing and shadowing. Approaches for the real-time application of such effects must be proposed with the terrain representation.

- The representations and algorithms that operate on it must be able to handle fairly large terrains as long as they fit in the memory. Paging schemes can be used to render extremely large terrain datasets that does not fit in the memory. Such a usage, however, is out of scope of this research. We assume that the entire terrain data can be loaded into the main memory.

- Another important design goal is to make it possible to benefit as much as possible from existing simple and efficient algorithms related to real-time terrain rendering, and avoid having to re-invent most of it from scratch.

1.5 Overview of the Proposed Approach

Heightmap-based representations cannot handle volumetric terrain features. Voxel representations can represent anything, but they cannot be directly rendered using hardware-acceleration, require very large amounts of memory, and lack the extensive set of algorithms required for high quality real-time rendering. We propose a hybrid terrain representation. A relatively low-resolution voxel representation is used to model coarse volumetric features of the terrain. Then 2D surface patches are created to construct the polygonal terrain surface for rendering. Heightmaps are used to displace these 2D surface patches in a third dimension in order to further increase the resolution. In the proposed approach, the surface patches are the logical units on which most algorithms that run on CPU operate, except the actual rendering performed on the GPU which works on vertices, triangles and pixels. Modern GPU features such as vertex and fragment shaders are used in the rendering pipeline for visualization effects such as level-of-detail management, lighting, texturing, and shadows.

1.6 Summary of Contributions

A complete and practical real-time terrain representation approach that can handle volumetric terrain features is proposed in this thesis. Throughout the thesis, the theoretical basis and implementation details of the approach are described and typical performance characteristics are discussed. The specific contributions of this thesis are as follows:

- A detailed survey of existing terrain representation and rendering approaches used for real-time terrain rendering, including visualization techniques, such as level-of-detail approaches,
- A novel hybrid terrain representation that is able to represent terrains with volumetric features such as caves, overhangs, arches, and cliffs,
- A surface extraction method that can be used to extract a polygonal surface of a volumetric representation where the terrain surface is constructed using surface patches,
- An artifact-free level-of-detail management scheme with geometry morphing to support smooth transitions, that can be used with the proposed terrain representation and rendering pipeline,
- A reference implementation for terrain creation and rendering using the proposed approach, which is used to demonstrate the abilities of the proposed terrain representation and practical performance characteristics of it, and
- Methods for applying lighting, texturing and shadowing to the rendering pipeline for the proposed terrain representation to achieve high-quality real-time rendering are discussed.

1.7 Organization of the Thesis

The organization of the rest of this thesis is as follows:

- The related work in the field of real-time terrain rendering is described in Chapter 2.
- The proposed terrain representation, data structures that are used to store terrain data, and the method used to generate the terrain surface are discussed in Chapter 3.
- In Chapter 4, elements of real-time terrain visualization, such as lighting, textures, shadows and level-of-detail, and how these methods can be used with the proposed terrain representation are discussed.
- Chapter 5 discusses the implementation details of the reference rendering pipeline and the sample application that is used to create and edit terrains in addition to determine performance and memory usage characteristics of the proposed approach in practice. It also includes a discussion of how the proposed approach compares to other approaches.
- Chapter 6 concludes the thesis with concluding remarks and possible future directions of research based on the proposed approach.

Chapter 2

Background

Real-time 3D terrain rendering has been a popular topic for decades in computer graphics as it is essential for many types of applications. The research on this topic has been going on for so long since there is no silver bullet solution that addresses all kinds of needs and constrains of different types of applications. Furthermore, real-time terrain rendering is a topic that is very closely coupled with the advances in the GPU technology. As the GPU technology gets more advanced, unique approaches are proposed to make better use of the technology available.

Please note that the level-of-detail approaches are not separately investigated as level-of-detail algorithms are extremely tightly coupled with the terrain representation. Thus, the following sections describe terrain representation and level-of-detail approaches together. We will investigate different approaches to real-time terrain rendering in two categories depending on how the terrain model is internally represented:

1. Heightmap-based terrain representations are the ones that are simply based on displacement of a planar surface such as an approach that constructs the terrain surface by displacing the vertices of a regular 2D planar grid according to the values of the heightmap.

2. Volumetric terrain representations are the ones that are inherently able to represent volumetric features of the terrains such as overhangs, caves and

arches. Most prominent approaches in this category use voxels to represent the terrain model. Please note that the internal voxel representation is usually converted to a form that can be used for hardware-accelerated rendering, that is, a polygonal surface, for rendering purposes.

2.1 Heightmap-based Terrain Representations

These algorithms sample the top layer of the terrain surface, like in a top-down view, and consequently they cannot represent volumetric features of a terrain, such as caves and overhangs. The data that is constructed by such sampling of terrain height from a reference height-level is usually called a height-field or a heightmap. For every point on the terrain surface there is only one sample of height value. This sampling can be performed regularly or irregularly and different approaches use different methods of sampling.

Regular sampling, such as a uniform grid of vertices, is easy to work with as the geometry is extremely constrained and well-defined, and very memory efficient as well, since vertex positions and connections are implicitly defined by the index of the element in memory. In this case, it is sufficient to just store a grid of sampled height values. Consequently, regular grid representations are very memory-efficient.

Triangulated irregular networks (TIN), on the other hand, sample the heights irregularly [1]. An irregular triangulated network does not use a uniform grid of samples and consequently can represent more detailed areas of the terrain using more samples and decrease the number of samples in smooth areas. It can, thus, approximate the terrain better with the same number of triangles as the regular grid approach. Kumler, though, states that a regular grid representation requires less storage space than a TIN in case their detail-level is equal [2]. The computation of a TIN is usually more complex, though. Algorithms such as Delaunay triangulation [3] can be used to generate accurate TINs. The algorithm proposed by [1] actually creates the optimal triangulation of a terrain surface for a given number of triangles. It is also more difficult to manage TINs once they are created, as the whole model needs to be re-created every time the resolution level is changed, and texturing of TINs are also more complex than regular grids [4].

Furthermore, TIN generation is a very CPU intensive process. Garland and Heckbert propose an optimized algorithm for generating a TIN from a heightmap [5]. The resulting TIN is not optimal, though, and may result in visual artifacts such as very thin triangles unlike the TINs created with Delaunay triangulation.

The approaches mentioned so far do not use real-time view dependent level-of-detail in terrain representation. Gross et al. propose an efficient real-time level-of-detail computation approach that uses the quadtree data structure to represent the terrain surface [6]. Cohen-Or et al. then propose a continuous level-of-detail approach for TINs that are generated with Delaunay triangulation [7]. In this approach, several, typically three or four, TINs at different levels-of-detail are generated and blended in real-time to avoid inconvenient popping artifacts. The blending is performed at the vertex-level and not at the pixel-level. Lindstrom et al. propose a continuous level-of-detail approach for terrains that use regular grid sampling of a heightmap [8]. Their approach divides the terrain up into blocks of different levels-of-detail and represents the blocks in a quadtree. The level-of-detail computation is performed at both the block-level by selecting the appropriate block for rendering, and then at the vertex-level by selecting the important vertices for rendering. Even though the internal representation of the terrain in this approach uses a regular grid, the resulting geometry used for rendering is, in fact, a TIN. As the approach uses a simple regular grid for internal representation, it does not required the intensive preprocessing step of generating TINs and consequently allows real-time terrain deformation unlike other TIN-based approaches.

Hoppe proposes the progressive meshes algorithm in 1996 [9]. This algorithm is not specific for terrains and, in fact, it can work on any type of mesh. This approach defines the original mesh as a very coarse mesh and a set of edge-collapse and edge-split operations that transform the coarse mesh to the original mesh. This approach is stated to be more accurate than previously proposed level-of-detail approaches. The approach is later updated to refine the mesh in a view-dependent way taking the view frustum, surface orientation and screen-space geometric error into account [10], and updated once more to utilize GPU parallelism [11]. Hoppe later adapts the approach to real-time terrain rendering and introduced geomorphs to provide temporal coherence [12].

Evans et al. propose a more restricted TIN representation in 1997, where the triangles have to be right-angled triangles unlike in typical a TIN [13]. This approach is called right-triangular irregular network (RTIN) and makes use of a bintree to ensure that the generated triangles are right-angled. The bintree provides more detail where it is needed by iteratively splitting the triangles at a higher level. This representation also allows for efficient level-of-detail computations according to the rendering viewpoint. Another advantage of this approach over TINs is that the position and connections of vertices do not need to be stored explicitly as these attributes are implied by the position of the triangle in the bintree. The RTIN, in this respect, has the memory-efficiency advantage of regular grids when compared to TINs. Duchaineau et al. propose another algorithm that uses the RTIN representation called ROAM, real-time optimally adapting meshes [14]. This algorithm is one of the most popular real-time terrain rendering algorithms of all time as it addresses some of the most important and difficult problems in an efficient way, such as level-of-detail, view frustum culling and creating triangle stripes for efficient rendering. The level-of-detail in ROAM is also viewpoint dependent and level-of-detail changes are smooth thanks to geometry morphing between different levels-of-detail. The ROAM approach utilizes frame-to-frame coherence to reduce the CPU intensity of the algorithm by using parts of the terrain surface that is computed in the preceding frame. ROAM is one of the few real-time terrain rendering approaches that supports terrain deformation in real-time. One advantage of ROAM is that it can control the generated triangle count thanks to the hierarchical terrain representation used. This, however, is also the reason why ROAM is not so popular on modern GPUs as it requires geometry updates on every frame. This is not a desired situation for modern GPUs as the GPUs are much more powerful now the chances are very high that it will stall waiting for the CPU to update the geometry and upload it to the GPU for each and every frame rendered [15]. Consequently, several other algorithms are proposed that are similar to ROAM but with rather more lightweight geometry updates compared to ROAM and work on batches of primitives instead of on individual primitives [16, 17, 18, 19].

Chunk-based level-of-detail algorithms are not very precise as they assume that the required level-of-detail for the entire chunk is the same and approximate by, usually, the distance to the center of the chunk rather than to the actual vertices. Röttger et al. propose a precise continuous level-of-detail algorithm for heightmap-based terrains [20]. This approach makes use of a quadtree data structure similar to some other approaches mentioned so far and it works at the vertex-level. The surface is generated by recursively visiting the nodes of the quadtree in a top-down manner. Not only the distance to the observer is regarded, though, but also the surface roughness is taken into consideration such that smooth surface are rendered with fewer vertices even if they are closer to the observer. The representation inherently supports level-of-detail as the iteration of quadtree nodes can stop at a higher level depending on the distance or roughness of the surface. The continuity of the surface, however, requires that there is at most one level-of-detail difference on the borders. Smooth level-of-detail is obtained by geometry morphing similar to some other approaches and view frustum culling is easily performed by bounding box checking during the visiting of quadtree nodes.

In the very early 2000s dedicated graphics processing units became mainstream and more powerful than CPUs for graphics processing. The real-time terrain representations and algorithms, consequently, adapted to this by reducing CPU intensity of the algorithms and trying to utilize the GPU more. As the GPUs became much more powerful than CPUs, providing GPU enough primitives to render at each frame became a problem. Thus, the newer algorithms focused more on feeding GPU the data to render, even if some significant portion of it is redundant, rather than trying to fine tune and sort out every single vertex on the CPU. One such method is the use of geometrical mipmaps by de Boer [17]. The method is said to be using geometrical mipmaps because of the similarity of the basics to texture mipmapping [21]. This approach uses a regular grid terrain representation where the grid is divided into equal-sized square vertex batches. The mipmaping is performed per-batch where each higher level of batch mipmap contains a quarter of the vertices of the lower-level mipmap. The mipmap level of a particular batch is determined by the distance of the batch to the observer and vertex morphing is used to prevent popping. The unique feature of this method is that it does not continuously update the geometry of the terrain but rather updates the connection between vertices as necessary. Hence, at each frame much less data is uploaded to the GPU.

Ulrich proposes an approach to render massive terrains by combining the quadtree representation with RTINs [16]. In this approach, each internal node of the quadtree stores its own chunk of geometry and texture and when a node is to be rendered the geometry is sent to the GPU collectively making its use efficient for modern GPUs. View frustum culling is easy as in most approaches whose terrain representation uses a quadtree. He also propose a paging scheme where the data of a node is loaded on demand. This allows the algorithm to render massive terrains that may not even fit in the main memory. A view-dependent level-of-detail approach with vertex morphing is employed. Another approach that works on batches of vertices is proposed by Cignoni et al. [22] where a hierarchical representation of vertex batches are stored in a bintree. Each vertex batch is, in fact, a TIN approximation of the area defined by the bintree node. Each TIN patch and the bintree is constructed from a heightmap. This algorithm allows the rendering of massive terrains as it does not require the entire terrain data to be loaded in the main memory. The data is loaded when it is demanded to be rendered. One downside of the algorithm is the complexity of the pre-processing needed to create the data structures as it takes hours for large terrains. They later extend this method to render planet-sized terrains [23]. The approach uses a pre-fetch algorithm to guess the soon-to-be-needed chunks of data and loads it to the main memory. This helps the application to smoothly run and not stall waiting for I/O operations to complete. The performance of the rendering approach is stated to be mostly dependent on the GPU processing power as its CPU intensity is very low.

Losasso and Hoppe propose the use of geometry clipmaps targeting a more efficient level-of-detail scheme for modern GPUs [24]. This approach is based on the texture clipmap algorithm [21] but operates on the geometry of the terrain rather than the textures. It uses a regular grid representation for the terrain favoring its simplicity and manageability. The terrain is divided to grids of different levels-of-detail where the level-of-detail of a particular grid is simply determined by its distance. The approach makes use of vertex buffers for efficient rendering as vertex buffers are optimized for rendering. The fact that vertex buffers are stored on the video memory, rather than the main memory, allows very fast access to the data by the GPU. The entire terrain data is loaded in a compressed format on the main memory. When a grid is needed to be updated, the relevant part of the data is decompressed and the vertex buffers are updated. The approach uses a similar level-of-detail scheme for texturing as well. The compression of terrain data allows very large terrain datasets to be rendered in real-time. Transition regions are defined to prevent visual artifacts caused by different levels-of-detail where the border vertices are interpolated between different levels-of-detail. Each vertex is also morphed geometrically between different levels-of-detail to prevent popping artifacts. In this approach, the geometry is simply updated and sent to the GPU for rendering. This approach is soon updated by Asirvatham and Hoppe such that almost all computation is done on the GPU [25]. This is one of the first real-time terrain rendering approaches that uses the GPU processing power so extensively. The original algorithm used vertex buffers, but vertex buffers cannot be modified on the GPU. The new algorithm uses textures to store vertex data and these textures are sampled in the vertex shaders. Vertex shader is essentially the counterpart of vertex transform function in the old fixed-function rendering pipeline which became programmable on modern GPUs in around 2001. Later the GPUs gained the ability to sample textures in the vertex shader and, consequently, it was possible to compute vertex coordinates based on a geometry texture in the vertex shader. With this approach, almost all of the work done on the CPU in the original geometry clipmap algorithm is moved to GPU. The only operation that still takes place on the CPU is the decompression of the compressed terrain data.

Vertex shaders are used to operate on vertices and compute vertex attributes. Pixel shaders (i.e., fragment shaders), on the other hand, operate on pixels and became programmable, soon after vertex shaders did, in around 2002. Pixel shaders are usually used to create special image-based effects, e.g., post-processing effects on the rendering. A very different approach, however, is proposed by Mantler and Jeschke to perform ray casting in the pixel shader in order to render a terrain model [26]. The CPU and the vertex shader do almost nothing in this approach. The terrain data is stored as a texture representing the elevation data where each texel of the texture stores a height value. Ray marching is the used in the pixel shader to determine the point that the ray originating from that pixel intersects the terrain model, that is, if there is an intersection. Otherwise the pixel is discarded and nothing is rendered. Ray casting is optimized by an efficient empty space skipping method very similar to the one proposed by Kolb and Rezk-Salama [27]. Interestingly, the performance of this algorithm is independent from the size of the terrain or the number of vertices. The performance of the algorithm is merely dependent on the number of pixels that is used to render the scene since all the computation is done per-pixel in pixel shaders. The maximum size of the terrain is limited by the largest texture size the GPU supports, though, unless the CPU is used to continuously update the elevation texture. Please note that this approach uses ray casting but unlike most other ray casting renderers it cannot render volumetric representations as the elevation data is stored as a texture as in a heightmap rather than as a voxel representation. Therefore, it is not possible to render terrains with volumetric features using this approach, and any other approach mentioned in this section for that matter.

Although the heightmap-based approaches cannot represent volumetric features, some approaches use slight modifications to the heightmap-based regular grid approach as to allow simple cases of volumetric features. One example of this is introduced by McAnlis [28]. His approach to terrain representation initially uses a regular grid representation of a heightmap but it allows individual vertices to be displaced by a full vector-field displacement along the x-, y-, and z-coordinate axes rather than only along the y-coordinate axis like a typical heightmap-based approach. This makes it possible to add simple overhangs and vertical faces to the terrain whereas even these features are not possible with a typical heightmap-based approach. The downside of the approach is that the resolution of the terrain needs to be increased significantly to make up for the displacement of vertices. It is also still not possible to represent complex volumetric terrain features such as complex overhangs, caves and arches using this approach. A similar approach is proposed by McRoberts which uses geometry images to store the displacement of regular grid vertices along three axes instead of one [29]. A typical heightmap stores only a single channel of data per pixel representing the corresponding height value. A geometry image, on the other hand, stores three channels per pixel where the displacement of the corresponding vertex along the x-, y-, and z-coordinate axes. Please note that the vertices still form a regular uniform grid before the displacement is applied. The displacement along the three coordinate axes just makes it possible to create simple overhangs and vertical faces. It is not possible to represent complex volumetric features with this approach either.

2.2 Volumetric Terrain Representations

Volumetric terrain representations allow volumetric features of the terrain to be defined, such as caves, overhangs and arches. Most of the volumetric representations are based on voxels in which case the terrain model is discretely defined making it possible to fine tune the terrain. In several approaches, on the other hand, the terrain representation is a density function that is procedurally computed which makes it very difficult to control and fine tune the details of the terrain model.

The approach proposed by Geiss can use both a procedural density function or a discrete representation stored as a 3D texture [30]. If a density function is used, then the values returned by the density function is stored in a 3D texture and this texture is used in a second rendering pass to do the actual rendering of the terrain. We have already mentioned vertex and pixel shaders. The next advancement in the programmable rendering pipeline of modern GPUs resulted the programmable geometry shader. It was not possible to generate geometry on the GPU before the geometry shader. Vertex and pixel shaders can only operate on existing geometry. Geometry shader, on the other hand, can generate and stream geometry, i.e., triangles for rendering. This approach utilizes geometry shader to generate a polygonal surface for the volumetric representation of the terrain. Almost all real-time terrain renderers that use a volumetric representation convert the volumetric data to polygonal surfaces for rendering purposes. Otherwise, rendering the entire terrain using ray tracing, or even ray casting is still not possible at interactive frame rates. Almost all approaches, like this one, use the marching cubes algorithm [31] or a variant of it for this purpose. This terrain representation, in theory, is able to render terrains with volumetric features. The downside of the approach is that it is difficult to design density functions to obtain a desired terrain model, although it is possible to generate interesting arbitrary terrain models by using procedural modelling techniques [32]. It is also extremely challenging, if possible, to create a density function for a given terrain dataset. In the approach proposed by Geiss the surface normals and texture coordinates are not precomputed and ready-for-use as the geometry is generated on-the-fly. Surface normals are computed by the gradient of the density function which is done by sampling the density function six times around the point for which the normal is computed. Since the texture coordinates do not exist in this case, traditional texturing approaches cannot be used. Planar texture projection is instead used to project the geometry onto the three coordinate planes and the surface normal is then used to select one of these projections.

Forstmann et al. propose a similar approach that renders terrains represented by iso-surfaces [33]. This approach is based on the interactive view-dependant iso-surface rendering approach proposed by Gregorski [34] and is inspired by the geometry clipmaps approach of Losasso [24], which is basically the 2D counterpart of what Forstmann et al. propose for 3D. The approach basically uses clip-boxes in 3D instead of clipmaps in 2D. This approach is quite efficient as a volumetric rendering approach. It is able to reach a peek frame-rate of up to 120 with a hundred thousand untextured polygons using 5 clip-boxes. The algorithm is stated to be more memory efficient compared to the method proposed by Gregorski [34] as it does not require the use of a tree data structure to store the representation. The resolution of the rendered sample models, however, is quite low and further details are added to the extracted surface by applying noise. Furthermore, it shares most of the downsides of the method proposed by Geiss [30] as it is difficult to represent a detailed terrain model with iso-surfaces. It is also very difficult to fine tune a terrain represented by iso-surfaces as well as generating an iso-surface representation of a given terrain model.

Rendering voxel-based large volumetric terrains in real-time has not been very popular until recently due to the limitations of the GPU processing power, memory limitations as well as problems originating from the surface extraction algorithms that are used to extract polygonal surfaces of voxel representations. One such problem that is very closely related to rendering terrains is the difficulty of level-of-detail management in surface extraction. Marching cubes and other similar algorithms do not work very well when the resolution of the sampling grid is not constant. The level-of-detail approaches, however, require the sampling grid resolution to vary among different levels such that a lower level-of-detail produces less geometry. This causes inconvenient artifacts at the boundaries of levels-of-detail where surfaces with different resolutions do not align and visual artifacts such as cracks are inconveniently evident. Lengyel very recently proposes the Transvoxel algorithm [35], which is arguably the best real-time voxel-based terrain rendering approach. This approach eliminates all visual artifacts resulting from the use of marching cubes algorithm with varying grid resolution and therefore allows level-of-detail management of the extracted surface in real-time. The approach, however, is not without any problems. First of all, the popping artifacts occur because there is no morphing between different levels-of-detail. This degrades the visual quality of the rendering and requires much higher resolutions to be used so that the popping is not very disturbing. The approach is CPU intensive as it frequently updates the geometry by re-computing parts of the terrain surface as the viewpoint, and thus levels-of-detail of parts of the terrain changes. CPU to GPU geometry updates are also frequent for the same reason. The resolution of the terrains are typically low with this approach, though, where a much higher resolution is required for very detailed large terrains.

Chapter 3

The Proposed Approach

This chapter presents our approach to representing a complex three-dimensional (3D) terrain that may contain volumetric terrain features such as caves, overhangs, cliffs and arches for terrain editing and visualization in real-time applications. It describes the design goals of the approach and how each goal constrains and affects the design of the representation in various ways.

3.1 Goals

There are several important goals that we want to achieve with the proposed terrain representation approach and as a result each of these goals affected the design decisions along the way:

- The terrain representation should be more flexible and expressive compared to a simple grid and height map-based approach. More specifically, the representation should be able to handle anything a height map approach can and in addition it should be able to handle interesting terrain features such as caves, overhangs, cliffs and arches.
- The representation should support interactive frame rates, preferable real time; i.e., 30 frames per second. In order to achieve this it is required to

benefit from hardware accelerated rasterized graphics. Hence, the representation should be suitable for rendering using a modern Graphics Processing Unit (GPU).

- It should not be assumed that the terrain is completely static. The representation should be dynamically editable and deformable in real-time. As a result of the modifications, the data structures in the CPU and GPU must be updated. Therefore, the algorithms to update these structures should be able to work in real-time. The changes made to the data structures stored in the CPU must also be reflected to the data structures in video memory of GPU in real-time. As a result, the amount of data sent through the CPU-GPU data bus should not exceed capabilities of a modern GPU.

- The representation should be able to handle fairly large terrains as long as it can be stored in the main memory. This is roughly on the order of millions of vertices or several hundreds of megabytes of data. It should be able to yield about 1 *meter* resolution in each axis on a 1 km^3 space. Extremely large terrains can theoretically be stored on a high-speed secondary storage device and portions of data fetched to main memory on an as-per-needed basis. This, however, is outside the scope of this work as we assume that the terrain data is completely available on the main memory for random-access.

- The representation should be suitable for applying basic 3D visualization elements such as lighting, texturing and shadowing.

- Rendering large terrains in real-time without proper level-of-detail support is not plausible. Thus, a level-of-detail scheme should also be proposed with the terrain representation as to allow real-time rendering of large terrains.

3.2 Terrain Representation

Since one of our main goals is real-time rendering, the internal representation used to store the terrain data should be efficiently convertible to a suitable form for rendering.

Modern GPUs are designed to accelerate rasterization-based rendering. They are not good at rendering volumetric data. In fact, they do not support rendering volumetric data whatsoever. It is, however, possible to employ various hacks to simulate volumetric rendering using rasterization. Benefiting from hardware accelerated graphics requires converting any internal representation to a bunch of polygons, usually triangles, for rasterization-based rendering.

3.2.1 Heightmap-based Approaches

Simple heightmap based approaches involve a regular, and often uniform, 2D grid of vertices. These vertices are then connected in a straightforward manner to create polygons that represent the surface of the terrain. The values stored in the heightmap are used to displace these vertices in the third dimension and basically determines the height of each vertex. The heightmap basically represents the samples of Equation (3.1) at a fixed frequency, where x and z are the coordinates of the vertices in the x- and z-axes, respectively, and y is the coordinate of that vertex in the y-axis (i.e., the height of the vertex).

$$h(x, z) = y \tag{3.1}$$

This is the simplest, most compact and efficient representation possible. Since x and z values are implied by the structure of the regular 2D grid and do not need to be stored explicitly. Only a few bytes of data per sample is required to store the height map depending on the desired resolution in the height-axis. The main downside of this approach is its extremely limited expressive power in representing non-planar terrain features, such as caves, overhangs, and even steep cliffs. This representation can basically only define the top-level surface of the terrain. Everything below this surface is considered filled and everything above it is considered empty. It only allows the definition of one and only one height value per grid cell. Hence, it is not possible to represent volumetric features with this approach. This representation is, therefore, not sophisticated enough to handle complex terrains.

3.2.2 Voxel-based Approaches

Voxel-based approaches divide the working space into 3D grid cells constructing a regular 3D grid rather than a planar 2D grid like in heightmap-based approaches. Each 3D grid cell is called a voxel, similar to a 3D version of pixels on a 2D image. The most basic attribute of voxels are their status of being empty or filled. A filled voxel represents a subspace filled with material while an empty voxel means that subspace is not filled with material, i.e., filled with air. Depending on the application, each voxel may have other attributes, such as a normal vector, color information, and texture information.

Voxel representations are very popular in offline rendering and very rarely used in real-time rendering. The reason behind this is the fact that a voxel representation is not suitable to be directly used for rasterization-based rendering since a voxel representation defines a volumetric structure rather than a polygonal surface. Voxel representation suits well if rendering techniques such as ray casting and ray tracing are to be used. For rasterization-based hardware accelerated rendering, however, the voxel representation must first be converted to a polygonal surface and then the polygonal surface can be rendered efficiently. Unfortunately, extracting the surface of a very large terrain represented in voxels is not an easy and smooth process.

Voxel representations of large and detailed 3D models are also not memory efficient enough to be used in real-time applications. In order to achieve a 1 *meter* resolution in each coordinate axis in a 1 km^3 working space it is required to store at least a billion voxels ($(10^3)^3 = 10^9$). Even if a single byte of data is used to store each voxel, this representation would still require about 1 GB of memory just for the voxel representation of the solid terrain model and nothing else. Processing such large amounts of data for editing and visualization in real-time applications is not very plausible.

3.2.3 The Proposed Hybrid Approach

Approaches based on heightmap and voxel representations alone do not suffice to achieve the defined goals. The proposed hybrid approach combines the voxel- and heightmap-based approaches in an effort to inherit advantages of both approaches:

- expressive power of the voxel-based approach, and
- simplicity and efficiency of the heightmap-based approach.

In this approach, the terrain geometry is generated in two steps:

1. A relatively low-resolution voxel representation is used to define the geometry of the terrain coarsely. The surface of the geometry is extracted using a novel technique in such a way that the surface consists of regular terrain patches.
2. Each terrain patch is then assigned a heightmap, which is used to displace the vertices of that terrain patch. This process increases the resolution of the terrain geometry in practice and allows for various details to be added to anywhere on the terrain surface.

3.3 Data Structures

Our approach uses a uniform voxel grid for voxel representation of the terrain. In such a representation the filled voxels and the empty voxels are usually grouped together. We made the following assumptions:

- if a voxel is filled, its surrounding voxels are probably filled, and
- if a voxel is empty, its surrounding voxels are probably empty.

There will obviously be exceptions to this assumption in the voxel representation but it can still be exploited to make a more compact representation and decrease the memory requirements for storing the voxel terrain data for typical terrains. For this purpose, our approach uses an octree to store the voxel data.

The root node of the octree is the entire workspace of the terrain. Each octree node can be divided into 8 equal-sized axis-aligned child nodes. This division is only performed if the extra level-of-detail subspace is demanded in that subspace (see Figure 3.1). If the subspace represented by an octree node is completely

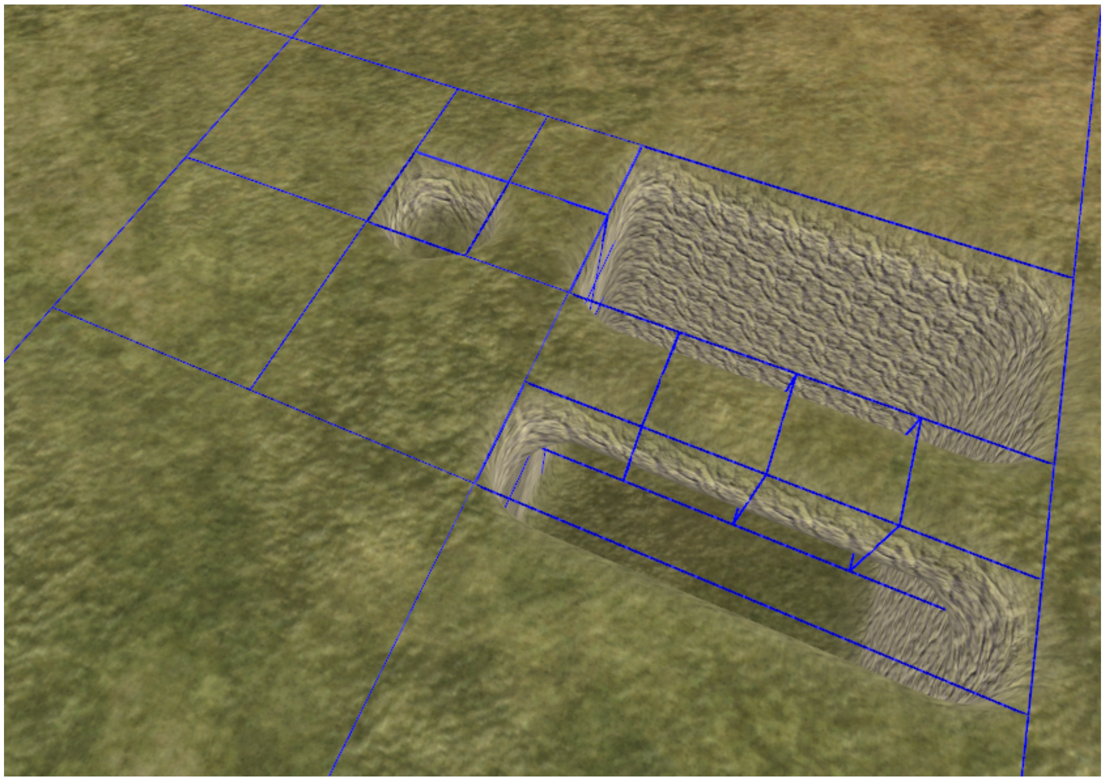


Figure 3.1: Octree representation of a voxel space can be used to increase resolution where it is needed.

filled or completely empty then that node is not divided further into child nodes. Hence, an internal node has either 8 children nodes or none at all. Employing octrees prevents additional memory usage where extra level-of-detail is not needed while being able to provide a higher resolution where it is needed. One downside of octrees compared to storing an uncompressed 3D voxel array is that octrees also store the internal nodes whereas a simple 3D voxel array only stores the leaf nodes. In a full octree of height h, the number of internal nodes is given by Equation (3.2), and the number of leaf nodes is given by Equation (3.3).

$$n_i(h) = 1 + 8 \times \frac{8^{h-1} - 1}{7}, \qquad \forall h >= 1 \tag{3.2}$$

$$n_l(h) = 8^h, \qquad \forall h >= 0 \tag{3.3}$$

The ratio of the number of internal nodes to the number of the leaf nodes is about 0.14. This extra cost of storing internal nodes is easily amortized in most cases, though. The number of leaf nodes in a full octree of height 6 is about $260,000$ and the number of total nodes, including intermediate nodes, is about $300,000$. Even if a quarter of the paths to the leaves stop at a height of 5 the number of total nodes is reduced to about $234,000$. In practice the efficiency gains in terms of storage are significantly higher since most of the paths do not reach to the maximum height of the octree.

One of the most important advantages of using an octree is the greater efficiency of running different queries on the geometry in an hierarchical manner. This is a feature that is required by terrain editors and is used for voxel selection and manipulation in the simple terrain editor that we have implemented as well. It can also dramatically speed up collision queries, culling queries and level-of-detail queries, especially in real-time applications.

3.3.1 Voxel Structure

Each voxel has a voxel index associated with it and this index is stored in memory with the voxel. The relation between the set of voxels in the octree and the set of voxel indices are one-to-one, meaning that

- any given voxel in the octree, whether it is a leaf node or an internal node, has one and only one index, and
- any given voxel index points to one and only one voxel in the octree.

A voxel index is represented using four bytes in memory as follows:

$$\begin{aligned} \text{voxel level} &: 4 \text{ bits}, \\ \text{x-index} &: 9 \text{ bits}, \\ \text{y-index} &: 9 \text{ bits, and} \\ \text{z-index} &: 9 \text{ bits.} \end{aligned}$$

The voxel index structure uses a total of 31 bits of the four bytes. The final bit is used to store whether the voxel is filled or empty when the voxel index is stored within a voxel. This bit is not used if the voxel index is used merely to point to a voxel.

A voxel index stored in this format allows up to nine additional levels to the root level since the index fields are stored in 9-bits. Consequently, the maximum height of an octree that uses this representation cannot exceed ten. An octree of height ten has over one billion leaf nodes and storing that many voxels in memory is not plausible. In practice, an octree that is of height five or six provides enough resolution for most terrains.

The voxel level field is the level of the corresponding voxel in the octree where the root voxel is on *level* 0, its child voxels are on *level* 1, etc. It is essentially the distance of a voxel to the root of the octree.

The x-index, y-index and z-index fields store the index of the voxel respectively in x, y and z axes. The i-th bit of these fields determine whether the voxel is the first or the second child of the parent voxel in $(i-1)$-st level of the octree on the corresponding axis. The most significant bit of each field are considered the first bits of the fields representing the child selection at *level* 0 of the octree, the second bits represent the child selection at *level* 1 of the octree, and so on. The number of meaningful bits in these fields is determined by and equal to the voxel level. If the voxel level is 1, then only the first bits of each field is meaningful since the child selection is done only on *level* 0 in this case.

This voxel index representation has several advantages compared to traditional memory pointers:

- It takes up just as much space as a memory pointer but stores additional information about the voxel: the level of the voxel in the octree.
- Given a voxel index, the index of the parent voxel, that is the index of the voxel that contains this one, can be computed just by decrementing the value of voxel level by 1.
- Given a voxel index, the index of the child voxels can be computed by incrementing the voxel level by 1 and setting the i-th bit of each index field

to 0 or 1 where i is equal to the incremented voxel level. Indices of all $2^3 = 8$ child voxels can be generated this way (see Figures 3.2 and 3.3).

- Given a voxel index, the index of the neighboring voxels at the same level of the octree can be computed by incrementing, decrementing or keeping the values of the each of the index fields. There are 3 possible operations (increment, decrement and keep value) that can be performed on 3 index fields to compute the voxel index of $3^3 = 27$ voxels, one of which is the current voxel. Therefore, voxel indices of all 26 neighboring voxels can be computed extremely easily this way (see Figures 3.2 and 3.3).

- It simplifies the implementation of algorithms that work on the octree to use voxel indices rather than traditional memory pointers to actual voxels. A voxel index can be used to iterate voxels, move to neighboring voxels etc. unlike a memory pointer which can only be used to access the data pointed by it. Voxel indices are essentially higher-level abstractions compared to memory pointers as they also contain contextual information.

- This representation also saves memory space. The size and position of any voxel can be computed given the voxel index preventing the need to explicitly store the size and position of each voxel in the octree in memory (see Sections 3.3.1.1 and 3.3.1.2). Instead, only the voxel index of each voxel is stored which only takes up 4 bytes of memory space per voxel.

3.3.1.1 Computing Voxel Size from Voxel Index

Computing the size of a voxel given its voxel index is extremely easy. The size of all voxels at any level of the octree are equal since the size of a voxel depends only on the level at which the voxel resides. Note that the size of the entire octree is already known since it is defined while creating the octree. The size of the octree is divided by 2 in each axis at each increment of level. The size of a voxel at level i can be computed using Equation (3.4) where $\overrightarrow{S}$ and $\overrightarrow{s}$ are three dimensional vectors representing respectively the size of the octree and the size of any voxel at level l of the octree.

$$s(l) = \overrightarrow{s} = \frac{\overrightarrow{S}}{2^l} \tag{3.4}$$

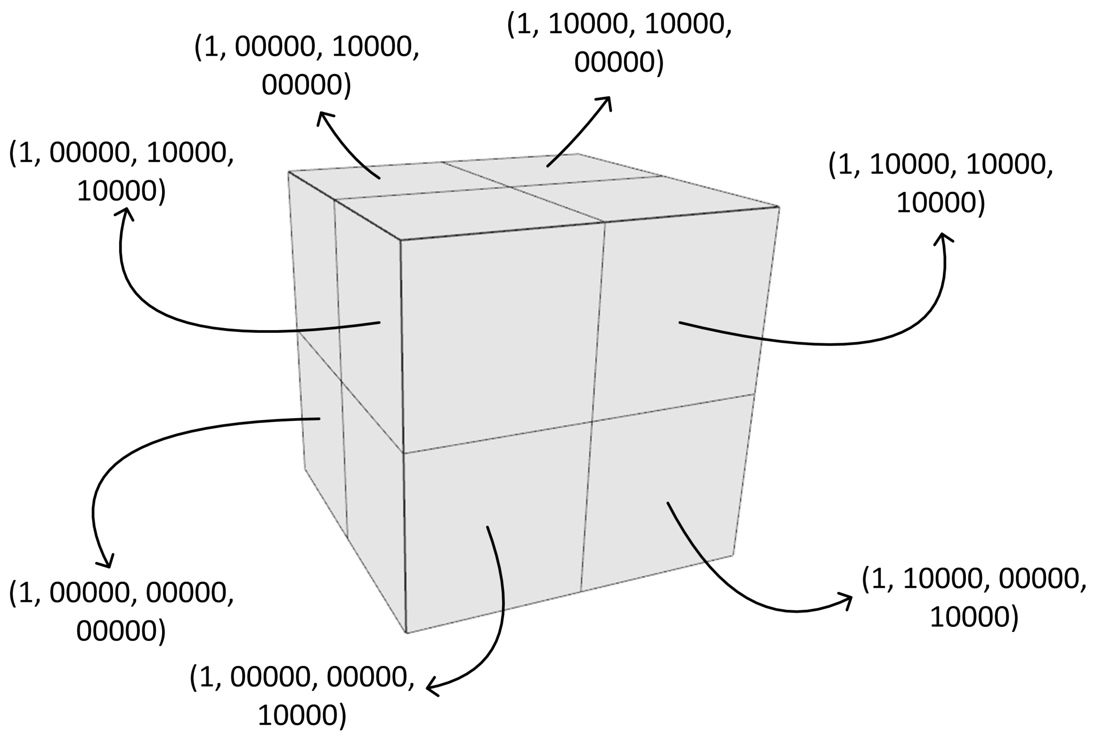


Figure 3.2: Voxel indices in (voxel level, x-index, y-index, z-index) format at level 1. Index fields are in binary representation.

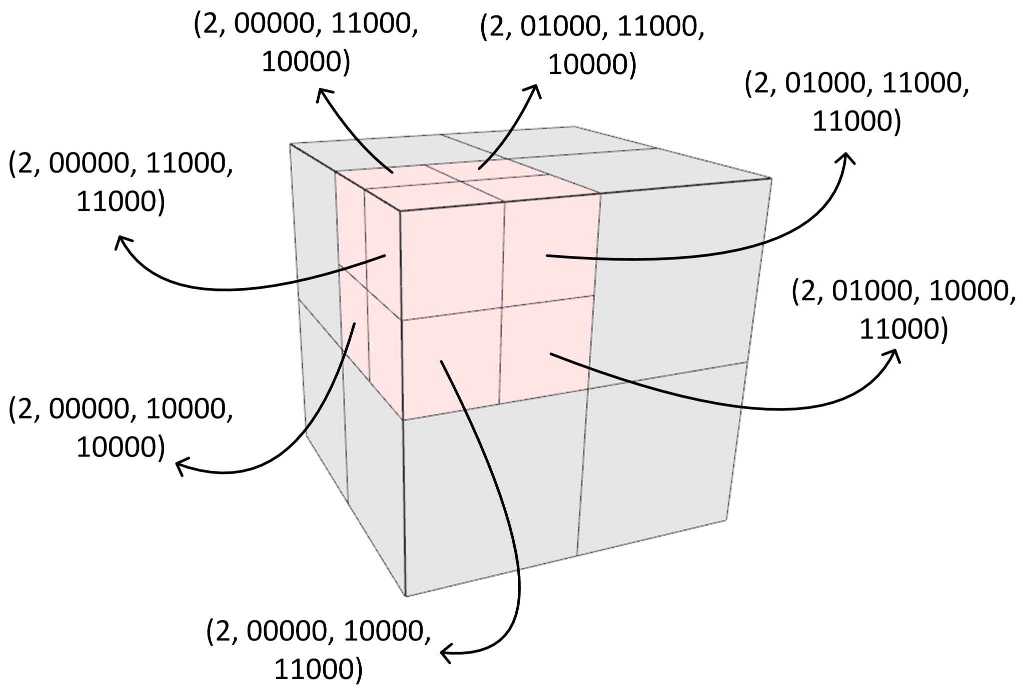


Figure 3.3: Voxel indices in (voxel level, x-index, y-index, z-index) format at level 2. Index fields are in binary representation.

3.3.1.2 Computing Voxel Position from Voxel Index

The position of a voxel is defined as the center of the volume contained by that voxel. In each level, a voxels position is displaced in each axis by an amount equal to the half of the voxel size at that level in that axis depending on the value of the index field bit for that level. Depending on whether the value of the index field bit is 0 or 1 the displacement is applied through respectively the negative or the positive side of the corresponding axis. p_x, the x-component of the position of a voxel at level l, can be computed using Equation (3.5). P_x is the x-component of the center position of the entire octree, $f_x(l)$ is the value of the l-th bit of the x-index field of the corresponding voxel index, and $s_x(l)$ is the x-component of the size of a voxel at level l (see Equation (3.4)). Y-component and z-component of the voxel position can be computed similarly (see Figure 3.4).

$$p_x = P_x + \sum_{i=1}^{l} \Big(\big(f_x(i) \times 2 - 1\big) \times s_x(i+1) \Big) \tag{3.5}$$

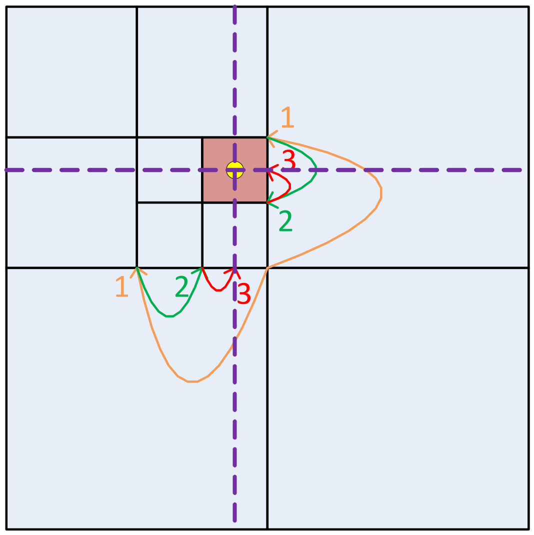


Figure 3.4: Computing the x and y components of the position of a voxel at level 3 from its voxel index (xy cross section of the voxel space is depicted. The z-coordinate would be computed similarly).

3.3.2 Patch Structure

The terrain patches are used to generate the surface geometry of the terrain represented by voxels in the octree. Each patch surface is then subdivided into a number of triangles that are used for rendering. Therefore, it is accurate to say that the primitive type of proposed terrain representation is patches while the primitive type used at the stage of rendering is triangles.

Each patch has the following attributes associated with it and stored in the main memory with the patch:

- A number of control points that define the surface of the patch.
- A vertex buffer that stores the vertices approximating the surface of the patch.
- An index buffer that stores the indices of the vertices in a particular order so as to generate the triangle list for rendering the surface of the patch.
- A heightmap associated with the patch to be used as a displacement map on the surface.

- Pointers to linked lists for vertices that are shared with other patches on the surface.
- Up to four pointers to neighboring patches with coinciding edges are stored.

These attributes are described in detail in the following sections.

3.3.3 Vertex Structure

Patches representing terrain surface cannot be directly rendered using hardware accelerated rasterization-based rendering. The surfaces must first be approximated using a number of vertices and these vertices must be connected with edges in a particular order to make up triangles. These triangle lists can then be used to render the terrain surface.

Each vertex has a number of attributes associated with it. The representation of vertices differ in main memory (used by the programs that run on CPU) and in video memory (used by the programs that run on GPU). This is mainly due to the fact that not all attributes that are stored in the main memory are required by the programs that run on GPU and video memory capacity is usually considerably smaller than the capacity of main memory. Hence, the unused attributes are stripped off while the vertex data is streamed to video memory, and some attributes are compressed.

Vertex representation in the main memory stores the following attributes:

- The original position of the vertex as a 3D floating point vector (12 bytes of data).
- The displacement normal of the vertex as a 3D floating point vector (12 bytes of data). This defines the direction of displacement that is applied to the vertex by the heightmap.
- The displaced position of the vertex as 3D floating point vector (12 bytes of data). It is equal to the original position of the vertex displaced in the direction of the displacement normal by an amount determined by the corresponding heightmap value.

- The actual normal of the vertex as a 3D floating point vector (12 bytes of data). This is the normal of the vertex that is computed after all the displacement operations are performed on the terrain. This normal vector is used for accurate texturing and lighting computations.

- Color of the vertex as a 4D floating point vector (16 bytes of data). The color can be applied as a post-processing effect on the color value obtained from textures for artistic purposes. It is also used for debugging purposes where some part of the visualized terrain data is desired to be highlighted.

- Level-of-detail transition distance of the vertex as a single floating point value (4 bytes of data). This value determines the distance at which the vertex will transform into a lower level-of-detail position and normal assigned to it by the level-of-detail management algorithm.

- Pointers to two other neighboring vertices that are on a lower level-of-detail (8 bytes of data). These vertices are accessed to compute the lower level-of-detail transition position and normal of the current vertex when the vertex data is needed to be sent to the video memory.

- Index of the vertex in GPU vertex buffers as an unsigned integer value (4 bytes of data). This value is actually a pointer to the data of this vertex stored in video memory and is used whenever the vertex data is updated on the CPU in main memory and as a result the old data stored in the video memory needs to be updated. The most significant bit of this field is used as a flag to indicate whether or not the actual normal of this vertex is invalidated and needs to be recomputed. This happens whenever one of the heightmaps are modified in a way to require the actual normal of this vertex to be recomputed.

Vertex representation in the video memory stores the following attributes:

- Position of the vertex as a 3D floating point vector (12 bytes of data).

- Normal of the vertex as a 3D floating point vector (12 bytes of data).

- Color of the vertex compressed to a 4D unsigned byte vector (4 bytes of data).

- Lower level-of-detail transition position of the vertex as a 4D floating point vector (16 bytes of data). The w-component of this (x, y, z, w)-vector stores the transition distance.
- Lower level-of-detail transition normal of the vertex as a 3D floating point vector (12 bytes of data).

3.4 Surface Extraction

Surface extraction is the process of computing the coarse surface of the terrain that is represented by the 3D voxel model. It should be noted that this is not the ultimate form of the terrain surface but only an intermediate representation which will later be used for terrain surface generation (cf. Section 3.5).

Marching Cubes algorithm and its derivations are very popular in extracting the surface of a volumetric representation. In our approach, however, we cannot use that kind of algorithm since its output is just a bunch of triangles, i.e., a triangle soup. If the Marching Cubes algorithm is used to extract the surface then the terrain model is reduced to a triangular irregular network (TIN) and using heightmaps and level-of-detail approaches with a TIN representation is considerably harder if possible. Our approach requires generating the surface from regular patches on which the heightmaps can then be applied to achieve higher resolution. There are several properties of the terrain patches that must be ensured:

- The surface consisting of the patches must be entirely connected and continuous. There must be no holes, overlapping or colliding patches on the surface. This is required to prevent rendering artifacts.
- Patches must be rectangular so that the vertices on the patch can be mapped to planar (u, v) coordinates. This way, the values in the heightmap can be used to displace the vertices on the patch.
- Being natural formations terrains are generally smooth surfaces. They usually lack sharp corners and edges. In order to be able to generate a smooth

terrain surface all patches must be smooth on the interior. The combination of the patches must also not introduce sharp edges, corners or dramatic slope changes for the same reason. It is possible to add these kinds of details later by displacing the vertices on the smooth surface using heightmaps.

- Each edge of a patch must exactly align with one edge of one and only one other patch. This is actually a requirement to simplify the surface structure so that level-of-detail algorithms can work more efficiently and produce visually better results.
- The vertices on the patch surface should be generated in a controlled manner such that the vertices that are on the overlapping edges of two patches coincide. This makes it easier to seamlessly combine patches since patches can share vertices on the edges with the corresponding adjacent patch.

The choice of patch representation in the proposed technique is Bézier surfaces [36] (The term *Bézier surface* and *terrain patch* (or just *patch*) are used interchangeably throughout the text). A Bézier surface is a three dimensional specialization of Bézier curves. The Bézier curves (and surfaces) have several useful properties that make it the best choice for our approach:

1. A Bézier curve has the endpoint interpolation property, meaning that the curve is guaranteed to pass through its endpoints. Similarly a Bézier surface is guaranteed to pass through all four control points that define the four corners of the surface. This is essential for ensuring connectivity of the patches generated.
2. Bézier surfaces are invariant under affine transformations and translations. If the control points of a Bézier surfaces is transformed using affine transformations and/or translations, then the surface will transform in the same way as its control points. This property is useful in surface extraction phase since many patch configurations can be obtained by either rotating or taking the mirror of the control points of a base configuration.
3. All edges of a Bézier surface are Bézier curves defined by the control points on that edge. This is useful for generating continuous and smooth surfaces by combining patches at the edges.

4. A Bézier surface is guaranteed to not exceed the convex hull of its control points. This means that the bounding box of the control points is also the bounding box of the corresponding Bézier surface. This is useful in making the bounding box computations and range checks more efficient since it is only needed to consider the control points in such computations rather than all the vertices that result from the subdivision of the surface. Note that the number of control points is usually several orders-of-magnitude less than the number of vertices that result from surface subdivision.

5. Subdivision of Bézier surfaces are easier in computational efficiency compared to other surface representations such as non-uniform rational basis splines (NURBS) [37]. It is possible to easily generate an $N \times M$ grid of vertices approximating the surface where N and M are arbitrary numbers. This is useful for controlling the subdivision level of the surface. NURBS provide extra freedom and more specific control over the surface at the expense of computational complexity. The proposed approach does not benefit from these properties greatly since the generated surface is only an intermediate form of representation which can then be enriched by the application of heightmaps.

One problem with Bézier curves in general is that it is difficult to find the intersection of a line and a Bézier curve. Thus, Bézier curves can almost never be used directly for rendering in ray-casters and ray-tracers. This is not a problem since the proposed approach uses a rasterization-based rendering scheme. Furthermore, the surface representation is only an intermediate form in the proposed approach where heightmaps are applied to patches for manipulation of surfaces. In this case, one would need to compute the intersection of a line with the ultimate form of the manipulated surface anyway, e.g., for collision detection purposes.

3.4.1 Two-Dimensional (2D) Case

Our approach obviously needs the surface extraction to work in 3D. A 2D case of the surface extraction process, however, is quite similar to the 3D case while being fairly easier to comprehend visually. The 2D case of surface extraction is explained first in this section and the 3D case is built on top of that in the next section.

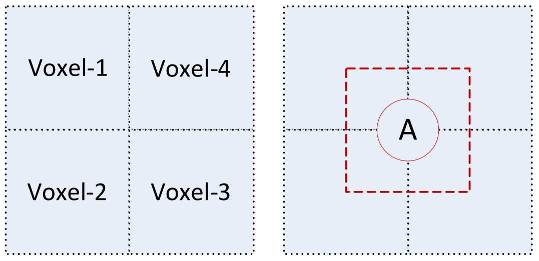


Figure 3.5: Left: Four neighboring voxels in the 2D-case, right: the intersection zone of these voxels. The vertex labeled with A is the center of the intersection zone.

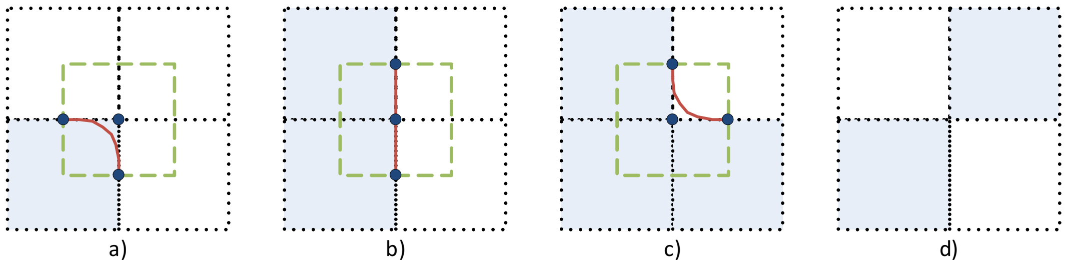


Figure 3.6: Several sample voxel configurations in the 2D-case.

The surface extraction algorithm works on each vertex that is the intersection of 4 voxels in the 2D-case. For each intersection vertex, the algorithm generates a curve inside the intersection zone. Intersection zone is rectangular, centered at the intersection vertex and its size is equal to the size of a voxel (see Figure 3.5).

In the 2D case the intersection zone overlaps 4 voxels. Each of these voxels can be either empty or filled. Thus, there are $2^4 = 16$ possible configurations for the intersection zone. Several possible configurations are shown in Figure 3.6. Before these configurations can be processed by the surface extractor the configurations must be normalized. A normalized configuration is one where all filled voxels in the configuration share an edge with another filled voxel in the same configuration. By this definition, the configurations in Figure 3.6 (a), (b), and (c) are already normalized while the configuration in Figure 3.6 (d) is not normalized and needs to be before the surface is extracted.

In the normalization process, first each voxel is put in a separate normalized group. Then the algorithm tries to combine any two groups by checking whether the voxels in the groups share an edge. If it finds such a voxel then the groups are combined. This step is repeated until no more normalized groups can be

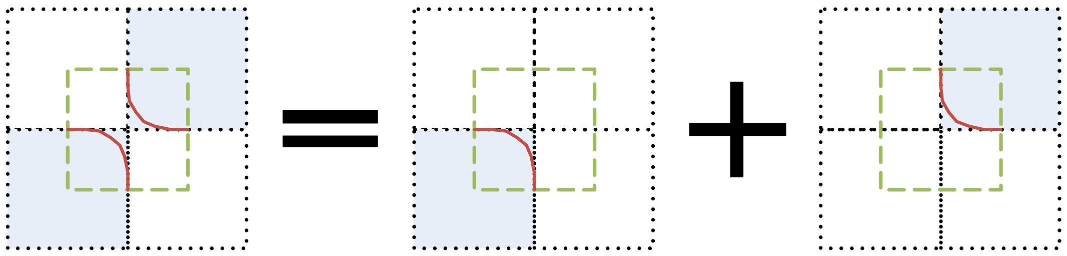


Figure 3.7: Configurations may be split into two during normalization.

combined. See Figure 3.7 for an example of normalization. Normalization is useful for decreasing the complexity of the surface extraction algorithm since a surface can be extracted for each normalized group and then combined if there are more than one normalized groups.

As it is shown in Figure 3.6, each intersection zone in a normalized voxel group has three control points for surface generation. Hence, the proposed approach uses quadratic Bézier curves basically because they are the simplest and they suffice for the job of generating smooth patch surfaces approximating the underlying voxels. The position of the vertices that approximate the surface can then be computed using the parametric curve equation of the quadratic Bézier curve as in Equation (3.6) where P_0, P_1 and P_2 are the three control points of the curve.

$$\begin{aligned} p(t) &= \sum_{i=0}^{2} \binom{2}{i} (1-t)^{2-i} t^i P_i \\ &= (1-t)^2 P_0 + 2(1-t)tP_1 + t^2 P_2 \end{aligned} \tag{3.6}$$

The control points of the surfaces are depicted as filled circles and generated curves are drawn with color red in Figures 3.6 and 3.7. It should be noted the surfaces extracted for some configurations are simply flat lines as the situation demands. The algorithm does not generate a surface in the following cases:

- all of the voxels in the intersection zone are filled, and
- none of the voxels in the intersection zone are filled.

To summarize the surface extraction algorithm in the 2D case, the following steps are performed for each intersection zone in the voxel space:

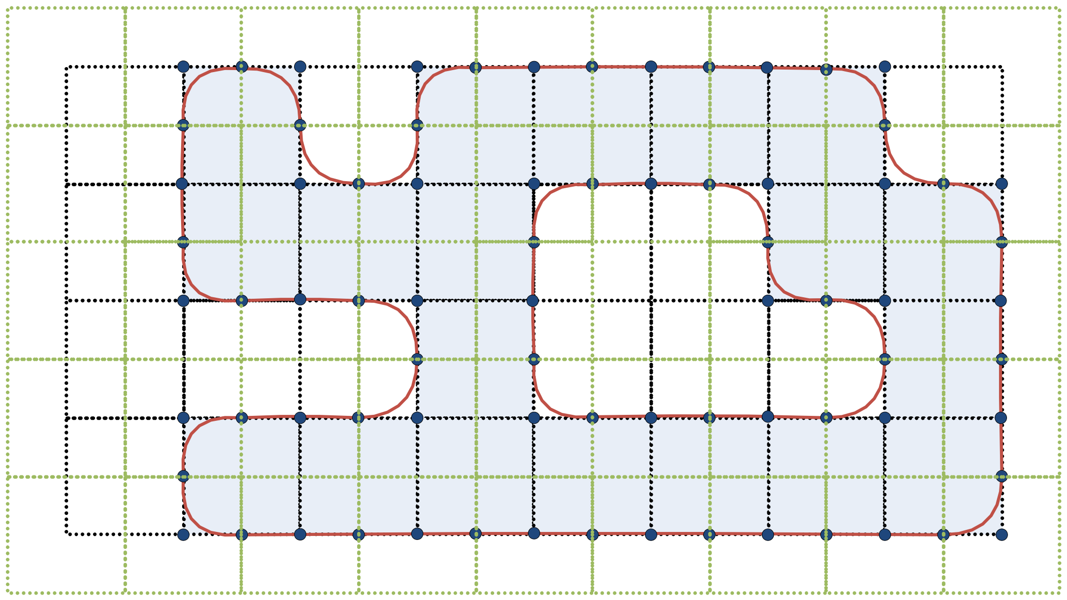


Figure 3.8: The surface extracted by the proposed surface extraction algorithm.

1. If all of the voxels in the intersection zone are filled or are empty, skip the intersection zone without generating any surface.
2. Split voxels into normalized voxel groups.
3. Determine the three control points for the surface of each normalized voxel group depending on which of the four voxels are filled.
4. Generate surfaces using the three control points for each normalized voxel group.

Figure 3.8 shows the final result of the 2D surface extraction algorithm on a simple voxel space. The blue rectangles are filled voxels, the white rectangles are empty voxels, the filled circles are control points of the curves of each intersection zone which are shown as green rectangles. The control points of the adjacent intersection zones are placed in such a way that one of the control points is shared between them. This ensures the continuity of the generated surface (cf. Section 3.4). Note that this algorithm can also be used to produce more than one surface that are not connected to each other if the voxel data represents such a volume, as it is shown in Figure 3.8.

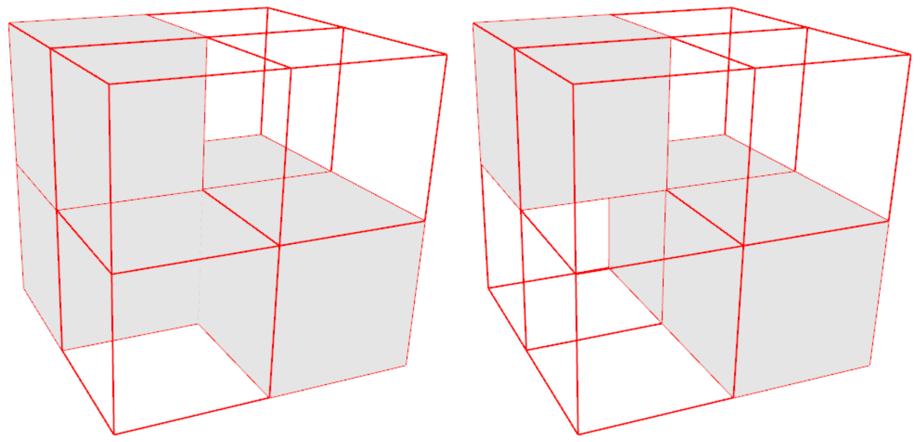


Figure 3.9: Left: a normalized voxel intersection volume, right: an unnormalized voxel intersection volume.

3.4.2 Three-dimensional (3D) Case

The surface extraction algorithm also works on intersection vertices in the 3D case. In the 3D case, the intersection zone is a volume centered at the intersection vertex. The size of the intersection volume is equal to the size of a voxel and it overlaps eight voxels at a time. The intersection volume configuration is defined by whether each of these 8 voxels are filled or not, so there are $2^8 = 256$ possible intersection volume configurations.

Configurations must be normalized before the surface extraction process is applied to the intersection volume similar to the normalization performed in the 2D case. In the 3D case, however, each one of the filled voxels in a normalized intersection volume must share a face with another filled voxel in the same normalized intersection volume rather than an edge as in the 2D case (see Figures 3.9 and 3.10). Computing whether a face is shared between two voxels is not a difficult task. Index fields of the voxel indices can be used to determine what is shared among two voxels that are in the same intersection volume:

- the two voxels share a face if two of the index fields are the same, e.g., the x-index and z-index fields of voxel indices of both voxels are equal;
- the two voxels share an edge if one of the index fields are the same;
- the two voxels share a vertex if none of the index fields are the same.

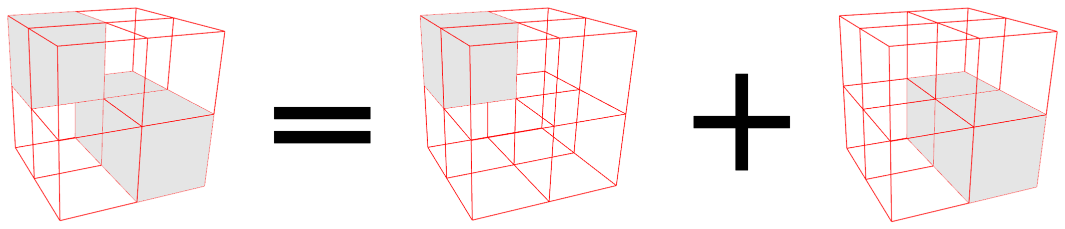


Figure 3.10: Normalization of an unnormalized voxel intersection volume in the 3D case.

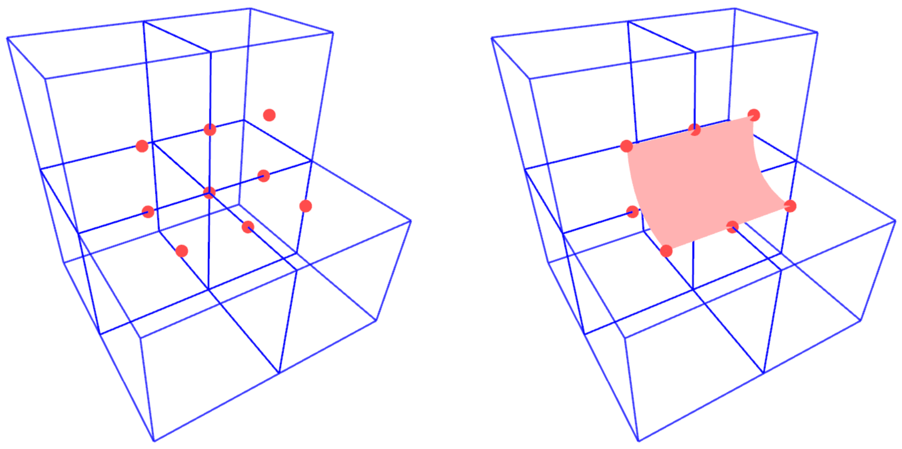


Figure 3.11: A sample normalized intersection volume configuration where six of the voxels are filled and the surface generated for this configuration.

In the 3D case, biquadratic Bézier surfaces are used to generate the surface for each intersection volume. $3 \times 3 = 9$ control points are required to define a biquadratic Bézier surface where eight of the control points are on the edges of the surface and one is on the interior (see Figure 3.11). The position of the vertices that approximate the surface of a biquadratic Bézier surface can be computed by using the parametric surface Equation (3.7) where $P_{00} \ldots P_{22}$ are the 9 control points that define the surface.

$$p(u, v) = \sum_{i=0}^{2} \sum_{j=0}^{2} \binom{2}{i} (1-u)^{2-i} u^i \binom{2}{j} (1-v)^{2-j} v^j P_{ij} \tag{3.7}$$

It should be noted that the algorithm does not generate a surface if all the voxels in an intersection volume are filled or all of them are empty. For all the other cases, there must be at least one Bézier surface for the intersection volume.

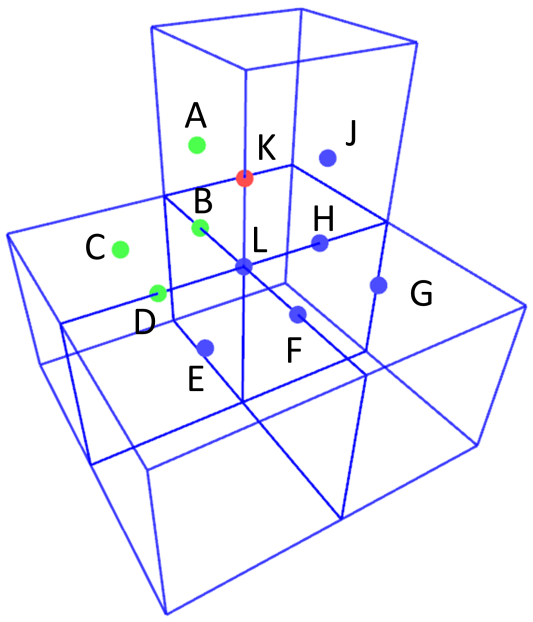
For some configurations, however, one surface does not suffice for a connected terrain surface and up to three Bézier surfaces may be required. This happens whenever there are more than four edges that needs to be included in the surface. In Figure 3.12, for instance, there are five edges that need to be patched up:

- Edge 1: A B C,
- Edge 2: C D E,
- Edge 3: E F G,
- Edge 4: G H J,
- Edge 5: J K A.

It is not possible to patch up five edges using a single biquadratic Bézier surface since a biquadratic Bézier surface has only four edges. In this case, the solution that our approach uses is to patch up the five edges using three different biquadratic Bézier surfaces as it is shown in Figure 3.12. Two of these surfaces patch up two external edges and one internal edge while one of the surfaces patch up one external edge and two internal edges. External edges are the edges that are shared between surfaces of different intersection volumes. Internal edges are introduced when there are multiple surfaces in an intersection volume and the edge(s) that are shared by the surfaces in the same intersection volume are called internal edges. The nine control points of each of these three surfaces are:

- Surface 1: {A, B, C}, {A, L, D}, {A, L, E}
- Surface 2: {J, H, G}, {J, L, F}, {J, L, E}
- Surface 3: {A, L, E}, {K, L, E}, {J, L, E}

Note how internal edges {A, L, E} and {J, L, E} are shared between different surfaces. This ensures connectivity of these surfaces. Also note that for some configurations it is essential to collapse an edge of the Bézier surface in order to obtain a surface with three edges (kind of a triangle instead of a quad). In this case all three of the surfaces are generated in that way (see Figure 3.13 for a different example). Three of the control points of surface 1, for instance, are all



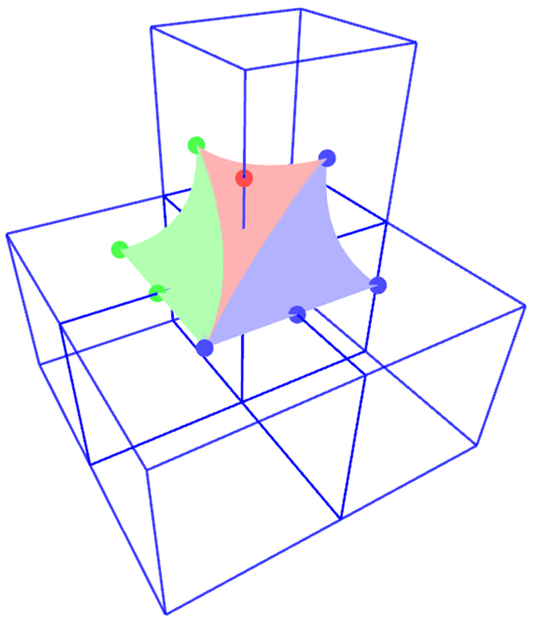


Figure 3.12: An instance of voxel intersection volume configuration where three biquadratic Bézier surfaces are required to generate a connected surface.

A in which case one of the edges of surface 1 is essentially collapsed into a single point. It is not a problem, though, since it is still possible to map this surface to 2D planar (u, v)-coordinates using the parametric Equation (3.7).

It is possible to compute a flag value of 8-bits that represents the intersection volume configuration. Each bit of the flag value represents whether the corresponding voxel is filled or not. This flag value can then be used as a pointer to a lookup table to retrieve the precomputed control points for the surface of that intersection volume configuration. There can be 9, 18 or 27 precomputed control points for each intersection volume configuration depending on the number of surfaces defined for that case. It is of utmost importance for the connectivity and continuity of the terrain surface that the control points of the adjacent intersection volumes coincide. Furthermore, if a configuration contains multiple Bézier surfaces, then the control points of each edge must either be shared with another surface defined for the same configuration or must be shared with a surface defined for an adjacent intersection volume configuration (see Figure 3.12). Algorithm 3.1 is used to extract the surface of the volumetric terrain representation.

Each of the generated surfaces is added to the list of Bézier surfaces that is stored in one of the voxels in the corresponding intersection volume (as a

```
voxelModel: (input) The voxel representation of the terrain
surfaces  : (output) The list of surfaces, which is stored per-voxel
begin
    foreach voxelIndex in voxelModel do
        // construct the voxel intersection volume
        voxelIndices[8]
        for $i \leftarrow 0$ to 8 do
            // >> is the right-shift operator
            // % is the modulus operator
            voxelIndices[i] = getNeighborVoxelIndex(
              i >> 2, (i >> 2) % 2, i % 2)
        // normalize voxel intersection volume
        normalizedVoxelGroups = normalize(voxelIndices)
        foreach voxelGroup in normalizedVoxelGroups do
            fillFlag = 0
            for $i \leftarrow 0$ to 8 do
                // |= is the bitwise-or assignment operator
                // << is the left-shift operator
                fillFlag |= voxelGroup.isVoxelFilled(i) << i
            // get control points for each surface
            surfaces = getSurfaces(fillFlag)
            // get voxel pointer
            voxel = getVoxel(voxelIndex)
            foreach surface in surfaces do
                voxel.surfaces $\leftarrow$ surface
```

Algorithm 3.1: Surface extraction algorithm

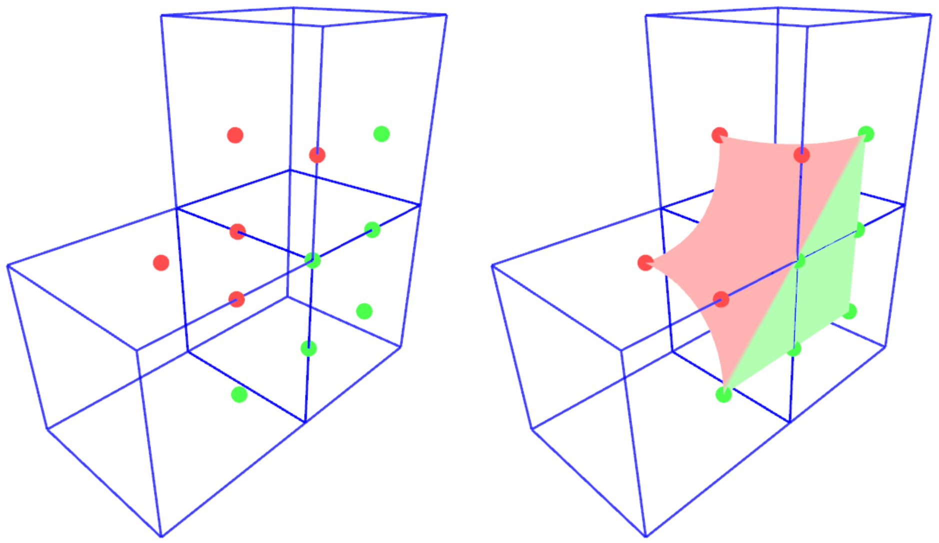


Figure 3.13: A sample normalized intersection volume configuration where three of the voxels are filled and the surface generated for this configuration.

convention, our approach stores the surfaces generated for an intersection volume in the voxel whose index fields in the voxel index have the minimum values). This choice does not really matter as long as it is consistent. By doing so, it is ensured that the adjacent voxels store the surfaces for the adjacent intersection volumes.

Although there are 256 different possible voxel intersection volume configurations, most of these configurations are related to a unique base configuration in one of the following ways:

- symmetric about the xy-, xz-, or yz-planes, and
- rotated by 90, 180 or 270 degrees around either x-, y-, or z-axes.

The surfaces for such configurations can be obtained by applying affine transformations to the control points of the Bézier surfaces of the corresponding unique base configuration (cf. Section 3.4). Applying such an affine transformation to the control points also correctly transforms the surface definition. Of all the 256 possible configurations, there are only 12 such unique configurations excluding the cases where no surface is generated (i.e., where either all or none of the voxels in the intersection volume are filled). These unique configurations, the surfaces

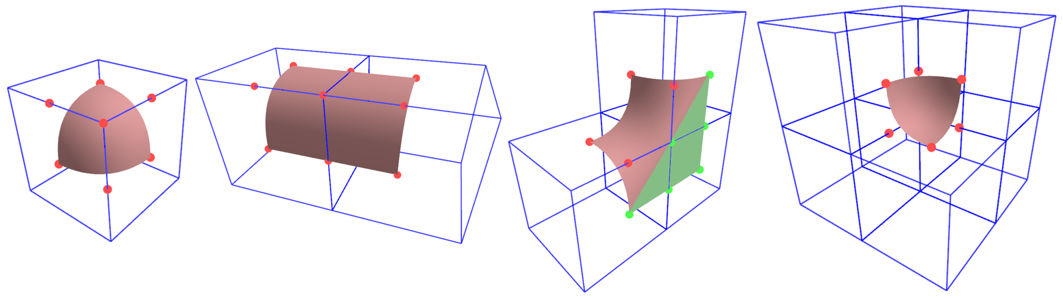


Figure 3.14: Unique voxel intersection volume configurations where, respectively, 1, 2, 3 and 7 of the voxels in the intersection volume are filled.

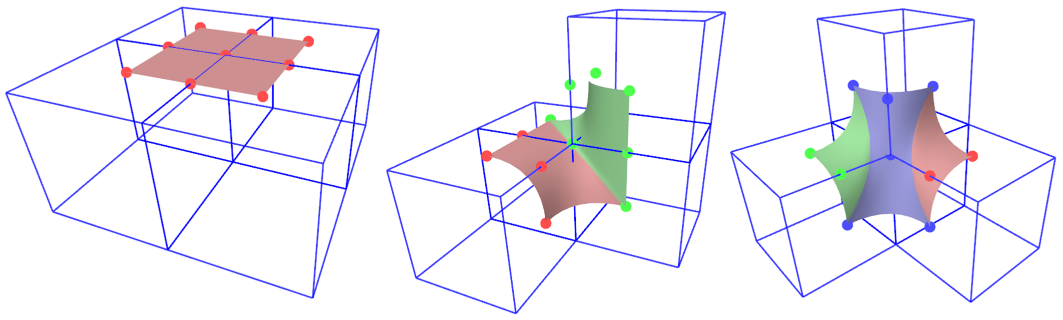


Figure 3.15: Unique voxel intersection volume configurations where four of the voxels in the intersection volume are filled.

generated for each of them and the control points for the surfaces are shown in Figures 3.14, 3.15, 3.16, and 3.17.

3.4.2.1 Handling Voxels at Different Levels

The surface extraction method presented here assumes that the size of all voxels that are part of an intersection volume are equal, that is, they are at the same level in the octree. In practice, however, this may or may not be true since the octree allows neighboring voxels to be at different levels. To be able to handle these cases, the surface extraction algorithm computes the minimum level of the voxels in the intersection volume and then generates several new smaller intersection volumes on the surface of each voxel that are at a higher level than the minimum level (see Figure 3.18). Then the surfaces for these smaller intersection volumes are generated rather than the actual larger one.

The new and smaller intersection volumes are generated only for the 3 faces

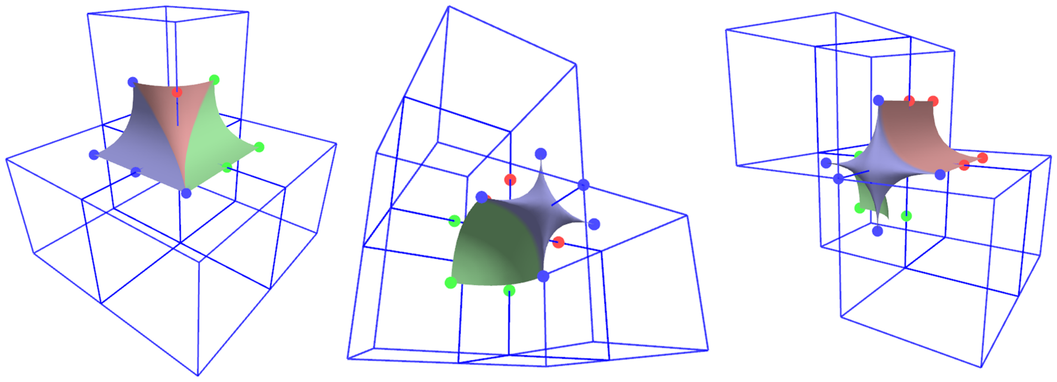


Figure 3.16: Unique voxel intersection volume configurations where five of the voxels in the intersection volume are filled (please note that the last two pictures show the same configuration from different viewpoints).

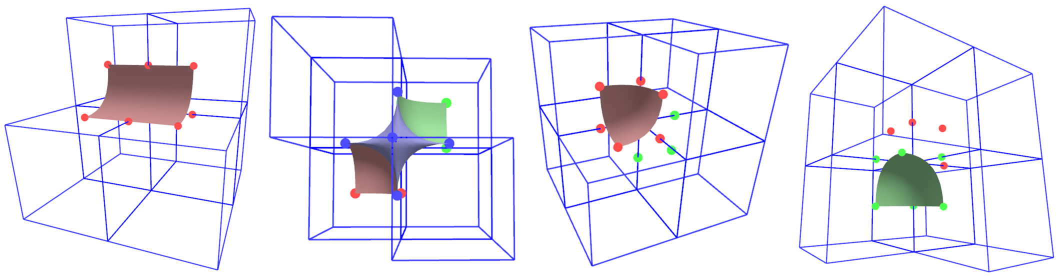


Figure 3.17: Unique voxel intersection volume configurations where six of the voxels in the intersection volume are filled (please note that the last two pictures show the same configuration from opposite viewpoints).

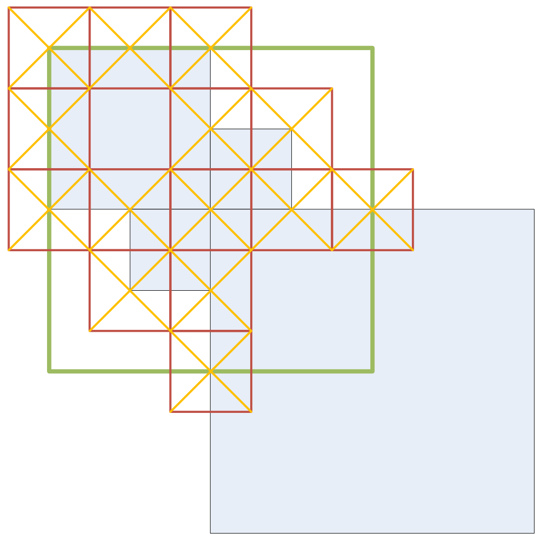


Figure 3.18: The division of the actual intersection volume (green rectangle) to several new and smaller intersection volumes (red squares with yellow X inside) as seen from a top-down view.

of larger voxels that overlap the actual intersection volume. The inner volume of these larger voxels are not processed since no surface is meant to be generated inside a voxel: even if such an intersection volume was generated, either all or none of the voxels would be full and as a result no surface would be generated.

Note that the larger voxels are not actually split into smaller voxels, though. The representation of the octree in does not change in memory. The smaller intersection volumes are generated and processed on-the-fly and the queries for smaller virtual sub-voxels are answered considering the larger voxel of which they are a part. Simply, if the larger voxel is filled then all smaller virtual sub-voxels that are a part of it are also filled and vice versa.

The result of the surface extraction method in the 3D case for a sample voxel representation is shown in Figures 3.19 and 3.20.

3.4.2.2 Static Surface Culling

The presented voxel surface extraction algorithm generates surfaces considering each and every voxel intersection volume. This method results in generation of redundant surfaces that are never meant to be visible in practice. Think of the

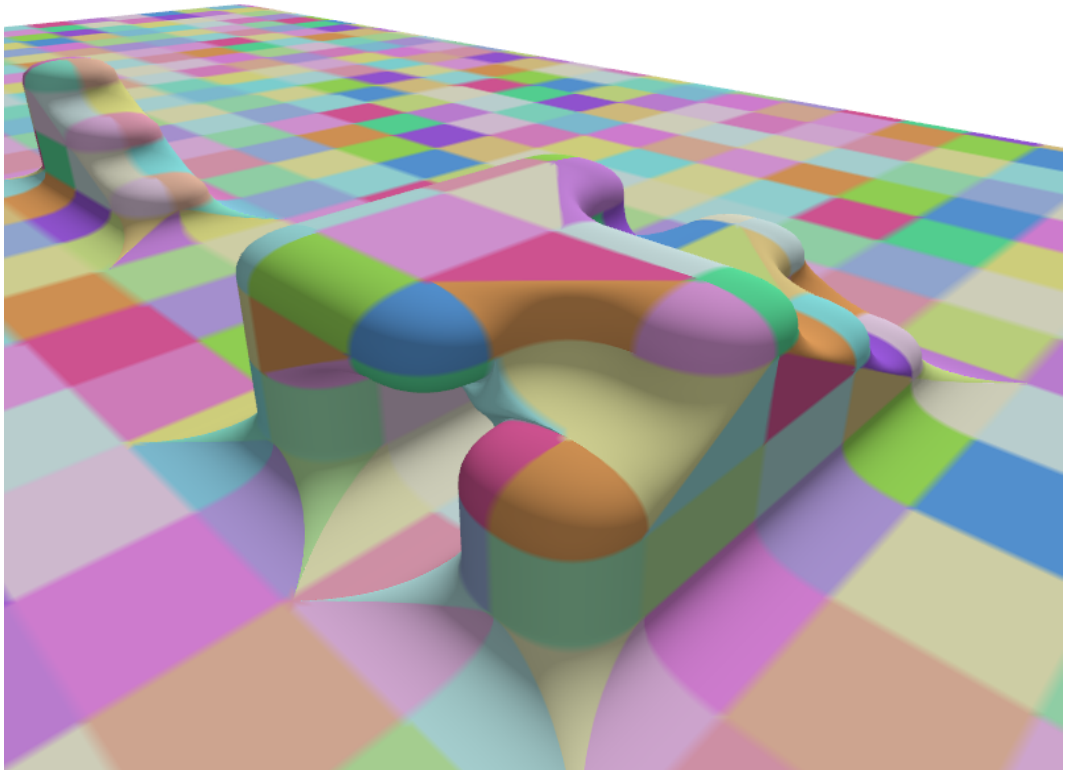


Figure 3.19: The final result of the surface extraction algorithm (patching surfaces are rendered in different colors).

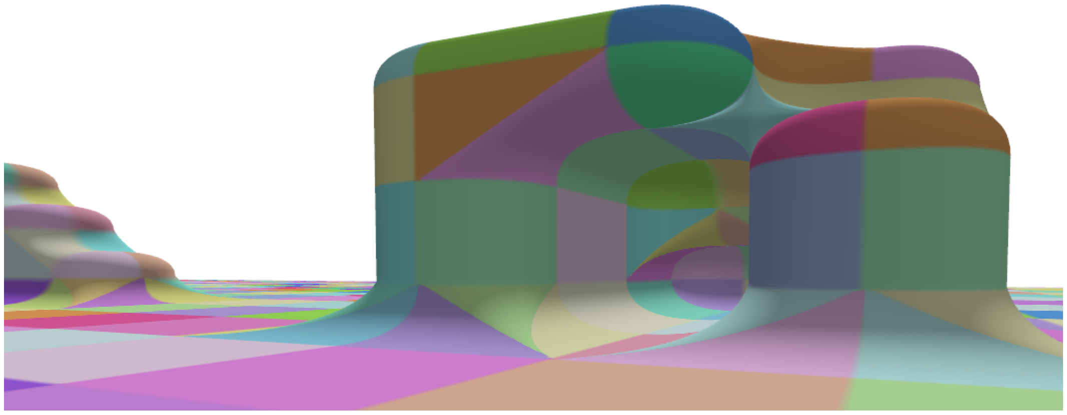


Figure 3.20: The final result of the surface extraction algorithm from another viewpoint (patching surfaces are rendered in different colors).

entire voxel model as a rectangular prism, for instance. Surfaces for each of the six faces of the rectangular prism will be generated by the algorithm since it is designed to generate connected and continuous surfaces without any holes. In reality, however, the surfaces generated at the four sides and at the bottom of the rectangular prism would never be visible to the observer, since the actual terrain is on the top face of the rectangular prism. Thus, it is possible to optimize the output of the surface extraction algorithm by culling these redundant surfaces early in the surface extraction process.

The purpose of the static surface culling is to significantly decrease the number of surfaces generated by discarding the surfaces that are never meant to be visible in the first place in order to improve the performance and efficiency of the terrain generation, editing and visualization. It is, fortunately, quite an easy task. The axis-aligned bounding box of the 3D voxel space is already determined at the creation time of the octree. As soon as a surface is generated, its control points are checked against the bounding box to find out if the surface is redundant and should be discarded. If all control points are either on the bottom face or one of the side faces of the bounding box then the surface is considered redundant and discarded. As a result of this process the extracted surface is not connected anymore, but the visible part of the surface is still connected and continuous, which is what matters for our purposes. In theory this process can discard as much as 83% of the surface generated (e.g., five out of six faces of the rectangular prism is culled). In practice the gains are lower since the top face of the rectangular prism is much more complex structurally, as it contains all the detail about the terrain, and therefore contains many more surfaces than the other faces. Our experiments have shown that about 40% to 55% of the generated surfaces can be safely discarded by this process on typical terrains. See Figure 3.21 for an example application of the static surface culling. For this sample voxel model, the number of generated surfaces is 8962 without culling and 3948 with culling. In this case, the static surface culling process managed to discard 56% of the surfaces as they are not meant to be visible. Discarding so many surfaces significantly improves the performance and decreases the memory consumption of the application overall without affecting the output visually in any way.

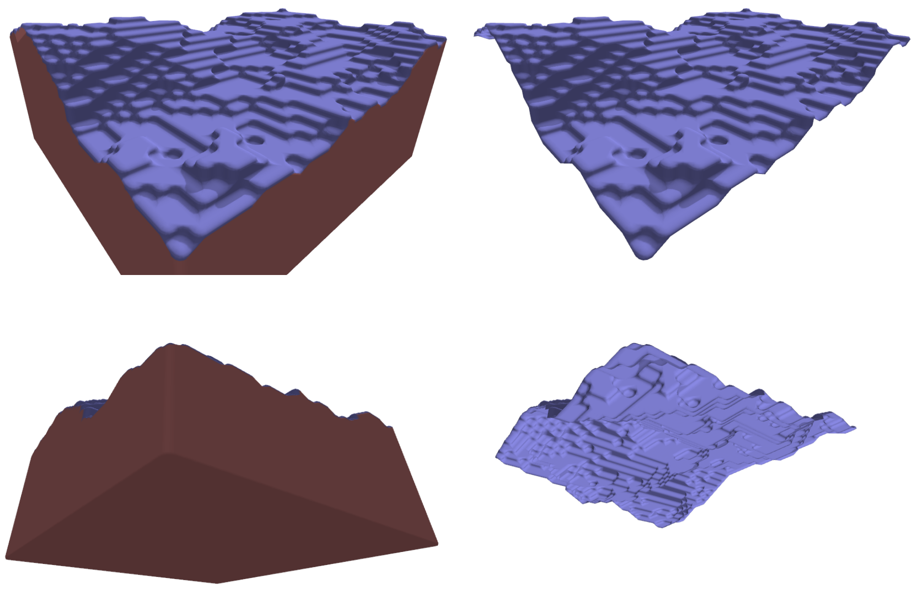


Figure 3.21: The redundant surfaces are tinted with brown. Top-Left: top-down view without static surface culling applied, top-right: top-down view with static surface culling applied, bottom-left: bottom-up view without static surface culling applied, bottom-right: bottom-up view with static surface culling applied (back-face culling is disabled for visualization).

3.5 Terrain Surface Generation

The 3D voxel model defines the coarse surface features of the terrain. Each voxel is related to a number of surface patches that consist in the complete terrain surface. These surface definitions are parametric definitions and they must be approximated by a finite number of vertices and faces before they can be used to visualize the surface using hardware acceleration. The process of approximating the parametric surfaces by a finite number of primitives and applying heightmaps to each surface patch in order to obtain the ultimate terrain surface is called terrain surface generation (cf. Section 3.4).

3.5.1 Generating Vertices

The first step of terrain surface generation is to generate a number of vertices for each Bézier surface (i.e., terrain surface patch). The minimum number of vertices to approximate each surface patch is nine in which case each vertex coincide with one of the surface control points. The actual number of vertices per surface patch is determined by the desired level-of-detail and is constrained by the memory available for storing these vertices. As the number of vertices generated per surface patch increases

- the approximation converges to the actual surface and as a result a rather smoother approximation is achieved, and
- the resolution of the surface patch increases allowing us to apply higher-resolution heightmaps to the patches and achieve a more-detailed terrain surface.

At this stage of the surface generation, only the original positions of the vertices are computed, rather than the displaced positions, since the heightmaps will be applied at a later stage (cf. Section 3.3.3). Computing the positions of vertices is straightforward using the parametric surface equation of biquadratic Bézier surfaces (see Equation (3.7)). First, the (u, v)-coordinates for each vertex are generated assuming a uniform grid of vertices. Then these (u, v)-coordinates are used to compute the corresponding position on the surface by the parametric surface equation. In the proposed surface generation method, the number of vertices in each edge of the surface are equal meaning that the u and the v subspaces are sampled at the same frequency. Thus, the total number of vertices on a surface patch is N^2 where the number of vertices per edge is N. In theory, it is possible to use different frequencies for u and v subspaces where edges that align with one axis has N vertices and the other has M vertices, resulting in a total vertex count of $N \times M$ per surface patch. This usage has an undesired side-effect of vertex alignment problems in neighboring surface patches that share an edge, though, e.g., when one edge with N vertices coincide with an edge of M vertices of another surface patch. It is ensured, therefore, that the vertices on the edges of neighboring surface patches align correctly by generating N vertices on each edge of the surface patches (please note that this constraint is later relaxed for level-of-detail purposes, but for now we shall assume that it holds).

Each vertex generated for a surface patch can be categorized as either an internal vertex or a border vertex (see Figure 3.22).

Border vertices are vertices that are part of at least one of the edges of the surface patches. These vertices can easily be classified based on their assigned planar (u, v)-coordinates:

- $(u = 0,\, v = 0)$: First corner
- $(u = 1,\, v = 0)$: Second corner
- $(u = 1,\, v = 1)$: Third corner
- $(u = 0,\, v = 1)$: Fourth corner
- $(0 < u < 1,\, v = 0)$: First edge
- $(u = 1,\, 0 < v < 1)$: Second edge
- $(0 < u < 1,\, v = 1)$: Third edge
- $(u = 0,\, 0 < v < 1)$: Fourth edge

Internal vertices are all vertices of a surface patch except the border vertices. These vertices are located in the inner-area of the surface patch and have planar coordinates such that $(0 < u < 1,\, 0 < v < 1)$.

Border vertices are shared vertices, meaning that multiple vertices spatially coincide and are located at exactly the same coordinates in the 3D working space.

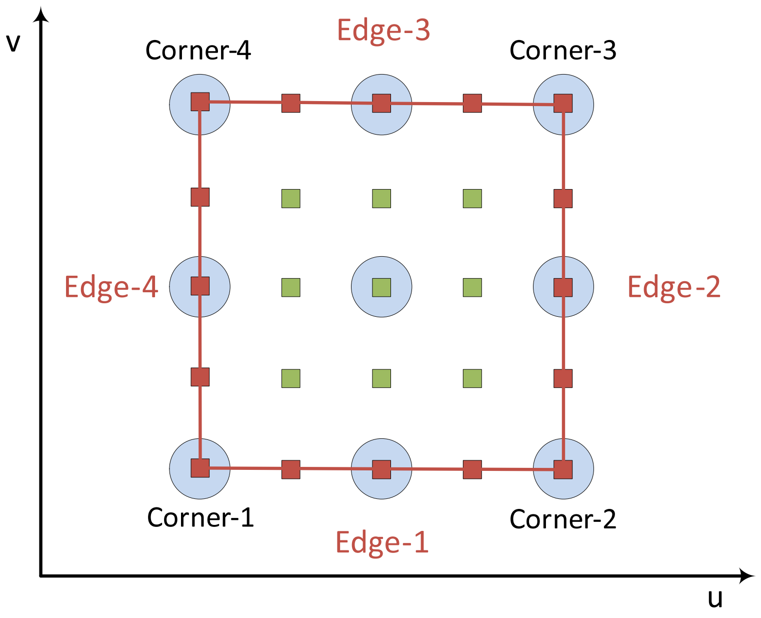


Figure 3.22: Vertices that approximate a surface are shown where $N = 5$. Red lines are the edges, red boxes are the border vertices, green boxes are internal vertices and blue circles are control points of the surface.

The vertices can be shared externally or internally relative to the surface patch for which the vertex is generated.

External sharing of a vertex means that multiple vertices that belong to different surface patches are located at the same coordinates.

Internal sharing of a vertex means that multiple vertices that belong to the same surface patch are located at the same coordinates.

Both external and internal sharing happen on the edges and corners of surface patches. Thus, they are only defined for border vertices and not for internal vertices. Internal vertices are never shared and are called unique vertices as each of them is the only vertex located at that specific coordinate.

3.5.1.1 Internal Sharing of Vertices

Internal sharing of border vertices occurs whenever one edge of a surface patch is collapsed into a single point such that there are three edges instead of four.

In this case, all control points of the surface that are on the collapsed edge are located at the same coordinates (cf. Section 3.4.2). Recall that all edges of a Bézier surface are Bézier curves defined by the control points on that edge (cf. Section 3.4). When all control points of an edge collapse into a single point, the curve defined by that edge also collapses into a single point. Thus, all vertices that are supposed to approximate that curve are located at the same coordinates, that is, they are shared.

The separate storage of internally shared vertices is redundant. It is easy to find out if any of the edges of a surface patch is collapsed (see Figure 3.23):

- If the seventh and the ninth control points are at the same location then the first edge is collapsed.
- If the third and the ninth control points are at the same location then the second edge is collapsed.
- If the first and the third control points are at the same location then the third edge is collapsed.
- If the first and the seventh control points are at the same location then the fourth edge is collapsed.

Please note that it suffices to only check the locations of the control points that are on the corners of the surface to find out if an edge is collapsed. This is due to the fact that there is no case in the presented surface extraction algorithm where the two cornering control points are at the same location while the middle control point is in another location. Thus, if two control points on adjacent corners are in the same location then it implies that the control point in-between them is also at that same location.

If an edge of the surface patch is determined to be collapsed by the mentioned rules then only one vertex is generated on that edge and the generation of the other vertices for that edge are skipped.

3.5.1.2 External Sharing of Vertices

External sharing occurs on two occasions:

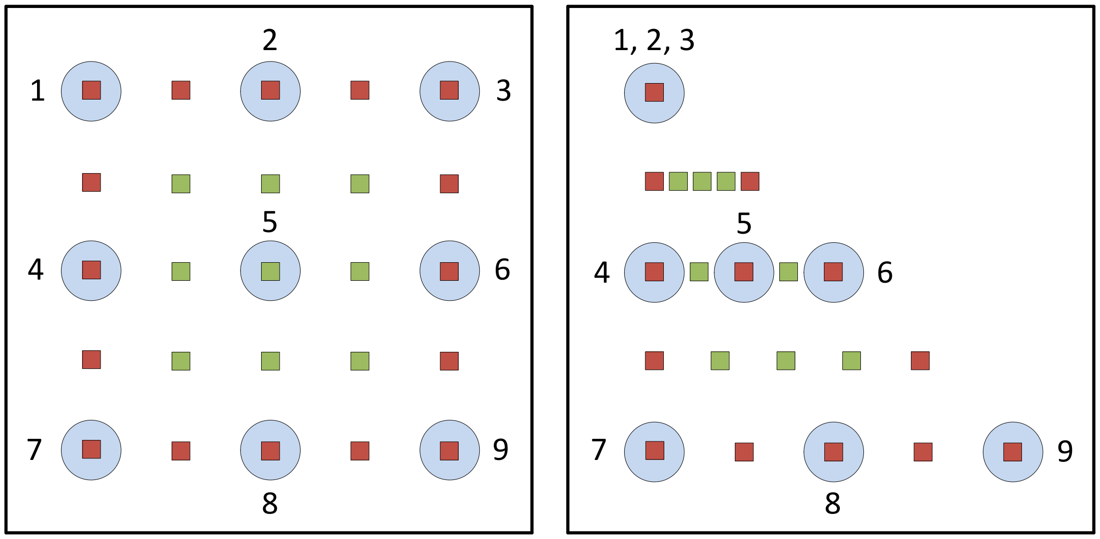


Figure 3.23: Left: a surface without internal sharing of vertices, right: a surface where the third edge is collapsed and the vertices on the third edge are internally shared (numbered are the control points of the surface).

- on the common edge of two neighboring surface patches, and
- on the common corners of multiple neighboring surface patches.

Vertices that are strictly on the edges of the surface patches (and not on the corners) are shared by exactly two surface patches since each edge of a surface patch can coincide with an edge of one and only one other surface patch (cf. Section 3.4.2). Vertices that are on the corners, on the other hand, are shared among at least four surface patches. In some configurations the number of sharing surface patches of a corner vertex can be many more than four (see Figure 3.24).

Unlike internally shared vertices, it is not possible to detect externally shared vertices before they are generated. This is due to the requirement that the location of the vertices must be known for comparison with the locations of the border vertices of neighboring surface patches. As a result, all vertices of a surface patch, except the internally shared ones, are generated first and then the ones that are found to be externally shared are discarded, except the one and only instance of the shared vertex. The externally shared vertices can only be shared by neighboring surface patches and neighboring surface patches must be contained by neighboring voxels.

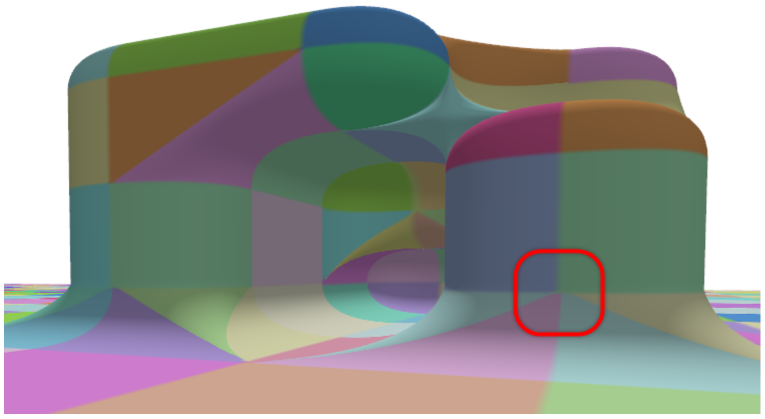


Figure 3.24: An externally shared vertex is circled with red where it is shared by six neighboring surface patches.

In order to find out the externally shared instances of a vertex that is part of a surface patch P, therefore, it suffices to look at the vertices of the surface patches of the current voxel P and the neighboring voxels of P. It should be noted that there may be many surface patches related to a voxel depending on the level of that voxel and the neighboring voxels in the octree (cf. Section 3.4.2.1). To improve the performance of finding externally shared voxels, first the bounding boxes of the surface patch pairs of the neighboring voxels are compared to see if there is an overlap. Externally shared voxels are searched for only if the bounding boxes overlap, since when the bounding boxes do not overlap there cannot be any shared vertices between these surface patches. Another performance optimization is possible by checking only border vertices of the surface patches instead of all vertices. Checking against internal vertices would be redundant because

- two internal vertices of different surface patches cannot be at the same location, and
- an internal vertex and a border vertex of different surface patches cannot be at the same location.

Therefore, when searching for externally shared vertices of two surface patches, only the locations of the border vertices of each surface patch are compared with each other. This reduces the runtime complexity of the algorithm from $\mathrm{O}(V^2)$ to $\mathrm{O}(V)$, where V is the number of vertices of a single surface patch.

3.5.1.3 Storing Vertices

There are a total of N^2 vertices in a surface patch where a single edge is approximated by N vertices. Of those N^2 vertices, $4 \times (N-1)$ are shared (i.e., border) vertices. The attributes of these shared vertices should not be stored separately in each one of the surface patches that share them because

- storing the same attributes separately is redundant and causes the application to use more memory space unnecessarily,
- the displaced positions of these vertices may differ since different heightmaps are applied to different surface patches causing gaps in the terrain surface, and
- storing these attributes separately causes the surface normals to be computed incorrectly as it is explained in the next section.

Thus, for correct operation and efficiency of the terrain surface generation algorithm, shared vertices must be stored in a common data structure used by different surface patches that share the vertex. The number of surface patches that share a border vertex is not a constant as it can be as few as one in the case where all neighboring surface patches are culled by static surface culling, and it can be quite a few patches depending on the voxel intersection volume configurations around the vertex. A linked list is a good choice for storing the shared vertex data where each node of the linked list represents an instance of the shared vertex and each node is stored by a surface patch that shares the vertex.

Each shared vertex list node stores a pointer to the first and next node in the list, a pointer to a surface patch that shares the vertex and a pointer to the actual vertex where the vertex attributes is stored. The number of shared vertex list nodes for a shared vertex is always equal to the number of surface patches that share that vertex. Initially, that is at the stage of vertex generation, a separate node is created for each border vertex. A vertex is created with each node and each node points to a distinct vertex. Once all vertices are generated for all surface patches, the algorithm searches for externally shared vertices by comparing the vertices of neighboring surface patches. The locations of all border vertices of the neighboring surface patches are compared and if two vertices are

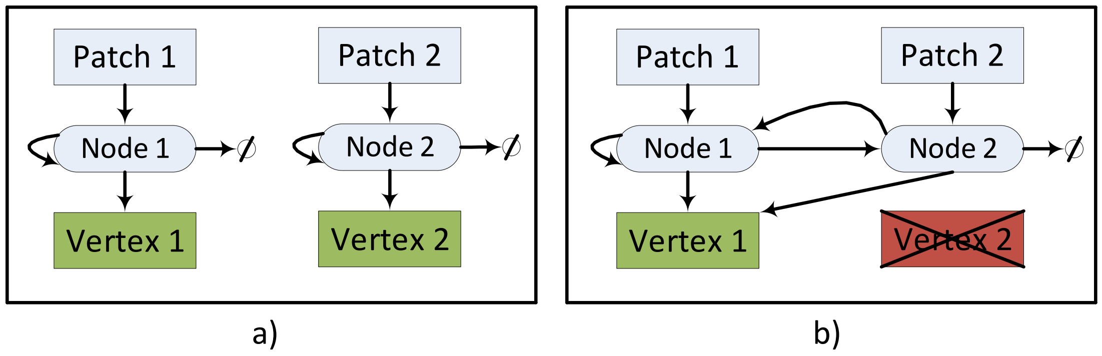


Figure 3.25: (a) two distinct shared vertex lists, each having one node and pointing to different vertices, (b) the shared vertex lists are merged into a single shared vertex list, all of the nodes now point to the same vertex and the remaining vertices (vertex 2 in this case) are deleted.

in the same location then their shared vertex lists are merged (see Figure 3.25). After the merge operation, all nodes in the two former shared vertex lists are now in the same shared vertex list and all nodes point to the same vertex data.

It should be noted that it is essential for the presented approach that the mapping between the vertices of a surface patch and a 2D plane is maintained. The vertices of a surface patch must, therefore, be quickly accessible with a 2D vertex index in the format (i, j) as this representation provides a mapping between the vertices and the 2D (u, v) coordinates of the heightmap (this relation is given by Equations (3.8) and (3.9)). The range of both i and j is $[0, N-1]$ where N is the number of vertices per edge.

$$u = i/(N-1) \tag{3.8}$$

$$v = (N-1-j)/(N-1) \tag{3.9}$$

In order to allow fast access to vertices through 2D vertex indices a one-dimensional array of vertex pointers of size N^2 is created and filled with pointers to the corresponding vertex attributes. The one-dimensional array index is then computed as $i + j \times N$. It should be noted that different indices of the array can store pointers to the same vertices where the vertices are shared externally or internally. The attributes of the shared vertices, however, are strictly stored at one place in the main memory without any copies, although there may be many

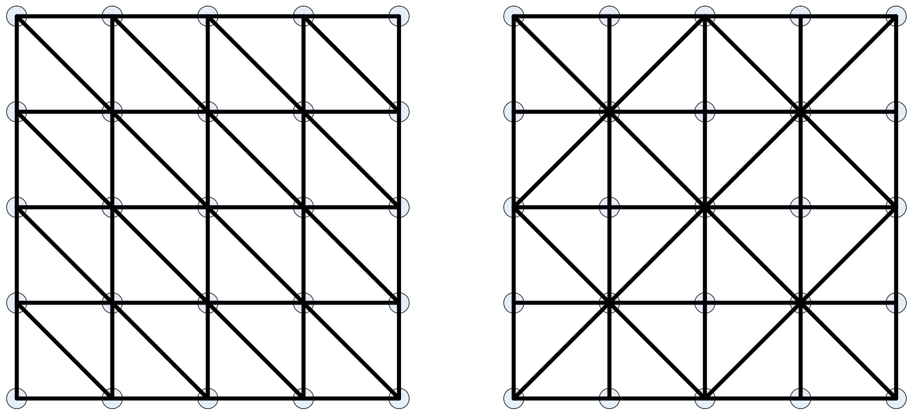


Figure 3.26: Two different patterns for connecting vertices of a surface patch ($N = 5$) to form triangles. The pattern on the right yields better results when used with level-of-detail algorithms. Please note that both patterns result in the same number of triangles albeit in different orientations.

pointers from several surface patches to a particular vertex.

3.5.2 Generating Faces

The next step of terrain surface generation is the generation of faces. The most efficient primitive type of faces in hardware accelerated rasterization-based renderers is triangles. In this stage, the vertices generated for each surface patch are connected in groups of three such that they form triangles that approximate the surface without any holes and cracks. It is possible to use different connection patterns for this task (see Figure 3.26). Although the pattern shown on the left is quite straightforward and seems to do the job of connecting vertices to form triangles, the pattern on the right yields better results when level-of-detail is introduced to the presented approach. A more-detailed explanation about this is given in the next sections. Thus, the presented approach uses the pattern shown on the right.

Please note that the minimum number of vertices per edge N required by this connection pattern is three (see Figure 3.27). The primitive connection pattern of size 3×3 can then be tiled in both the u- and the v-axes for $\frac{N-1}{2}$ times. This places a constraint on the number of vertices per edge in a surface patch such that $N = 2 \times k + 1$ where k is a positive integer. The primitive pattern is then tiled

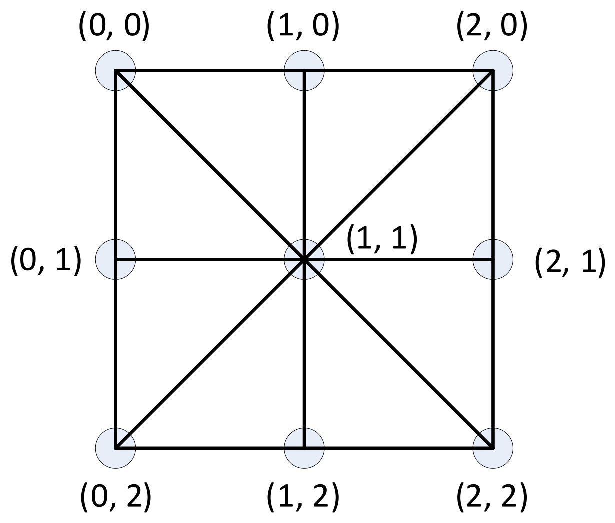


Figure 3.27: Primitive vertex connection pattern for batches of 3×3 vertices. The vertices are annotated with 2D vertex indices in (i, j) format.

k times in both the u- and the v-axes. The total number of triangles generated for a surface patch is $8 \times k^2$ which is equal to $2 \times (N-1)^2$.

The presented approach does not require the explicit storage of the triangle vertex indices as they can easily be computed on-the-fly whenever required with the help of a static lookup table that defines the primitive connection pattern of the vertices. (see Algorithm 3.2). It should be noted that each vertex is shared by multiple triangles not only those belong to the same surface patch but also by triangles of different surface patches (through shared vertex lists). It is now possible to render the surface of the terrain using a hardware-accelerated rasterization-based renderer by sending the vertices to the renderer in this particular order and using triangles as the rendering primitive.

It is extremely important that the triangles are always constructed in a way such that the vertices of the triangle are ordered in a counter-clockwise direction. This must be ensured for correct computation of the face and vertex normals as well as correct visualization of the terrain surface. This is explained in more detail in the next section about face and vertex normals.

```
triangleIndex          : (input) The zero-based index of the triangle
numVerticesPerEdge: (input) Number of vertices per edge ($N$)
vertexIndices          : (output) The vertex indices of the vertices of the
                         triangle in $(i, j)$ format
begin
    const indices[8][3][2] = {
        { {0, 0}, {0, 1}, {1, 1} }, // face-1
        { {1, 1}, {0, 1}, {0, 2} }, // face-2
        { {0, 2}, {1, 2}, {1, 1} }, // face-3
        { {1, 1}, {1, 2}, {2, 2} }, // face-4
        { {2, 2}, {2, 1}, {1, 1} }, // face-5
        { {1, 1}, {2, 1}, {2, 0} }, // face-6
        { {2, 0}, {1, 0}, {1, 1} }, // face-7
        { {1, 1}, {1, 0}, {0, 0} }, // face-8
    };
    // % is the modulus operator;
    // << is the left-shift operator;
    // >> is the right-shift operator;
    // & is the bitwise-and operator;
    numFacesPerEdge = numVerticesPerEdge >> 1;
    i = (batchIndex % numFacesPerEdge) << 1;
    j = (batchIndex / numFacesPerEdge) << 1;
    k = triangleIndex & 0x7;
    for $v \leftarrow 0$ to 3 do
        vertexIndices ← (i + indices[k][v][0], j + indices[k][v][1]);
```

Algorithm 3.2: Generating triangle vertex indices given the index of the triangle

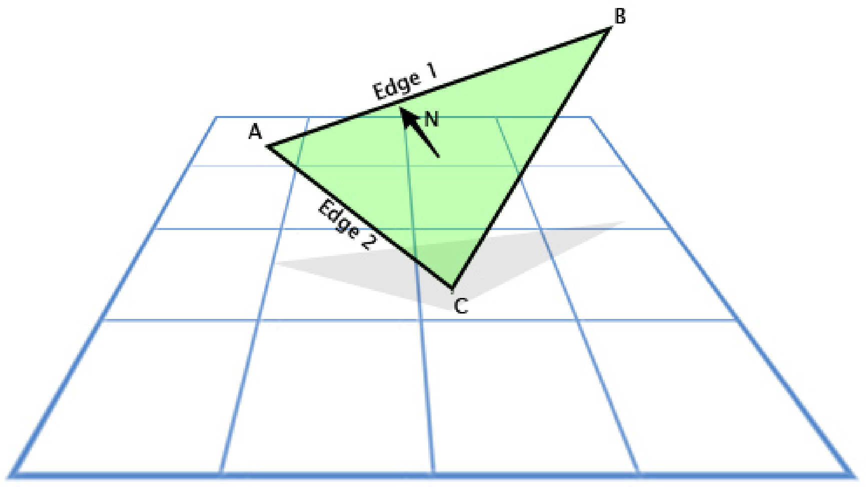


Figure 3.28: The normal vector $\overrightarrow{N}$ of the triangle BAC where $\overrightarrow{AB}$ is the first edge and $\overrightarrow{AC}$ is the second edge.

3.5.3 Computing Face and Vertex Normals

The normals in 3D models are usually computed per face since it is an attribute defined for flat surfaces such as triangles. A vector is called the normal vector of a triangle if it is perpendicular to the triangle (see Figure 3.28). The normal vector $\overrightarrow{N}$ of a triangle whose vertices are at locations $\overrightarrow{P_1}$, $\overrightarrow{P_2}$ and $\overrightarrow{P_3}$ can be computed easily according to Equation (3.10). It is crucial for correct computation of the triangle normal that the vertices $\overrightarrow{P_1}$, $\overrightarrow{P_2}$ and $\overrightarrow{P_3}$ are in counter-clockwise orientation, otherwise the computed normal will point to the opposite direction since we take the cross-product of the edge vectors and the order of the edge vectors determine the direction of the normal vector.

$$\begin{aligned} \overrightarrow{v_1} &= P_2 - P_1 \\ \overrightarrow{v_2} &= P_3 - P_2 \\ \overrightarrow{N} &= \overrightarrow{v_1} \times \overrightarrow{v_2} \end{aligned} \tag{3.10}$$

The computed normal vector is usually normalized such that its length is equal to one before it is used in further processing (see Equation 3.11 where l is the length of $\overrightarrow{A}$ and $\overrightarrow{B}$ is the normalized form of $\overrightarrow{A}$). This is due to the fact that usually the length of the normals is not interesting as they merely define a direction.

$$\begin{aligned} l &= \sqrt{A_x \times A_x + A_y \times A_y + A_z \times A_z} \\ \overrightarrow{B} &= \overrightarrow{A} \times \frac{1}{l} \end{aligned} \tag{3.11}$$

The computation of triangle normals are discussed so far, rather than the vertex normals. The presented approach requires vertex normals to be computed since most operations are performed per-vertex and not per-triangle, such as the displacement operation by the heightmap application. During the displacement, the triangles are not displaced as a whole, but rather the individual vertices are displaced depending on the values of the applied heightmap. Thus, the vertex normals are required to compute the displacement of each vertex. In the proposed approach each vertex is supposed to have two kinds of normals assigned to it:

1. The displacement normal merely defines the direction through which the vertex is displaced according to the applied heightmap values. The displacement normal depends only on the voxel definition of the terrain surface and does not change as the vertices are displaced by a heightmap.
2. The surface normal is the actual normal of the vertex that is used for visualization purposes such as lighting and texturing. The surface normal of a vertex changes whenever the displaced position of that vertex or of one of the connected vertices is modified. The surface normals must be recomputed in these cases.

The vertices of a triangle defines a flat surface in 3D. As a result, the normal vector of the triangle is also the normal vector of each of the vertices. There is a problem, though. As it has already been mentioned in previous sections, a vertex can be shared by many triangles that are part of different surface patches. A vertex cannot have more than one normal vector assigned to it; i.e., one for each triangle by which the vertex is shared. The solution is to compute the average of normals of all triangles that share the vertex. This method of averaging normals can yield awkward results when the slope of the neighboring triangles dramatically differ; i.e, in places where the derivative of the slope of the terrain is too high. The proposed surface extraction method produces a smooth terrain surface with the help of Bézier curves, though. As a result, dramatic changes in the slope of neighboring triangles is normally not encountered in the proposed approach.

In our experiments, it is observed that the obtained results are slightly better in most cases if the areas of the triangles that share a vertex are also taken into account while computing the normal of that particular vertex. The area A of a 3D triangle with vertices P_1, P_2 and P_3 can be computed by Equation (3.12).

$$\begin{aligned} \overrightarrow{v_1} &= P_2 - P_1 \\ \overrightarrow{v_2} &= P_3 - P_2 \\ A &= \frac{1}{2} \times |\overrightarrow{v_1} \times \overrightarrow{v_2}| \end{aligned} \tag{3.12}$$

Equation (3.12) implies that the area of a triangle is proportional with the unnormalized normal vector of the triangle. Therefore, if the sum of normal vectors are computed by taking the unnormalized normal vectors then the areas of the triangles are also taken into account. Then the summation of the unnormalized normal vectors can be normalized to obtain the ultimate normal vector for the vertex. Algorithm 3.3 computes the vertex normals of the terrain surface vertices.

The same algorithm is used for computing both the displacement normals and the surface normals of the vertices. Each vertex has two different position attributes and each of these positions are used to compute a different kind of normal for the vertex:

- The original position of the vertex is the position before any displacement is applied to the vertex. This is an intermediate position that is used to compute the displaced position of the vertex. The vertex is not rendered at this location. This position is used to compute the displacement normals of the vertices.
- The displaced position of the vertex is the position after displacement operation is performed on the vertex. This is the ultimate position of the vertex that is used for computing the surface normals of the vertices as well as the visualization of the vertex.

```
   octree      : (input) Octree that defines the terrain model
   vertexNormal: (output) Vertex normals of each vertex
1  begin
2      // initialize all vertex normals to zero
3      resetVertexNormals(octree)
4      // compute the sum of all triangle normals that share each vertex
5      foreach voxel in octree do
6          foreach patch in voxel do
7              for triangleIndex ← 0 to numTrianglesPerSurfacePatch do
8                  vertexIndices = getTriangleVertexIndices(triangleIndex)
9                  for i ← 0 to 3 do
10                     vertices[i] = patch.getVertexByIndex(vertexIndices[i])
11                 $\overrightarrow{v_1}$ = vertices[1] - vertices[0]
12                 $\overrightarrow{v_2}$ = vertices[2] - vertices[1]
13                 $\overrightarrow{N} = \overrightarrow{v_1} \times \overrightarrow{v_2}$
14                 for i ← 0 to 3 do
15                     vertices[i].vertexNormal += $\overrightarrow{N}$
16     // normalize the sum of triangle normals
17     foreach voxel in octree do
18         foreach patch in voxel do
19             for triangleIndex ← 0 to numTrianglesPerSurfacePatch do
20                 vertexIndices = getTriangleVertexIndices(triangleIndex)
21                 for i ← 0 to 3 do
22                     vertices[i] = patch.getVertexByIndex(vertexIndices[i])
23                     x = vertices[i].vertexNormal.x
24                     y = vertices[i].vertexNormal.y
25                     z = vertices[i].vertexNormal.z
26                     len = sqrt($x \times x + y \times y + z \times z$)
27                     vertices[i].vertexNormal /= len
```

Algorithm 3.3: Computing vertex normals of the terrain surface *vertices*

3.5.4 Displacement of Terrain Surface Vertices

The displacement of terrain surface vertices is the last step for the ultimate terrain surface to be obtained. The displacement operation is quite simple once the original positions and the displacement normals of the terrain vertices have been computed as explained in detail throughout this chapter. In this section it is assumed that heightmaps are already somehow prepared and ready for use. In the next chapters a sample editor application and several methods for editing terrain heightmaps are presented.

The displacement operation is performed for each vertex on each surface patch of the terrain. The output of this process is the displaced positions of the vertices. The displaced positions of the vertices can be computed by Equation (3.13) where $\overrightarrow{P}$ is the original position, $\overrightarrow{N}$ is the displacement normal and $\overrightarrow{D}$ is the displaced position of the vertex. The 2D $h(u, v)$ function is the heightmap function returning a scalar value for planar (u, v) coordinates where the valid range for u and v parameters are [0, 1]. The relation between the 2D vertex indices and planar (u, v) coordinates is given by Equations (3.8) and (3.9).

$$\overrightarrow{D} = \overrightarrow{P} + \overrightarrow{N} \times h(u, v) \tag{3.13}$$

The displacement value of the neighboring vertices should not be too different from each other as this will result in jagged edges and corners on the terrain. In order to increase the visual quality and achieve better realism by generating a smoother terrain surface, Gaussian filter is applied on the heightmap while accessing the heightmap pixels. This filter helps to smooth the extreme values of the pixels in the heightmap. The kernel size dramatically affects the resulting terrain such that

- a larger kernel size will over-smooth the surface hindering the ability of applying fine detail on the terrain surface, and
- a lower kernel size will not be sufficiently effective in reducing the jaggedness of the edges and corners.

Using a larger kernel size for the Gaussian filter also considerably slows down the displacement operation. Our experiments have shown that a Gaussian kernel of size 3×3 (see Equation (3.14)) usually yields sufficiently good results (see Figure 3.29). The kernel is applied such that the pixel whose value is queried is in the center of the 3×3 kernel. Each pixel in the kernel is then assigned a weight depending on its distance to the pixel at the center of the kernel. The summation of the values of each pixel multiplied by its weight is used as the ultimate displacement value of the pixel at the center of the kernel. It should be noted that the sum of weights assigned to the pixels in the kernel is equal to one so there is no overall amplification or deamplification to the values of the heightmap. The presented approach partly delegates the task of smoothing parts of the surface to the editor application responsible from the creation of the heightmaps. This is explained in more detail in Section 5.2.2.

$$G = \begin{bmatrix} 0.0625 & 0.1250 & 0.0625 \\ 0.1250 & 0.2500 & 0.1250 \\ 0.0625 & 0.1250 & 0.0625 \end{bmatrix} \tag{3.14}$$

The displacement operation is performed on each vertex of all surface patches of the terrain. However, due to the shared usage of vertices among surface patches, displacement may be applied multiple times on some vertices. This is not a desired consequence of the vertex displacement process. Therefore, the concept of vertex owners is introduced for externally shared vertices. The owner of an externally shared vertex is the surface patch that is pointed by the first node of the shared vertex list of that vertex. Displacement is applied on externally shared vertices only by the surface patch that owns the vertex. Consequently the chance of multiple application of displacement to the same externally shared vertex by different surface patches is eliminated.

The displacement operation can also occur multiple times on the internally shared vertices as multiple vertex indices can point to the same vertex if an edge of the surface patch is collapsed. In this case, the displacement must be applied to only the first vertex index that points to the internally shared vertex. All other vertex indices that point to an internally shared vertex is called inactive vertex indices. Algorithm 3.4 is used to determine whether a vertex index is inactive. If the vertex index is inactive, then the displacement is not applied to the vertex pointed by that vertex index as that would be a re-application of displacement to the same vertex. Algorithm 3.5 is used to apply displacement to the entire

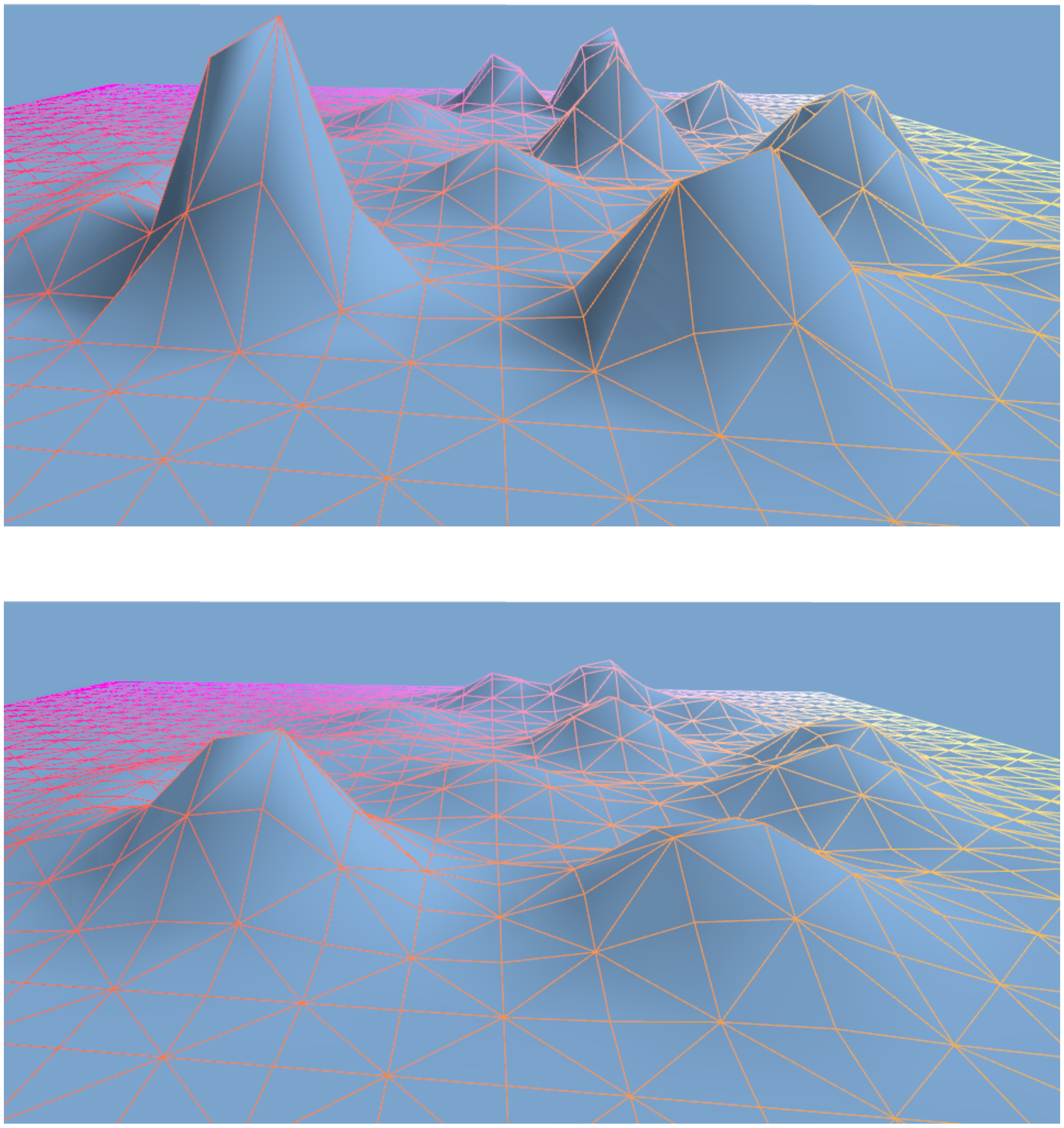


Figure 3.29: Top: the result of the displacement operation without Gaussian filtering, bottom: the same displacement map applied with Gaussian filtering (kernel size is 3 × 3). Note the extremely jagged edges when the displacement map is applied without Gaussian filtering.

terrain as explained in this section.

It should be noted that the computation of the surface normals must be performed after the application of displacement maps to each and every one of the surface patches. The vertex normals are computed using the displaced positions of the vertices as explained in Section 3.5.3.

3.5.5 Terrain Deformation

One of the goals of the proposed terrain representation is that it should be editable and deformable in real-time. Simple methods for terrain editing is explained in more detail in the next sections as part of a sample editor and visualizer application. How to deform the terrain, e.g., by erosion or by other objects in the world, is beyond the scope of this thesis. The proposed approach, however, tries to define a framework for making it possible to deform the terrain in real-time. The limitation of the proposed approach, however, is that the deformation of the terrain can only be performed at the heightmap-level in real-time. In other words, it is not possible to smoothly modify the voxel model in real-time as that would trigger the process of surface extraction process to rerun and cause dramatic changes to the terrain. It is possible to modify the heightmaps assigned to surface patches in real-time, though.

Editing heightmaps causes the displaced positions of the affected vertices to change. Recomputing the new displaced position is quite an easy task. The modification of the displaced position of a vertex, however, invalidates the previously computed face normals of all triangles of which the modified vertex is a part. It does not, unfortunately, stop here since the invalidation of a face normal of a triangle invalidates the surface normals of all vertices that constitute that triangle. This ripple effect causes a modification to the heightmap of one particular surface patch to spread to the vertices of the neighbor patches. See Figure 3.30 for an example of this ripple effect. If the displaced position of vertex V is modified, the face normals of the orange triangles are invalidated and must be recomputed. The invalidation of the face normals of these triangles invalidates the surface normals of the vertices A, B, C and D to be invalidated. Now the surface normals of these vertices have to be recomputed. Recomputing the surface normals of these vertices, however, requires access to the orange and green triangles as these

```
vertexIndex          : (input) 2D Vertex index whose activity is queried
vertexIndexActivity: (output) True if the vertex index is active, false
                       otherwise

begin
    switch collapsedEdge do
        case None
            return false
        case Edge1
            if vertexIndex.i == 0 and
            vertexIndex.j == numVerticesPerEdge − 1 then
                return true
            else
                return false
        case Edge2
            if vertexIndex.i == numVerticesPerEdge − 1 and
            vertexIndex.j == numVerticesPerEdge − 1 then
                return true
            else
                return false
        case Edge3
            if vertexIndex.i == numVerticesPerEdge − 1 and
            vertexIndex.j == 0 then
                return true
            else
                return false
        case Edge4
            if vertexIndex.i == 0 and vertexIndex.j == 0 then
                return true
            else
                return false
```

Algorithm 3.4: Finding out whether a vertex index is inactive and should not be displaced

```
octree            : (input) Octree that defines the terrain model
displacedPosition : (output) Displaced positions of each vertex

1  begin
2      $uvScale = 1/(numVerticesPerEdge - 1)$
3      foreach voxel in octree do
4          foreach patch in voxel do
5              for $i \leftarrow 0$ to numVerticesPerEdge do
6                  for $j \leftarrow 0$ to numVerticesPerEdge do
7                      $vertexIndex = \{i, j\}$
8                      if isVertexIndexActive(vertexIndex) == false then
9                          continue
10                     if isBorderVertex(vertexIndex) == true then
11                         if getVertexOwner(vertexIndex) ! = patch then
12                             continue
13                     $u = i \times uvScale$
14                     $v = 1 - j \times uvScale$
15                     $displacementValue =$
16                         patch.heightmap.getFilteredValue(u, v)
17                     vertex = patch.getVertexByIndex(vertexIndex);
18                     vertex.displacedPosition = vertex.originalPosition +
                       vertex.displacementNormal × displacementValue;
```

Algorithm 3.5: Displacement of terrain surface vertices

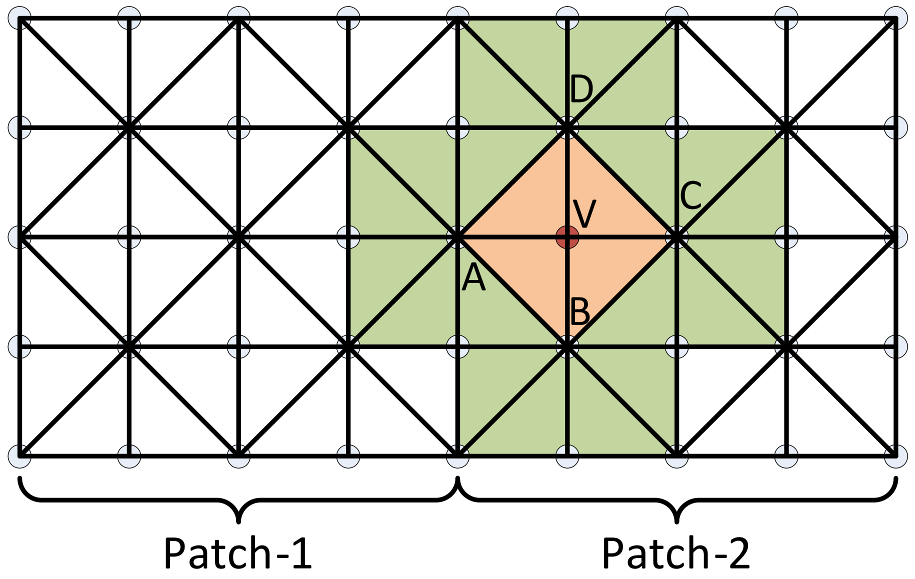


Figure 3.30: The ripple effect of modifying the displaced position of a vertex V causes the surface normals of the vertices A, B, C and D to be recomputed by accessing the face normals of the colored triangles.

vertices are among the ones that are part of these triangles. Even though the displaced position of an internal vertex of a surface patch is modified, it is now required to access the face normals of the triangles of the neighboring surface patches. The face normals are actually not stored in the proposed approach, only the vertex normals are stored, so the face normals of these colored triangles must now be recomputed. It should be noted that the vertex A is externally shared by the first and second patches and its owner patch is not clear, it can either be the first or the second surface patch. If its owner is actually the first patch, then not only is it required to access the triangles and vertices of a neighboring surface patch, but also to modify the attributes of its vertices.

The straightforward solution to this problem could simply be recomputing the surface normals of the entire terrain as it is described in Section 3.5.3. Recomputing the surface normals of the entire terrain is very slow, though, and it is dependent on the size of the terrain. Even for moderately large terrains it may not be possible to do this in real-time, and it is undoubtedly not plausible for larger terrains with millions of vertices. The granularity of surface normal recomputations is defined as surface patches by the proposed approach. This means that whenever the heightmap of a surface patch changes, the surface normals of all vertices of that surface patch are marked as invalidated and are required to be recomputed. The update of surface normals is performed in four steps:

1. Invalidate all vertices of the modified surface patch.
2. Invalidate all triangles of the modified surface patch and neighboring surface patches if at least one of the vertices of the triangle is invalidated.
3. Invalidate all vertices of all triangles that are invalidated.
4. Recompute the surface normals of the invalidated vertices.

This requires a 1-bit flag to be stored for each vertex and each triangle of the surface patches so that the invalidation status of vertices and triangles can be stored. Please note that the list of neighboring surface patches of a surface patch are not stored explicitly in the proposed representation. One way to find the neighboring surface patches is to scan the neighboring voxels in the octree and check if the bounding boxes of the surface patches of neighboring voxels overlap with the bounding box of the modified surface patch. Another yet faster method, though, is to scan the shared vertex list of the border vertices of the modified surface patch. Each node of each shared vertex list has a pointer to a patch that shares the vertex pointed by the shared vertex list. Thus, the set of surface patches that are pointed by the shared vertex lists of a surface patch is equivalent to the set of the actual surface patch and its neighboring surface patches. The proposed approach uses Algorithm 3.7 to update the displaced positions of the vertices of the modified patch, Algorithm 3.6 to find the neighboring surface patches of the modified surface patch and Algorithm 3.8 to update the surface normals of the vertices effected from the surface modification.

An overview of the terrain geometry generation process, which is a combination of surface extraction and surface generation phases, is depicted in Figure 3.31.

```
centerPatch         : (input) The surface patch whose set of neighboring
                      surface patches are queried
neighboringPatches: (output) The set of neighboring surface patches of
                      centerPatch. This set also contains the centerPatch
                      itself.

1 begin
2     foreach sharedVertexListNode in centerPatch do
3         firstNode = sharedVertexListNode.firstNode
4         while firstNode != NULL do
5             neighboringPatches = neighboringPatches ⋃
              firstNode.surfacePatch
6             firstNode = firstNode.next
```

Algorithm 3.6: Computing the set of neighbor patches of a given patch

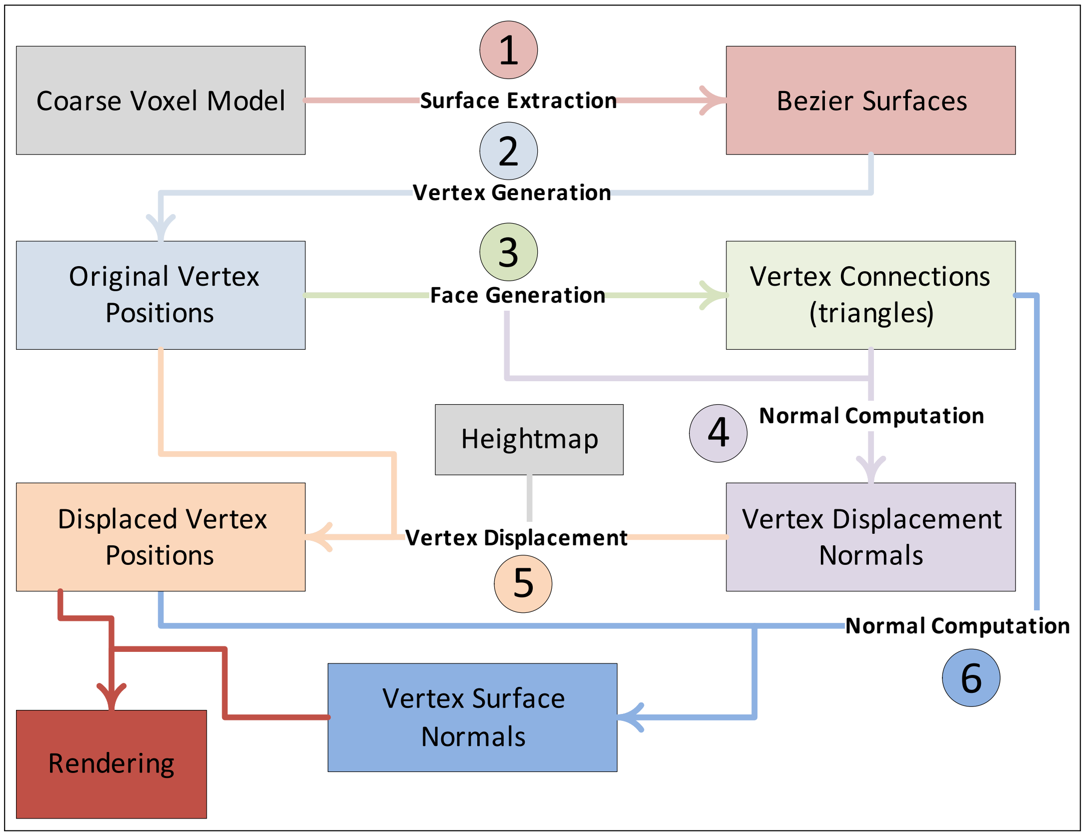


Figure 3.31: Overview of the terrain geometry generation process.

```
modifiedPatch     : (input) The surface patch whose heightmap is
                    modified
displacedPosition: (output) Displaced positions of each vertex
1  begin
2      $uvScale = 1/(numVerticesPerEdge - 1)$
3      foreach vertex in modifiedPatch do
4          $vertexIndex = vertex.vertexIndex$
5          if isVertexIndexActive(vertexIndex) == false then
6              continue
7          if isBorderVertex(vertexIndex) == true then
8              if getVertexOwner(vertexIndex) != patch then
9                  continue
10         $u = vertexIndex.i \times uvScale$
11         $v = 1 - vertexIndex.j \times uvScale$
12         $displacementValue = patch.heightmap.getFilteredValue$(u, v)
13         $vertex.displacedPosition = vertex.originalPosition +$
           $vertex.displacementNormal \times displacementValue$
14         // invalidate the surface normal of the vertex
15         $vertex.recomputeNormal = 1$
16         // reset the surface normal of the vertex
17         $vertex.actualNormal = \{0, 0, 0\}$
```

Algorithm 3.7: Updating the displaced positions of a surface *patch*

```
modifiedPatch: (input) The surface patch whose heightmap is modified
actualNormal : (output) Updated surface normals of the vertices effected
               by the deformation
begin
    neighboringPatches = getNeighboringPatches(modifiedPatch)
    foreach patch in neighboringPatches do
        foreach triangle in patch do
            foreach vertex in triangle do
                if vertex.recomputeNormal == 1 then
                    triangle.recomputeNormal = 1
    foreach patch in neighboringPatches do
        foreach triangle in patch do
            if triangle.recomputeNormal == 0 then
                continue
            foreach vertex in triangle do
                vertex.recomputeNormal = 1
    foreach patch in neighboringPatches do
        foreach triangle in patch do
            vertices = triangle.vertices
            if vertices[0].recomputeNormals == 0 and
            vertices[1].recomputeNormals == 0 and
            vertices[2].recomputeNormals == 0 then
                continue
            $\overrightarrow{v_1}$ = vertices[1] - vertices[0]
            $\overrightarrow{v_2}$ = vertices[2] - vertices[1]
            $\overrightarrow{N} = \overrightarrow{v_1} \times \overrightarrow{v_2}$
            for $i \leftarrow 0$ to 3 do
                if vertices[i].recomputeNormals == 1 then
                    vertices[i].actualNormal += $\overrightarrow{N}$
```

Algorithm 3.8: Updating the surface normals of the effected *vertices* after editing the heightmap of a surface *patch*

Chapter 4

Visualization

This chapter presents techniques used for visualization of the terrain surface whose generation is explained in detail in the previous chapter. Some of these techniques, such as the lighting and shadowing, aims to improve the visual quality and realism of the terrain rendering while others, such as level-of-detail and culling, aims to improve the performance of the terrain rendering, sometimes at the cost of a tolerable degradation in the visual quality.

4.1 Lighting

Lighting is crucial for all kinds of 3D rendering. Without proper lighting, the rendering is nothing more than a silhouette of the rendered object from the viewpoint of the observer (see Figure 4.1 to see the effect of lighting). Lighting can somewhat seem to be imitating shadows as well, when in fact it does not. Shadowing is explained in detail in the succeeding sections.

It is possible to use different types of lights when rendering a 3D scene. Some of these are point lights, spot lights, and directional lights. Point lights and spotlights are located at a specific position in the 3D space. These light types are usually used to simulate artificial lighting in the 3D scenes, such as the light emitted by a light bulb or a street lamp. Directional lights, on the other hand, are not located at a specific position. Directional lights are rather only defined

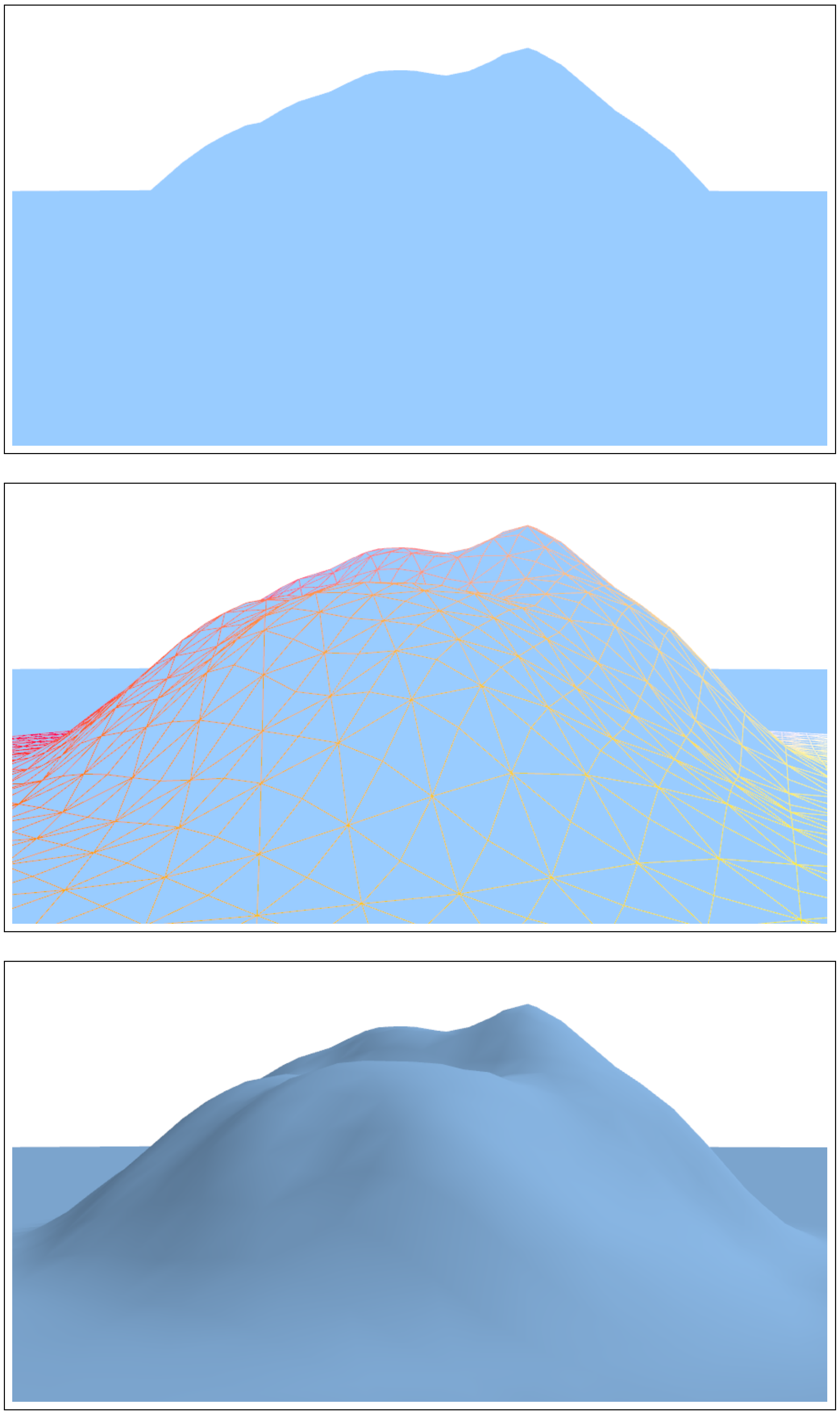


Figure 4.1: Top: rendering of a hill without lighting, middle: wireframe rendering of the hill geometry, bottom: rendering of the hill geometry with lighting.

by the direction of light that they emit. This type of light usually suits better to natural lighting for terrains, usually emitted by the Sun or the Moon. The light emitted by the Sun and the Moon can be very closely approximated by directional lighting on terrain models in spite of the fact that they both have specific positions in the universe. This is due to the fact that they are both very far away from the Earth and are very large compared to a terrain patch on Earth. This causes the light rays emitted by the Sun and reflected by the Moon that reach to a specific terrain patch on Earth to be almost parallel to each other. The proposed approach uses a simple lighting scheme where only directional lighting is used to simulate lighting of the Sun.

Typical matte materials reflect the most light when the surface is perpendicular to the direction of the incoming light rays. As the surface rotates and faces a different direction than the direction of the light rays the amount of reflected light reduces gradually. The amount of light reflected by objects determines how bright they are seen. Consequently, the most basic information required for lighting computation is the surface normals and the direction of the light rays, which is constant for directional lighting. In this case, the angle between the direction of the light rays and the direction of the surface normal at a particular point determines how bright a light is reflected by the terrain at that point. The dot product of vectors is commonly used in lighting computations as it yields the cosine of the angle between two vectors. The lighting factor s can be computed using the dot product by Equation (4.1) where $\overrightarrow{N_s}$ is the surface normal and $\overrightarrow{L_d}$ is the direction of the light.

$$s = \overrightarrow{N_s} \cdot (-\overrightarrow{L_d}) \tag{4.1}$$

There are two important things to note in lighting computations:

1. The inverse lighting direction is used in Equation (4.1) as the direction of the incoming light rays to the terrain surface point towards the surface while the surface normals point outwards of the terrain surface.

2. The lighting factor s computed by Equation (4.1) is in the interval $[-1, 1]$, the interval of the cosine function. It must be clamped to the interval $[0, 1]$ before it is used in further lighting computations. A lighting factor of one

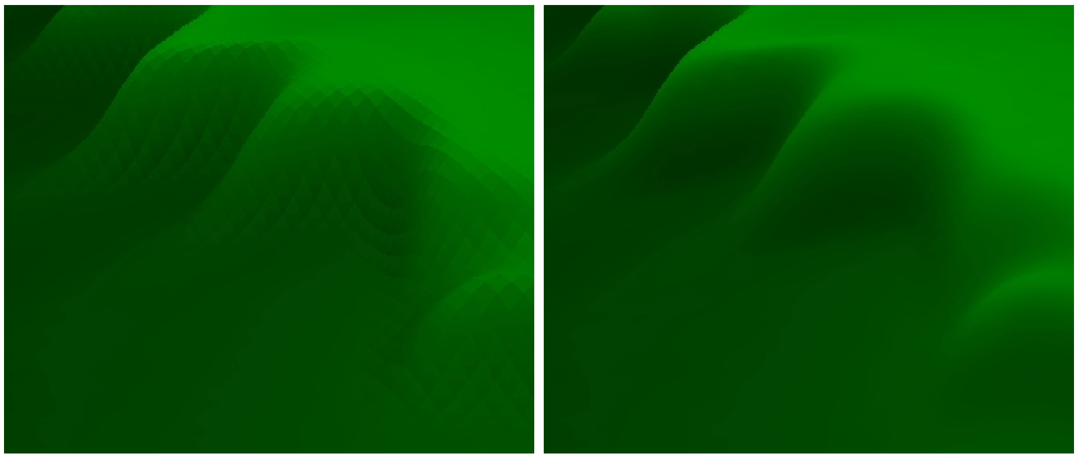


Figure 4.2: Left: lighting computed with face normals, right: lighting computed with vertex normals. Notice how the lighting computed with face normals is blocky and the boundaries of faces stand out.

means the surface reflects all the light that reaches to it and a lighting factor of zero means the surface does not reflect any light at all and is completely dark.

It is possible to use either the triangle normals or the vertex normals in lighting computations. The trade-off is that the computation of triangle normals is fairly easy as it is a local computation per triangle and is independent from the complexity of the geometry. The computation of the vertex normals, on the other hand, can be fairly cumbersome especially for arbitrary 3D models depending on the representation of the 3D model. If the face normals, i.e. triangle normals in the presented approach, is used for lighting computations then the lighting is said to be per-face. In this case, the surface normal is the same for every point on the face, hence the lighting factor is also the same for every point on the face. When per-face lighting is used, the lighting factor changes dramatically on the borders of faces as a result of the immediate change of surface normals. This results in a blocky rendering of the smooth surface where the borders of faces are inconveniently visible (see Figure 4.2). It is required to render too many faces in order to be able to achieve a smooth rendering with per-face lighting.

The proposed approach uses per-vertex lighting where the vertex normals are used for lighting computations. In this case, each vertex has the average normal value of all faces that are around it. The vertex normals are then interpolated for

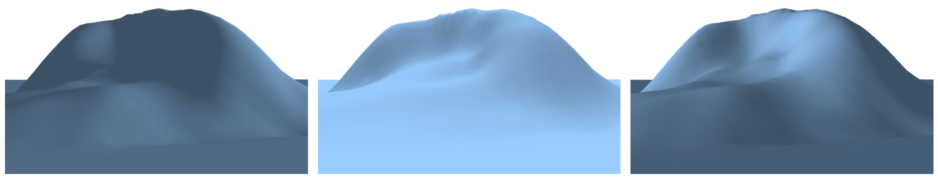


Figure 4.3: Left: light rays are coming from the east (sunrise), middle: light rays are coming from the above (midday), right: light rays are coming from the west (sunset).

the points in the interior of the faces. This results in a very smooth rendering of the surface and the smoothness of the lighting does not depend on the number of faces or vertices since the vertex normals are interpolated for each and every point on the surface. The computation of vertex normals for the proposed terrain representation is explained in detail in Section 3.5.3. It should be noted that only the surface normals, i.e. the actual normals, of the vertices are used in lighting computations, rather than the displacement normals. It is possible to change the direction of light based on the desired time-of-day of the rendering to obtain different lighting effects (see Figure 4.3).

Other types of lights such as point lights can be used especially inside closed spaces, e.g., caves, for artistic effects such as a torch on the wall of the cave (see Figure 4.4). In this case the light is located at a specific position in the 3D world, hence the direction of the light rays are not constant but depend on the position of the surface fragment. The lighting factor s for a simple point light can be computed by Equation (4.2) where $\overrightarrow{P_s}$ is the position of the surface fragment, $\overrightarrow{P_l}$ is the position of the point light, d is the distance of the light to the surface, and r is the maximum effective radius of the point light. More advanced terrain lighting is definitely possible. It is, however, a topic on its own and considered outside the scope of this thesis.

$$\begin{aligned}
\overrightarrow{L_d} &= \overrightarrow{P_s} - \overrightarrow{P_l} \\
d &= |\overrightarrow{L_d}| \\
s &= \left(\overrightarrow{N_s} \cdot \left((-\overrightarrow{L_d}) \times \frac{1}{d}\right)\right) \times \frac{max\{0, r-d\}}{r}
\end{aligned} \tag{4.2}$$

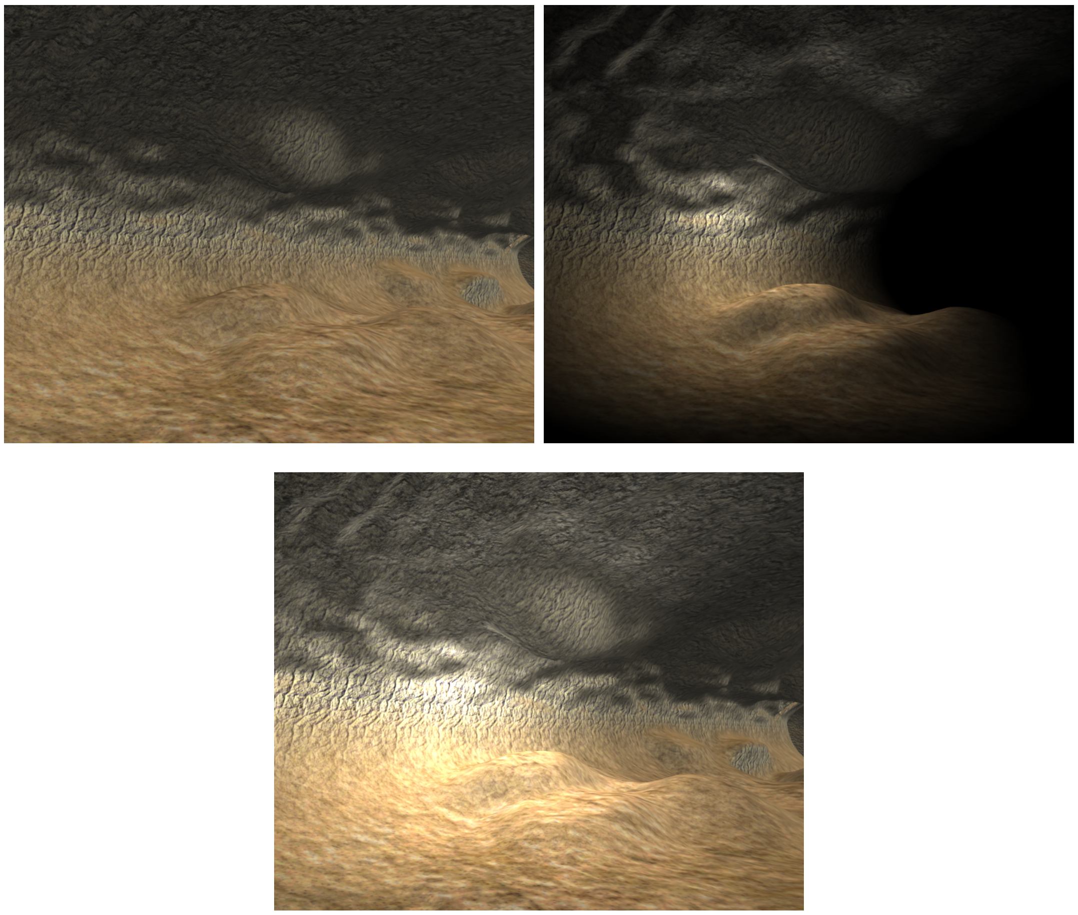


Figure 4.4: Top-left: cave rendering only with directional lighting, top-right: the effect of point light without directional lighting, bottom: the result of directional lighting is combined with point light.

4.2 Texture Mapping

Texture mapping is the method of wrapping up 3D objects with image data. Texture mapping is usually used for adding visual detail to 3D objects which otherwise would require a very high number of vertices. This extra detail provided by textures can be used for different visual attributes such as color data, lighting data, geometry variation data etc. This thesis focuses on surface texture mapping which is used to add color data to the terrain surface, such as a grass or dirt texture. Texture mapping tremendously helps in improving the visual detail and realism of virtual terrains and, thus, it is very important to provide ways to easily apply texture mapping to a terrain representation.

4.2.1 Generating Texture Coordinates

A texture map is applied to individual polygons, which are triangles in the presented approach. Each vertex must have 2D texture coordinates assigned to it for texture mapping to work. These texture coordinates actually map the 3D object to a 2D plane, which is actually the texture plane. The triangles of the surface must be mapped to a 2D plane such that the texture that is mapped to each triangle seamlessly blends at the edges of the triangle. Computing a seamless planar mapping for complex 3D objects is a very difficult task and it may not even be possible for sufficiently complex objects. There is another disadvantage to this technique. It requires extremely large texture image data for detailed objects of high resolution since each vertex corresponds to a unique texel (i.e., texture element, or texture pixel).

The proposed approach instead uses procedurally generated texture coordinates. In order to procedurally generate seamless texture coordinates for vertices the 3D world coordinates of the vertices must be mapped to 2D texture coordinates in some way. This is very easy for traditional heightmap-based terrain representations since the terrain grid is already planar in this case. Assuming that the terrain grid lies on the xz-plane the 2D texture coordinates (u, v) for a vertex at position $\overrightarrow{P}$ can be computed by Equation (4.3) where s is the texture mapping scale and $\overrightarrow{C}$ is the texture coordinate offset.

$$\begin{aligned} u &= P_x \times s + C_u \\ v &= P_z \times s + C_v \end{aligned} \tag{4.3}$$

This basically projects the entire 3D geometry to the 2D xz-plane in order to map the 3D vertex coordinates to 2D texture coordinates. For the proposed terrain representation approach, though, this does not suffice and produce artifacts. The proposed terrain representation allow very steep hills, and even cliffs that are perpendicular to the xz-plane. Assume two different vertices V_1 and V_2 such that V_1 is at coordinates $(x, 0, z)$ and V_2 is at coordinates $(x, 100, z)$. If xz planar mapping is used to assign 2D texture coordinates to V_1 and V_2, the same 2D texture coordinates will be assigned to both vertices as only the y-component of the vertex coordinates differ and the y-component of the vertex coordinate has no effect on the 2D texture coordinates generated (see Equation (4.3)). This causes an artifact because these vertices are actually far away from each other in the 3D world, nevertheless they have the same texture coordinates. This will cause the texture map to be skewed and look awkward (see Figure 4.5). This does not cause such a significant problem in traditional heightmap-based approaches where the vertex grid is planar because such representations do not allow two vertices at the same planar coordinates to have different heights. In other words both V_1 and V_2 cannot exist at the same time in such representations. This does not only happen for vertices with the same xz-coordinates, though. It is also evident in very steep hills and cliffs where the rate of change of the xz-coordinates is much less than that of the y-coordinate of the vertices. In such cases, again, two vertices that are geometrically very far away from each other are mapped very closely in the texture plane.

In order to overcome the issues of planar texture coordinate generation, the proposed approached uses another method called tri-planar texture coordinate generation. Tri-planar texture coordinate generation, in fact, is very similar to the planar texture coordinate generation. Tri-planar texture coordinate generation also projects geometry to a 2D plane to generate texture coordinates. Instead of projecting the geometry onto a single plane like in planar texture coordinate generation, however, tri-planar variation of the algorithm projects the geometry to 3 planes simultaneously: the xy-, yz-, and zx-planes. Consequently, three different 2D planar texture coordinates are generated, one for each plane. Each of these

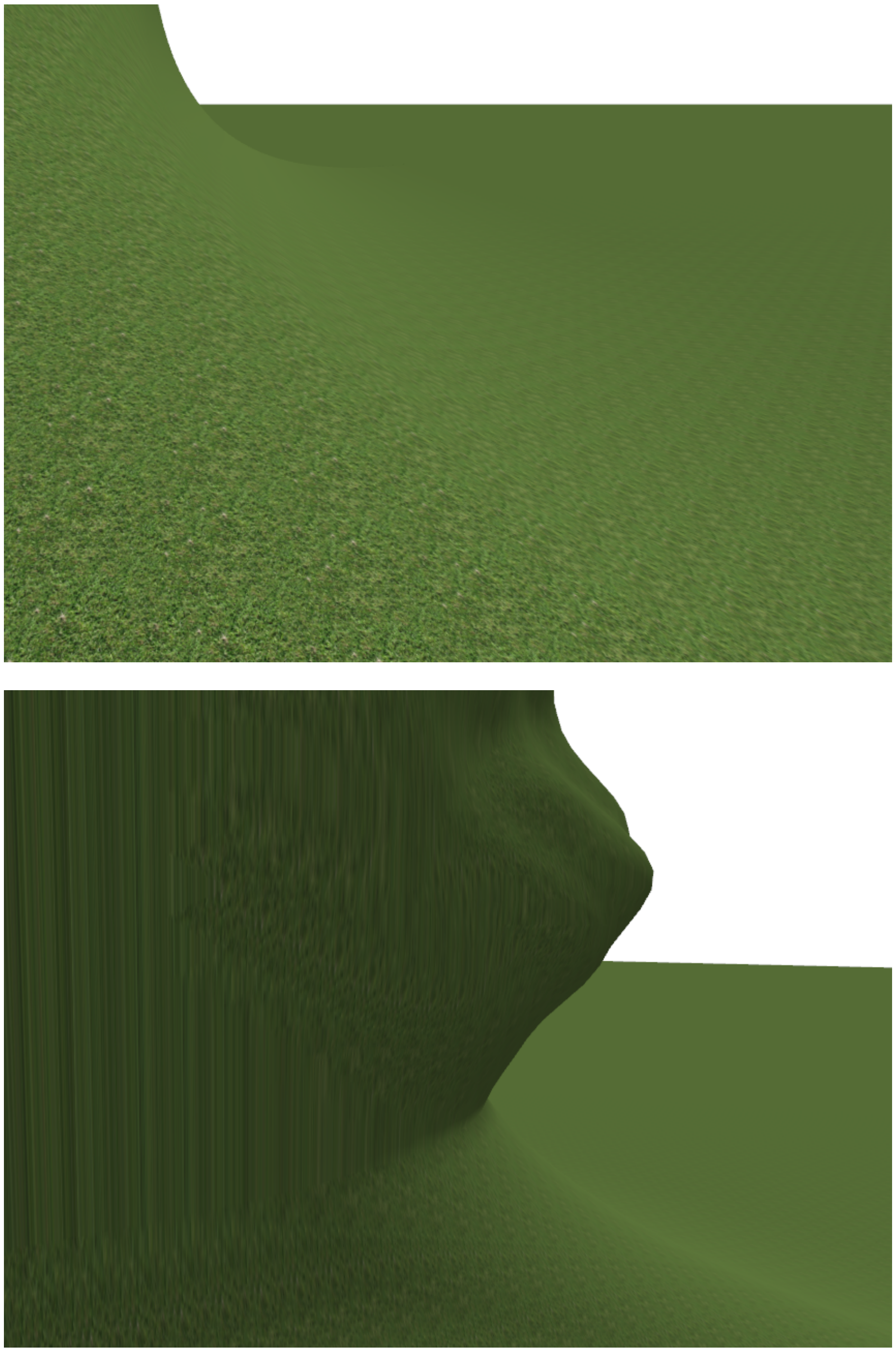


Figure 4.5: Top: planar texture coordinate generation yields sufficiently good results when the rate of change of the xz-coordinates is greater than the rate of change of the y-coordinate of vertices, bottom: for steep hills and vertical cliffs planar texture coordinate generation produces visual artifacts.

three texture coordinates are then assigned weights for blending. The assignment of the weights are performed according to the value of the vertex normal (i.e., the actual surface normal of the vertex, not the displacement normal) such that

- the weight of the xy-plane coordinate is proportional to the magnitude of the z-component of the vertex normal,
- the weight of the yz-plane coordinate is proportional to the magnitude of the x-component of the vertex normal, and
- the weight of the zx-plane coordinate is proportional to the magnitude of the y-component of the vertex normal.

Three texture coordinates obtained are used in three different texture lookups and three different texture values are retrieved. These values are then blended according to the weights of each of the texture coordinates that are used to retrieve them (see Figure 4.6). One disadvantage of this method over planar texture coordinate generation is that it requires three texture lookups rather than one.

With the use of this method, if the vertex normal is pointing upwards (i.e., $(0, 1, 0)$) then only the zx-plane coordinate is used, just like the planar texture coordinate generation. If the vertex normal is pointing sideways (e.g., $(-1, 0, 0)$) then only the yz-plane coordinate is used. If the vertex normal is in between these values then the color values obtained by zx- and yz-plane coordinates are blended accordingly. The artifacts caused by planar texture coordinate generation is, therefore, prevented in this way (see Figure 4.7).

The tri-planar texture coordinates of a vertex at position $\overrightarrow{P}$ with surface normal $\overrightarrow{N}$ can be computed by Equation (4.4) where s is the texture mapping scale, $\overrightarrow{R_{xy}}$, $\overrightarrow{R_{yz}}$ and $\overrightarrow{R_{zx}}$ are planar texture coordinates for the three planes, and $\overrightarrow{W}$ is the normalized weight vector. It should be noted that the weights assigned to the three texture coordinates must be normalized such that their sum is equal to one. Otherwise the texture data that is retrieved and blended is amplified and can cause visual artifacts (e.g., too dark or too bright colors).

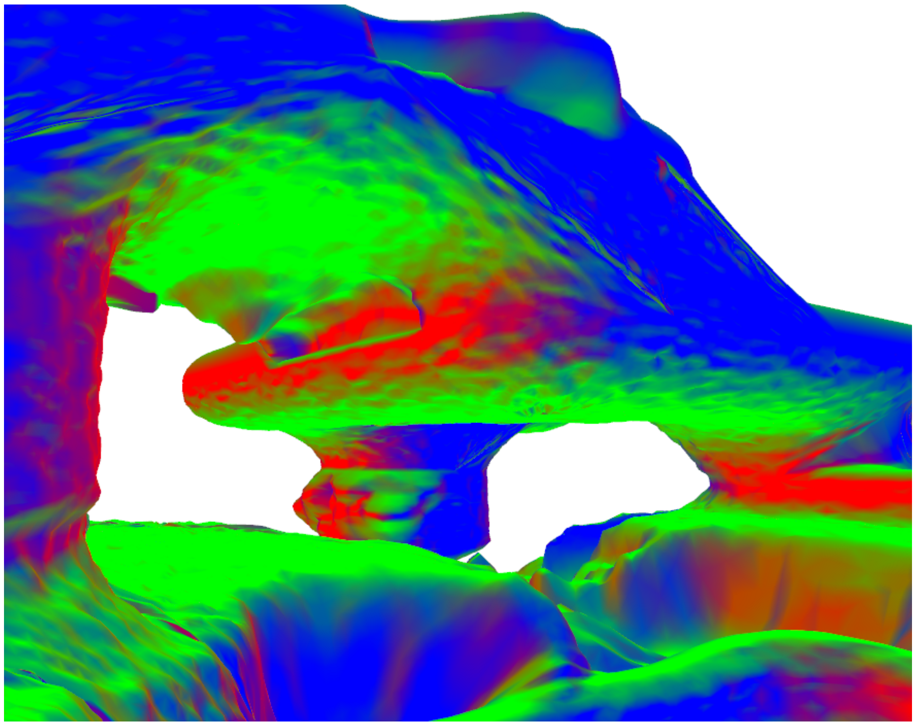


Figure 4.6: Color-coded rendering shows the usage of each of the three planes in the tri-planar texture coordinate generation, red: yz-, green: zx-, and blue: xy-planes. Notice how the texture values retrieved by each of tri-planar texture coordinates are blended in the areas where surface normals change.

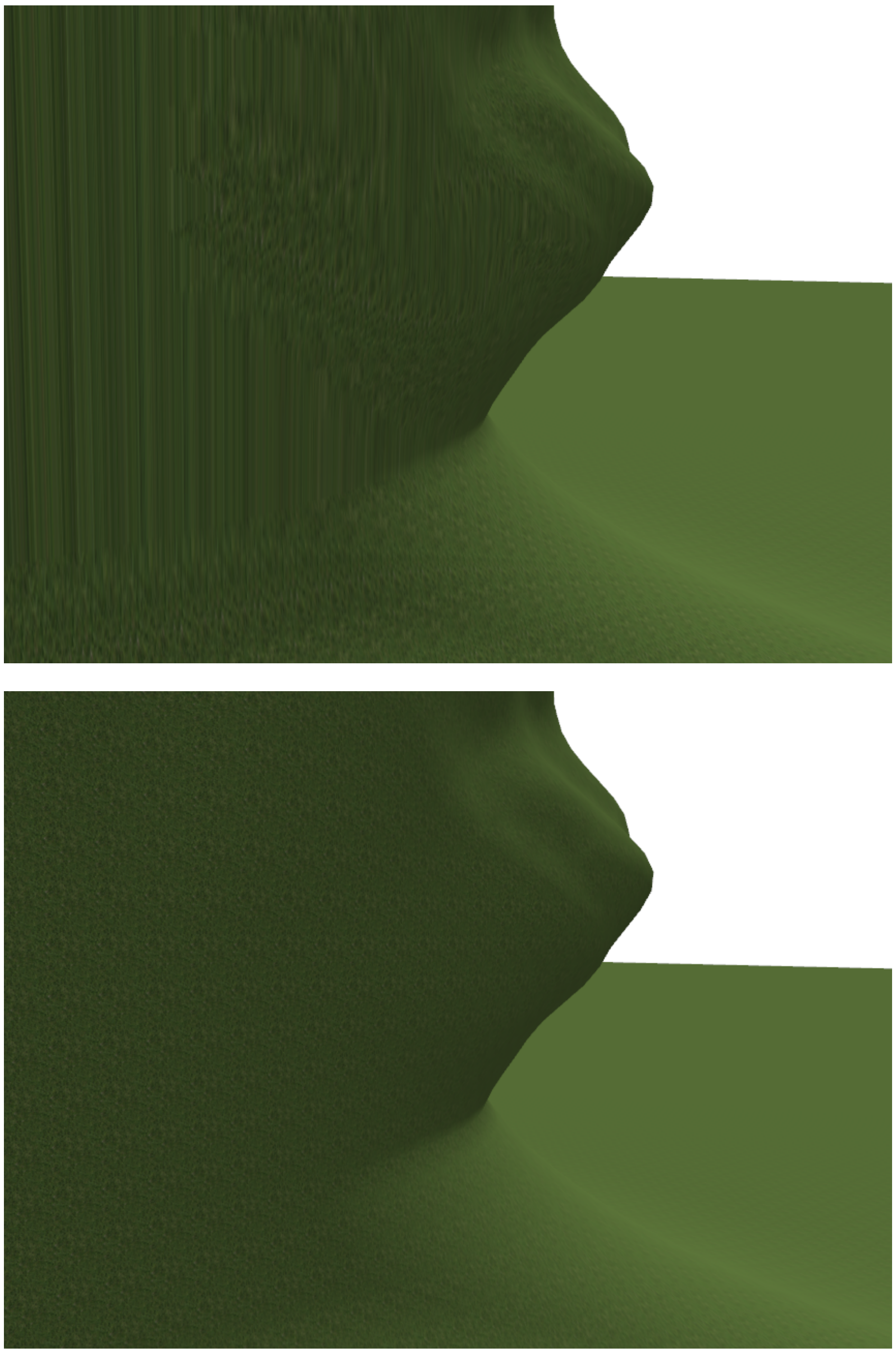


Figure 4.7: Top: the result of planar texture coordinate generation, bottom: the result of tri-planar texture coordinate generation.

$$
\begin{aligned}
\overrightarrow{R_{xy}} &= (P_x \times s, P_y \times s) \\
\overrightarrow{R_{yz}} &= (P_y \times s, P_z \times s) \\
\overrightarrow{R_{zx}} &= (P_z \times s, P_x \times s) \\
\overrightarrow{Z} &= (max\{0, |N_x| - \epsilon\}, max\{0, |N_y| - \epsilon\}, max\{0, |N_z| - \epsilon\}) \\
\overrightarrow{W} &= \overrightarrow{Z} \times \frac{1}{Z_x + Z_y + Z_z}
\end{aligned}
\tag{4.4}
$$

The scalar value ϵ in Equation (4.4) is just a factor to tighten up the width of the blending zone. If the blending zone is too large and different textures are used for different planes then the textures are blended almost everywhere on the terrain surface. This can produce undesired results. Increasing the value of ϵ decreases the area where the tri-planar texture coordinates are blended. We have observed in our experiments that the values in the range [0.15, 0.25] tend to produce good results. The ultimate texture value $\overrightarrow{T}$, i.e., the color vector in this case, can be computed by Equation (4.5) where the functions h_{xy}, h_{yz}, and h_{zx} are the texture lookup functions for the three planes. These functions can be equal if the texture mapping of the three planes are desired to be performed with a single texture (as in Figure 4.7)

$$
\begin{aligned}
\overrightarrow{C_1} &= h_{xy}(\overrightarrow{R_{xy}}) \times W_z \\
\overrightarrow{C_2} &= h_{yz}(\overrightarrow{R_{yz}}) \times W_x \\
\overrightarrow{C_3} &= h_{zx}(\overrightarrow{R_{zx}}) \times W_y \\
\overrightarrow{T} &= \overrightarrow{C_1} + \overrightarrow{C_2} + \overrightarrow{C_3}
\end{aligned}
\tag{4.5}
$$

4.2.2 Multi-texturing

The proposed approach to terrain visualization uses texture tiling as described in the previous section. Tiling frequency is an important decision in texture tiling. Higher frequencies of tiling gives good detail at close proximity and does not require the usage of very high resolution texture images for detail. The disadvantage of higher frequency tiling is that as the observer views the surface from further away the patterns start to show up inconveniently. These patterns show up because the same texture image is repeated many times over the large terrain surface. Lower frequencies of tiling, on the other hand, may help preventing the visibility of the repeating patterns on the surface when the surface is viewed from far away. However, close-up views of the terrain surface do not present enough detail unless the texture image is very high resolution.

To improve the visual quality without sacrificing performance and memory usage by increasing the resolution of the textures our terrain visualization approach uses multi-texturing, which is used commonly in terrain rendering anyway. In this method, multiple textures are used for texture mapping a surface where each texture is sampled at a different frequency. Usually one texture is used for close-up views and contains detail image data and another texture is used for far-away views and contains variations to prevent patterns emerging from the use of the first texture. It should be noted that the texture scale s in Equation (4.4) is different for each texture in the use of this method. See Figure 4.8 for a simple comparison of single-texture mapping and multi-texture mapping when the terrain surface is viewed from different distances. Notice how the patterns caused by single-texture mapping is evident when the terrain surface is viewed from far away.

A noise function can be used to introduce more variation to the terrain surface by applying another texture depending on the noise value. The presented approach uses Simplex noise for this purpose. Simplex noise is a modified and improved variant of the Perlin noise function [38, 39]. The noise function can be sampled multiple times to obtain a more realistic variation such that different octaves of noise are used in each layer. In Equation (4.6), for example, three octaves are used where in each higher octave the amplitude is halved and the frequency is doubled. The amplitude is slightly changed rather than exactly halved so that the emerging patterns are prevented.

$$n = h(x) + h(x \times 2) \times 0.487 + h(x \times 4) \times 0.231 \tag{4.6}$$

The computed noise value can then be used to smoothly blend and make a

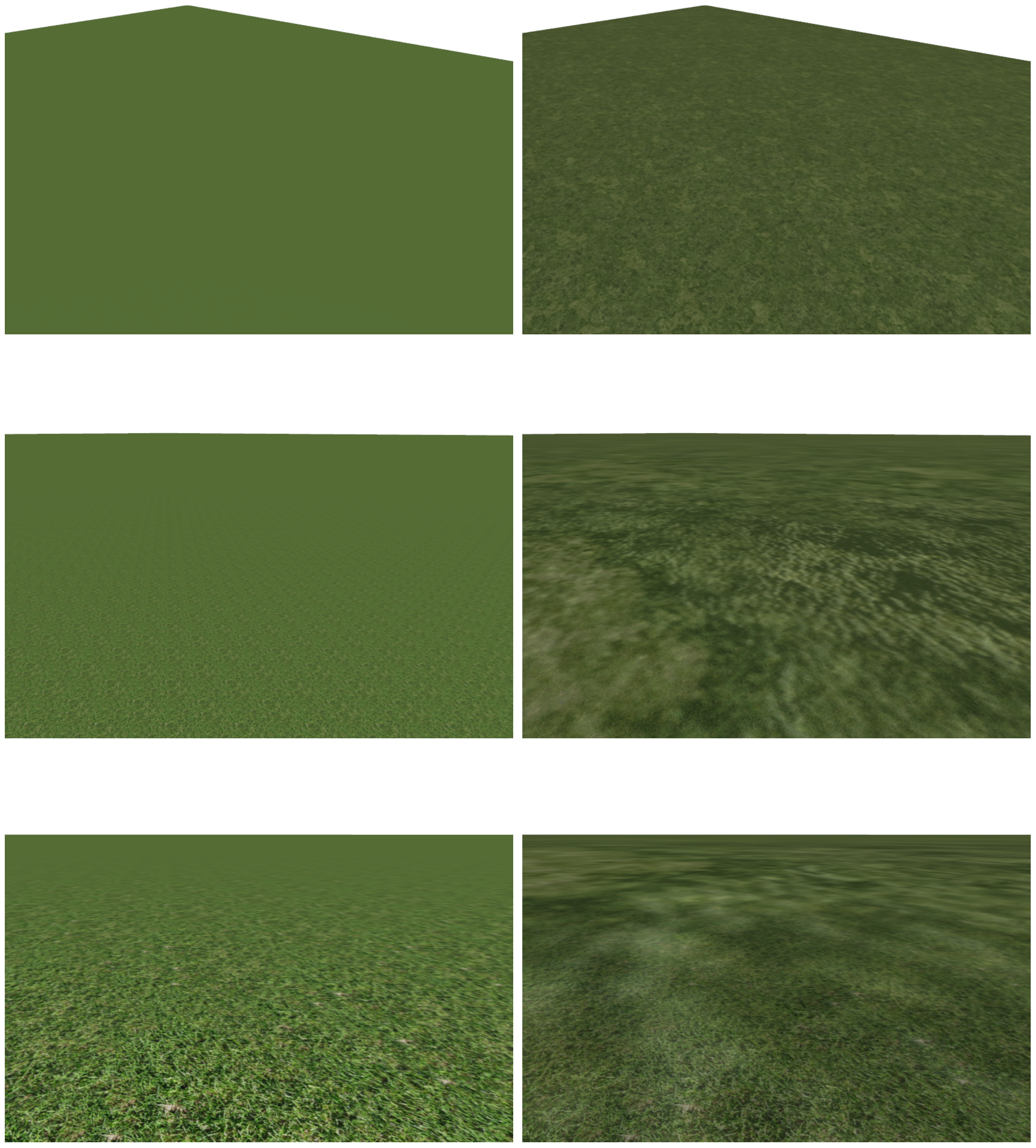


Figure 4.8: A comparison of single-texture mapping and multi-texture mapping. Top: the distance to the terrain surface is greatest, middle: the distance is moderate, bottom: close-up view of the terrain surface.

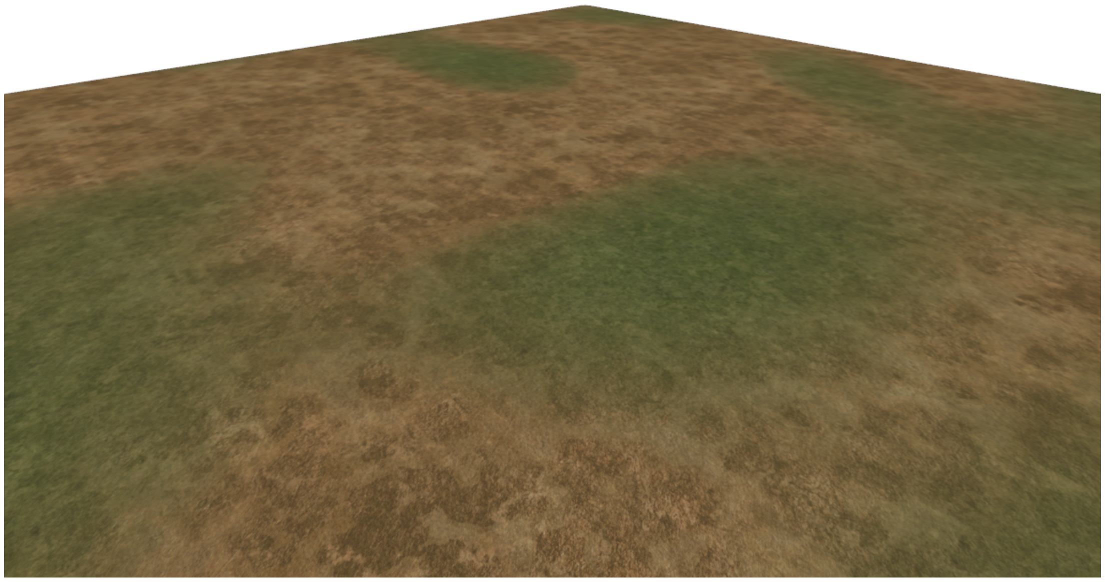


Figure 4.9: The use of multi-texturing where transitions are made between textures according to a noise value computed by the world coordinates of the vertices.

transition between different textures on a surface (see Figure 4.9). A 3D noise function is used in this case. The input to the noise function is computed from the world coordinate of the corresponding point on the surface.

4.2.3 Texture Splatting

Texture splatting is the application of multiple textures on top of each other and blending the color values according to an alpha channel. Since the terrain can be modified and deformed dynamically in our approach, storing a static alpha channel texture map would not be very useful. Instead the presented approach dynamically computes texture weights based on different vertex attributes. The two basic vertex attributes used for this purpose are the surface normal and the height of the vertex. See Figure 4.10 for a sample application where a grass texture is used in horizontal surfaces and a rock texture is used in vertical surfaces. The transition is made between textures according to the y-component of the surface normal. Another sample application could use vertex height such that surfaces at a higher altitude could use a snow texture while surfaces at a lower altitude could

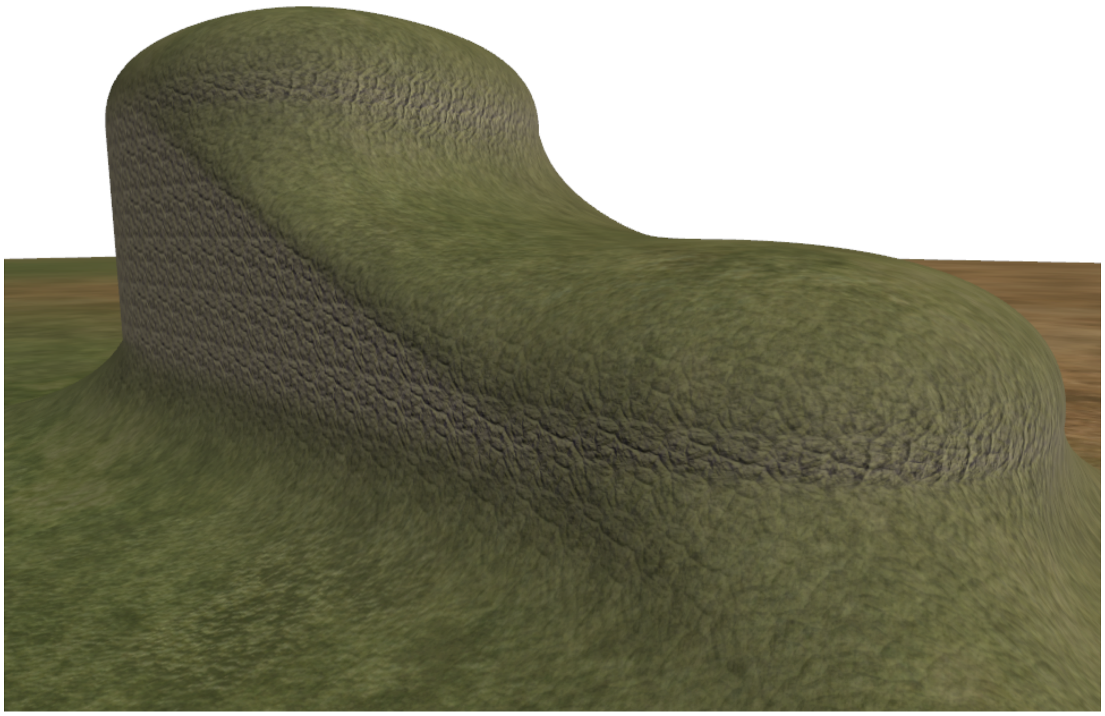


Figure 4.10: A sample usage of texture splatting where a grass texture is used for the xz-plane and a rock texture is used for the xy- and yz-planes.

use a grass texture. The implemented terrain renderer also uses another texture for cave ceilings. The cave ceiling texture is applied where the y-component of the surface normal is a negative value and the grass texture is applied where the y-component of the surface normal is a positive value. It is possible to add various creative rules in this texturing framework.

4.3 Shadows

Shadows are another crucial visual element for realistic 3D rendering. It is arguably even more important for the visualization of the proposed terrain representation since the proposed representation allows irregular geometry such as arches, caves and hanging cliffs. The terrain is a single surface and parts of the terrain can project shadows on other parts of the terrain. One surface patch can even project a shadow on itself depending on its geometry. This is called self-shadowing. Shadowing scheme for a terrain rendering, therefore, must allow

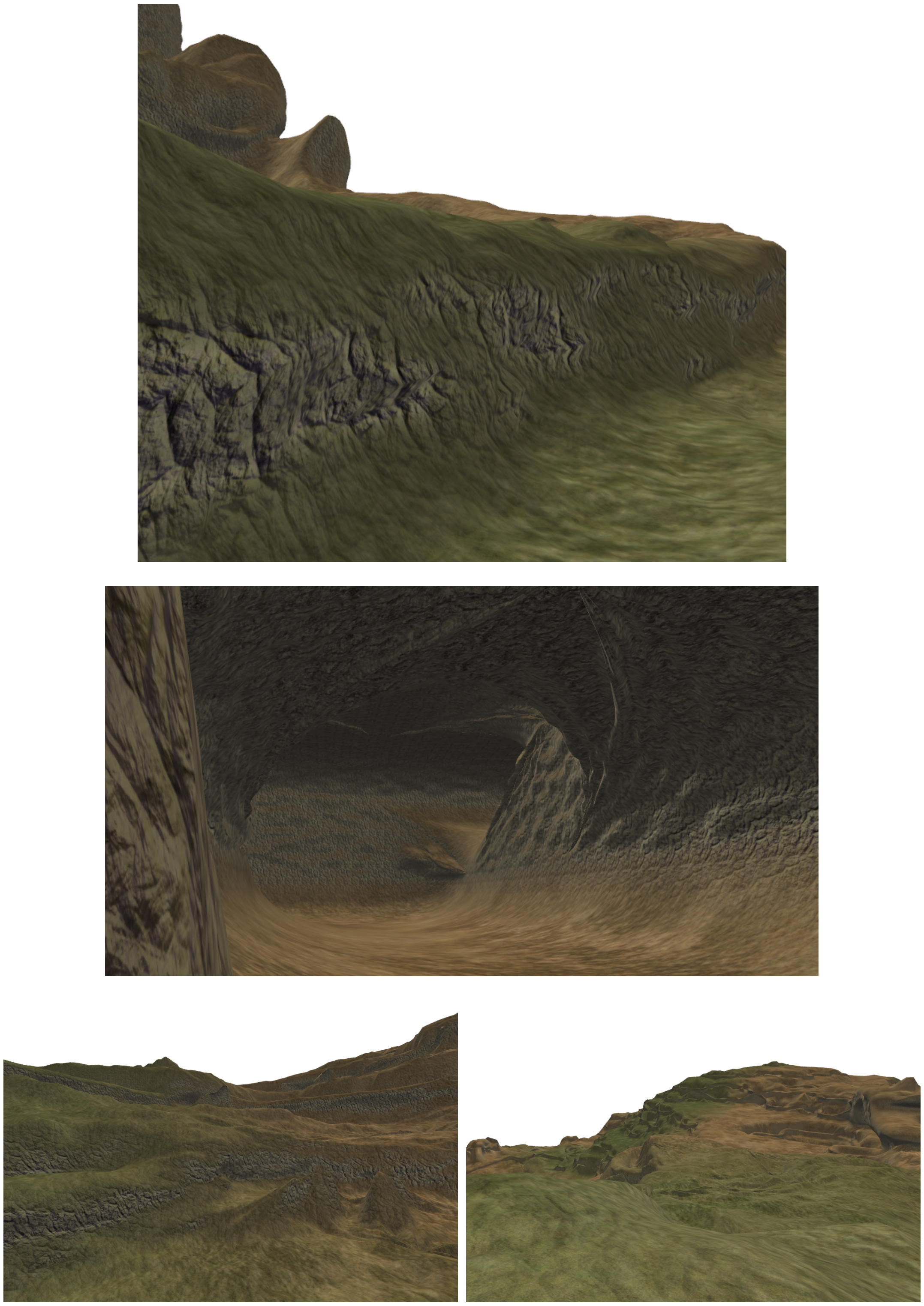


Figure 4.11: Presented texturing approach applied to a sample scene including cliffs and caves.

self-shadowing. This is one of the main reasons why shadow mapping is very popularly used in terrain rendering approaches. The presented terrain visualization approach uses a variation of shadow mapping as well.

4.3.1 Shadow Mapping

Shadow mapping is quite a simple technique, compared to some other shadow projection techniques, as it is a screen-space method and is mostly independent from the geometry of the scene, although the complexity of the geometry still has an effect on the visual quality of the shadows. To be able to apply shadowing to a scene, a method is required for telling whether a scene fragment is in shadow or not. If a scene fragment is visible by the light, such that there is no other fragment between the light and that fragment, then that fragment is said to be lit. Otherwise, the fragment is shadowed since the light rays is occluded by another fragment and does not reach the shadowed fragment.

Shadow mapping basically renders the 3D scene from the point of view of the light to produce a shadow map. This shadow map is actually a depth map. It does not contain color data as usual texture maps, instead it merely contains the depth values of the fragments that are visible by the light. These depth values represent the closest distance where each light ray is occluded by a scene fragment. When rendering the actual scene, the position of each fragment can be transformed into the viewpoint of the light and then the depth value of the fragment can be compared to the depth value that is retrieved from the shadow map. If the depth value of the fragment is greater than the one retrieved from the shadow map then the fragment is in shadow, otherwise it is lit.

Shadow mapping can be used with both point lights and directional lights. Using a perspective projection for rendering the shadow map simulates shadows projected by a point light, whereas using an orthogonal projection for rendering the shadow map simulates shadows projected by directional lighting. The presented visualization approach uses directional lighting as described in previous sections. Thus, orthogonal projection must be used for rendering the shadow map.

Computation of the frustum of light is very important when using shadow mapping. The frustum of light must contain the entire frustum of the observer and any other objects that may cast shadows in the frustum of the observer. That is, the frustum of light is actually a bounding box for the frustum of the observer. The near side of the frustum of light must then be extended to accommodate all objects that may cast shadows inside the frustum of the observer. We do this by basically using the bounding box of the entire terrain since any part of the terrain can cast shadows on the other parts.

The view matrix of the light M_{lv} is computed by Equation (4.7) where $\overrightarrow{s}$, $\overrightarrow{u}$ and $\overrightarrow{f}$ are the side, up and forward direction vectors of the direction of the light, respectively.

$$M_{lv} = \begin{bmatrix} s.x & u.x & -f.x & 0 \\ s.y & u.y & -f.y & 0 \\ s.z & u.z & -f.z & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{4.7}$$

The identity orthogonal projection matrix M_{op} is computed by Equation (4.8) where n is the near and f is the far distance.

$$M_{op} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{f-n} & 0 \\ 0 & 0 & \frac{-n}{f-n} & 1 \end{bmatrix} \tag{4.8}$$

The world coordinates of each corner point of the frustum of the observer is then projected to the light space by multiplying with M_{ls} given in Equation (4.9).

$$M_{ls} = M_{op} \times M_{lv} \tag{4.9}$$

Each corner point of the observer frustum is now transformed into the light space, that is, the observer frustum is transformed into the light space. The coordinates of the lower left (*min*) and upper right (*max*) corners of the bounding box of the view frustum can now be easily computed from the eight transformed corner points. Please note that this is an axis-aligned bounding box, hence two corner points are sufficient to define it. It is now possible to compute the ultimate projection matrix M_{lp} of the light by Equation (4.10) where the lower left (min) and upper right (max) corners of the observer frustum bounding box is used.

$$M_{lp} = \begin{bmatrix} \frac{2}{max_x - min_x} & 0 & 0 & -\frac{max_x + min_x}{max_x - min_x} \\ 0 & \frac{2}{max_y - min_y} & 0 & -\frac{max_y + min_y}{max_y - min_y} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{min_z - max_z} & 0 \\ 0 & 0 & \frac{-max_z}{min_z - max_z} & 1 \end{bmatrix} \tag{4.10}$$

The view-projection matrix of the light is then computed as $M_{lvp} = M_{lp} \times M_{lv}$. If the model matrix of the scene is not equal to the identity matrix then it must be multiplied with the view-projection matrix as well, in order to compute the model-view-projection matrix of the light. This matrix is used to transform each vertex while rendering to the shadow map. In the shadow map generation process, the z-component of each scene fragment is stored in the corresponding texel of the shadow map. It should be noted that writing the exact z-values of the fragments to the shadow map usually causes artifacts called depth-fighting (or z-fighting) where the fragments cast shadow on themselves on occasion as the depth of the fragment is very close to the depth stored in the shadow map. In order to prevent these artifacts a sufficiently large offset is added to the depth values of the fragments before they are stored in the shadow map. If the offset is too large, on the other hand, shadows cast by fragments close to each other may not be rendered correctly. We have observed in our experiments that an offset of $\epsilon = 0.0015$ yields visually good results when a shadow map with 24-bit depth values is used.

The generation of the shadow map is described so far. The second stage of rendering is to render the terrain from the point of view of the observer. This is the actual rendering stage and in this case it uses the shadow map created earlier to decide if shadow must be cast on a given fragment. In this stage, the vertices are transformed by the model-view matrix of the observer. That is, the coordinates of the vertices are in the observer space and must be transformed into the light space before the depth values can be compared. This transformation can be done by multiplying eye coordinates of each vertex in the observer space by the matrix M given in Equation (4.11) where M_{omv}^{-1} is the inverse of the model-view matrix of the observer.

$$M = M_b \times M_{lp} \times M_{lv} \times M_{omv}^{-1} \tag{4.11}$$

Please note that the matrix M_b is applied to the projected coordinates of vertices in the light space to transform the coordinates from the $[-1, 1]$ range to the $[0, 1]$ range as the texture coordinates and depth values of the shadow map are in the range $[0, 1]$. In order to do this, it basically multiplies the coordinates by 0.5 and then adds 0.5. The bias matrix M_b is given in Equation (4.12).

$$M_b = \begin{bmatrix} 0.5 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ 0.5 & 0.5 & 0.5 & 1 \end{bmatrix} \tag{4.12}$$

Projecting scene fragments in the observer space to a single corresponding texel in shadow map works in theory but causes aliasing on the edges of the shadow. These artifacts get even worse when the observer view is changing since the edges of the shadow appear to be sliding on the surface. In order get a better visual result we apply a Gaussian filter on the corresponding area of the shadow map. Instead of retrieving a single value from the shadow map, values of the neighboring texels are also retrieved and assigned weights according to their distance to the texel at the center. This method essentially blurs the edges of the shadow while having no effect whatsoever on the interior area of the shadows. The disadvantage of Gaussian filtering is that it requires N^2 shadow map texture fetches instead of a single one, where N is the width of the Gaussian kernel. As the number of texture fetches increase quadratically it is not possible to use much larger Gaussian kernels for rendering in real-time. We have observed in our experiments that a kernel size of 3×3 yields sufficiently good visual results (see Figure 4.12) and going beyond that size does not gain much in terms of visual quality while costing valuable performance.

Figure 4.13 shows the shadow map created for the scene in Figure fig:shadows when it is viewed at that specific position and viewing direction.

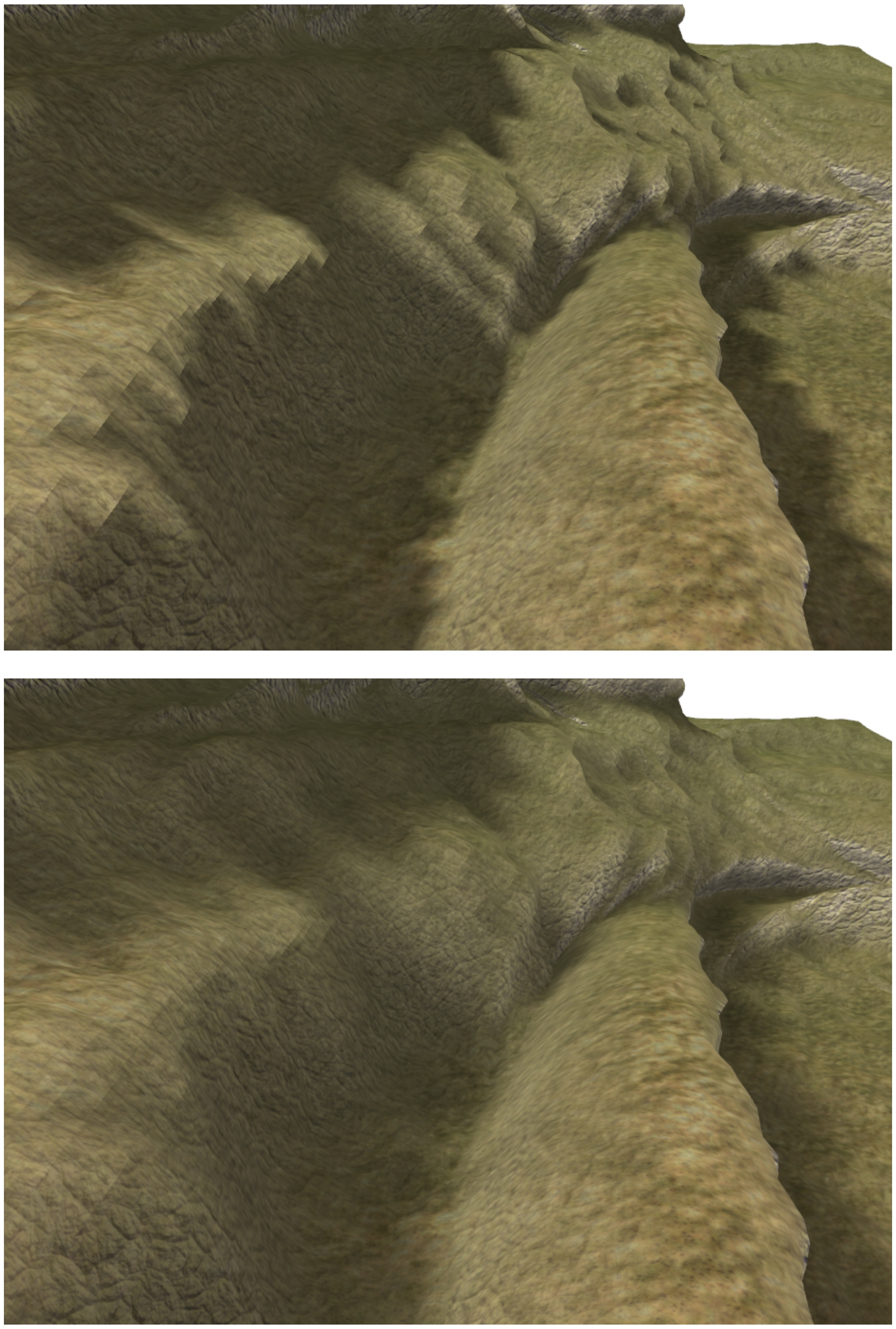


Figure 4.12: Top: shadow mapping without Gaussian filtering applied, bottom: shadow mapping with Gaussian filtering applied, kernel size is 3×3.

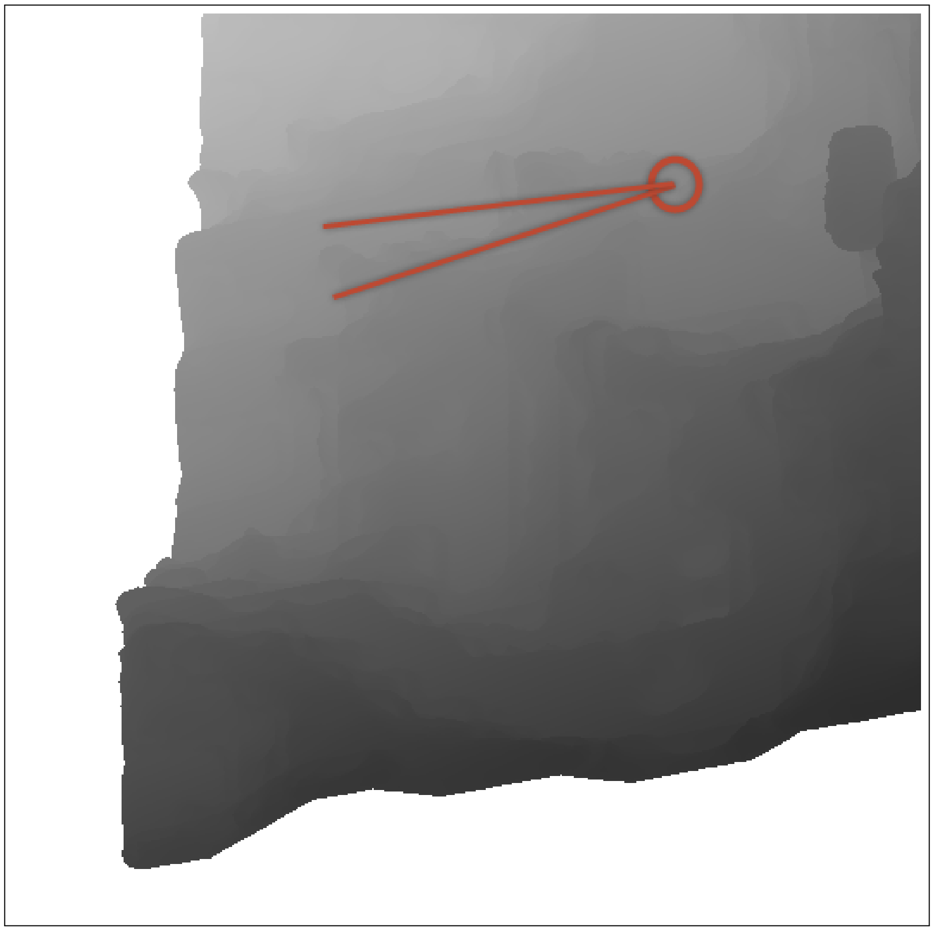


Figure 4.13: Shadow map generated for the scene in Figure 4.12. The position of the observer is shown with the red circle and approximate field of view of the observer is drawn with red lines.

4.3.2 Cascaded Shadow Maps

Using Gaussian-filtered sampling on shadow maps helps with aliasing but does not help much with the sliding edges of shadows when the observer is moving around the scene, or even changing the direction slightly. This is a chronicle problem of shadow mapping when used in large spaces like a large terrain as in our case. Shadow mapping provides about the same resolution for scene fragments regardless of their distance to the observer. In practice the fragments that are close to the observer occupy more screen-space compared to the fragments that are far away. As a result, ideally, we would like fragments that are closer to the observer to have a higher resolution in the shadow map. Fragments that are far away from the observer can have much lower resolution, without sacrificing visual quality. There is no way to increase the resolution of the closer fragments in the same shadow map without changing the shape of the frustum of light, which would change the type and direction of light unexpectedly. Capturing everything in a single shadow map requires impractically high resolution for the shadow map. Increasing the resolution of the shadow map is very costly because as the resolution increases

- the amount of valuable video memory used increases since the shadow maps are stored in the video memory (48 Megabytes of video memory is required to store a 24-bit 4096×4096 shadow map), and
- the shadow map needs to be generated from scratch for each frame rendered which costs millions of pixel fill operations to the graphics processing unit (e.g., 16 million pixel fill operations per frame for a shadow map of size 4096×4096).

Cascaded shadow mapping is using multiple shadow maps instead of a single one. The number of shadow maps usually varies between two to four depending on the size of the scene and the desired amount of detail. Each shadow map is used to cover a different range of the observer frustum:

- The shadow map that is used to store the depth values of the fragments that are close to the observer has the least amount of fragments. As a result, the resolution available for each fragment is higher.

- The shadow map that is used to store the depth values of the fragments that are far away to the observer, on the other hand, has the greatest amount of fragments. As a result, the resolution available for each fragment is lower.

Using cascaded shadow maps is very similar to using a single shadow map:

- First, N frustums are computed, one for each shadow map, such that the z-ranges of the each frustum cover a different part of the depth range of the observer.
- Generate N shadow maps, rendering to one at a time, using the previously computed frustums for each shadow map. It should be noted that this results in a different projection matrix for each shadow map.
- Render the actual scene using the N shadow maps. To determine whether a scene fragment is in shadow or not one of the N shadow maps must be used. Which shadow map to use is determined depending on the z-value of the fragment. If the z-value of the fragment is contained in the z-range of the first shadow map, then it is used. Otherwise the z-range of the second shadow map is checked against the z-value of the fragment and so on. Depending on which shadow map is used to store the depth value for the scene fragment, a different light matrix is used to transform the vertex coordinates into the shadow map texture space.

Our experiments have shown that using three or more shadow maps with a resolution of 2048×2048 yields visually very good results, not only for still renderings but also for animations where the observer is moving (see Figure 4.14). The cascaded shadow maps created for this scene are shown in Figure 4.15. It should be noted that the order of the shadow maps are important because the z-range of the most detailed shadow map is also contained by the z-range of the least detailed one. The algorithm, therefore, compares the z-value of each fragment to the first shadow map first and uses it if its z-range contains the z-value of the fragment. Otherwise, it checks the z-range of the second shadow map and so on.

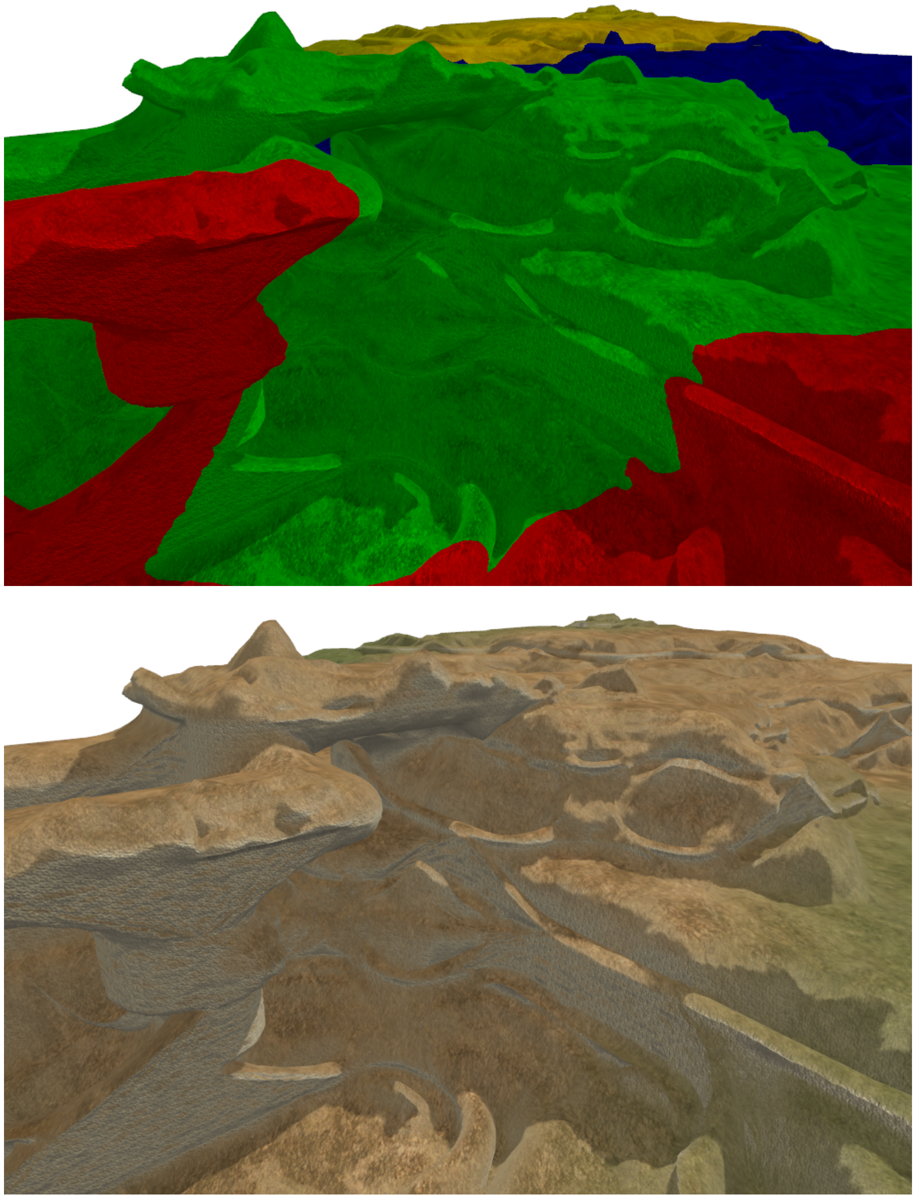


Figure 4.14: Top: colored areas show the z-range of each cascaded shadow map (four cascaded shadow maps are used, red shows the area covered by the most detailed shadow map and yellow shows the area covered by the least detailed one), bottom: the resulting rendering with cascaded shadow maps.

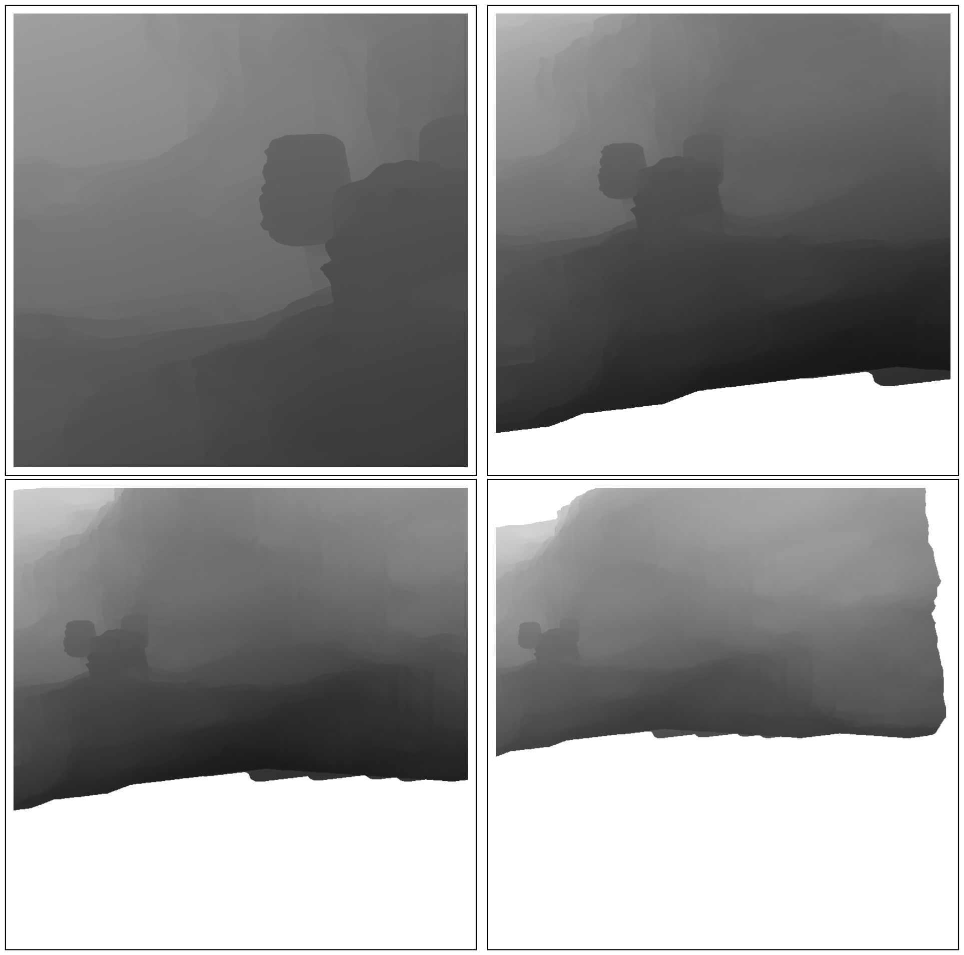


Figure 4.15: Four cascaded shadow maps created for the scene in Figure 4.14. Top-left: first shadow map, top-right: second shadow map, bottom-left: third shadow map, and bottom right: fourth shadow map.

4.4 Level of Detail

Real-time terrain rendering usually requires a level-of-detail management as well. Terrains are usually very large in dimensions and without a level-of-detail scheme the resolution must be high everywhere around the terrain as it is desired to provide a highly detailed geometry close to the observer. This can easily mean tens of millions of vertices for large and detailed terrains. Although modern graphics processing units are very powerful, and getting more powerful everyday, these numbers are still much more than what a real-time renderer can handle. What is more is that rendering so many vertices so densely everywhere around the terrain will cause aliasing artifacts on surface parts that are far away from the observer. This is similar to the artifacts caused by sampling large textures in small areas with high frequency, e.g., when texture mipmaps are not used. These artifacts are essentially caused by an unfiltered many-to-one mapping; in this case, many vertices are mapped to a single pixel because the further the surface patch is from the observer the smaller the polygons become and eventually size of a single polygon will become smaller than a pixel.

The solution, similar to cascaded shadow mapping described in the previous section, is to provide high resolution where it is needed, i.e., close to the observer, and lower resolution as it gets further away from the observer. This is, in fact, one of the main reasons that heightmap-based terrains are so popular for real-time rendering. Effective level-of-detail schemes can easily be implemented for heightmap-based terrains. We propose a level-of-detail management scheme that is very similar to simple level-of-detail schemes applied to heightmap-based terrains but with slight modifications so that it can be used with the proposed terrain representation.

The idea of level-of-detail for terrain rendering is easy. Its application to a terrain representation, however, is not trivial as it comes with its own set of problems:

- The primary aim of applying level-of-detail to terrains is to improve the rendering performance. Level-of-detail management algorithms run on almost every frame, especially when the observer is moving fast over the terrain. Even if the algorithm is not expected to run on every frame it still has to finish processing in the time interval of one frame as immediate drops in number of frames-per-second can be very inconvenient in real-time rendering applications. Thus, the algorithm must have minimal overhead and require the least amount of processing possible. Otherwise, it will slow down the rendering more than it speeds it up. This is especially difficult to balance in modern hardware as modern GPUs are usually much more powerful than CPUs and level-of-detail algorithms usually run on CPUs. A slow algorithm can cause the GPU to stall waiting for the CPU to feed data to operate on. Thus the level-of-detail algorithm has to run entirely on the GPU or must require very little processing such that it can be performed on the CPU without stalling the GPU.

- The most basic parameter of level-of-detail algorithms is the distance of the observer to the surface fragment. This can cause popping effects when the distance exceeds a thresholds and the geometry of the terrain is suddenly modified to switch to a higher- or lower-detail one. This is an undesired effect especially for terrains that is rendered in real-time where the movements of the observer cannot be predefined or controlled.

- It is difficult to maintain the smoothness and the continuity of the terrain surface as it is modified continuously and dynamically in real-time. Simple and straightforward approaches to level-of-detail can introduce visual artifacts such as holes and cracks on the terrain surface. Such artifacts are unacceptable in most rendering applications as they can have a serious negative impact on the visual quality and observer experience.

We propose a simple level-of-detail approach that tries to cope with each of these problems.

4.4.1 Basics of Level-of-Detail

Level-of-detail algorithms usually cannot work on vertex- or triangle-level due to performance considerations, as there are millions of these primitives in a typical terrain geometry. These algorithms, instead, work on higher-level geometry abstractions. The level-of-detail algorithm we propose work at the level of surface patches. There are usually hundreds to thousands of triangles in a single surface patch, hence there are quite a few orders-of-magnitude difference between

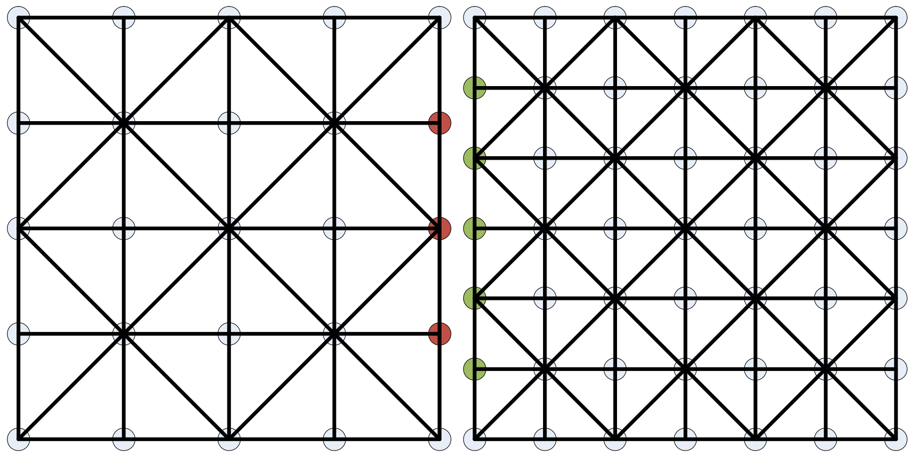


Figure 4.16: Left: a surface patch where $k = 2$, and right: a surface patch where $k = 3$. Note how the red and green colored border vertices do not match at all.

the number of surface patches and the number of triangles in a typical terrain geometry.

Each terrain patch is assigned a specific level-of-detail where level one is the least detailed and the detail increases as the level-of-detail index of a surface patch increases. The surface patches in the proposed approach have $2 \times k + 1$ vertices on each edge where k is a positive integer (cf. Section 3.5.2). This definition does not play well with different levels of detail (see Figure 4.16). Note that the border vertices of adjacent surface patches with different k values do not match. This means that most of the border vertices cannot be shared among surface patches anymore. This causes many more problems than it solves, e.g., the total number of unique vertices increases and as each unique border vertex can be at different locations it will be much more difficult to preserve the continuity of the surface where adjacent surface patches are assigned different level-of-detail indices.

The definition of the surface patches are consequently changed such that there are $2^k + 1$ vertices on a surface patch where k is a positive integer. This representation suits much better to our level-of-detail approach. In this case k is equal to the level-of-detail index of the surface patch. Neighboring surface patches that are assigned different level-of-detail indices now correctly share aligned border vertices (see Figure 4.17).

Level-of-detail indices of each surface patch can dynamically change at run-time. Computing the locations and normals of vertices of a surface patch each

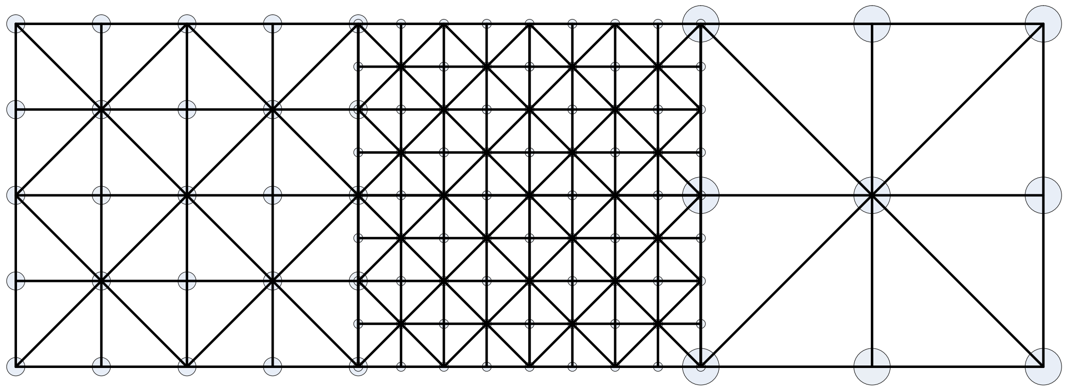


Figure 4.17: Left: a surface patch where $k = 2$, middle: a surface patch where $k = 3$, and right: a surface patch where $k = 1$. Note how some of the border vertices are aligned appropriately among neighbor surface patches.

time its level-of-detail index changes is too slow and impractical for a real-time application. Instead, a constant maximum level-of-detail is statically determined at the time of the creation of the terrain. All vertices at the highest level-of-detail are then computed and stored with each surface patch. Once a maximum level-of-detail is set, no surface patch can be rendered at a higher level-of-detail than the maximum. When it is desired to render a surface patch at a lower level-of-detail than the maximum, a subset of the vertices are selected accordingly (see Figure 4.18).

The vertex data itself does not suffice for rendering. The connection between vertices (i.e., triangle vertex indices) are required too. This data instructs the GPU about which vertices to connect in order to render each triangle. The

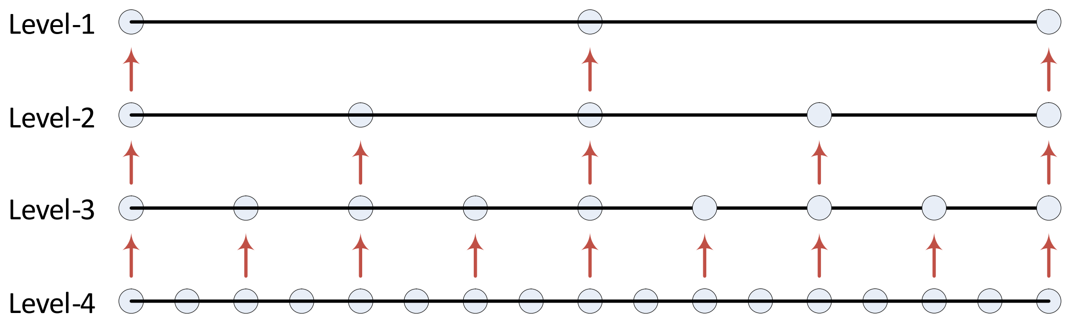


Figure 4.18: Side-view of the same terrain patch rendered at different levels. The maximum level-of-detail in this case is four. When a lower level-of-detail is to be rendered, a subset of the vertices are selected as shown here.

triangle vertex indices need to be computed per level-of-detail as they cannot be shared between different levels-of-detail. Although the connection pattern is the same for different levels-of-detail, the vertices connected by each triangle is different at each level. Thus, the triangle vertex index data must be computed separately for each level-of-detail. Even computing the triangle vertex indices is too slow for real-time processing, though.

Luckily, the triangle vertex indices never change once the surface patch is created from the voxel model. Even if the heightmap assigned to the surface patch is modified and the surface is deformed the connections between vertices are preserved. This data can, therefore, be computed as a preprocessing step during terrain surface generation.

The triangle vertex indices must be computed and stored separately for each level-of-detail. There are $2^{2\times k+1}$ triangles at level-k. This means that each higher level has four times more triangles than the lower level. Hence, storing index data for each level-of-detail, rather than just the maximum one, increases the memory required to store the index data by about 32%. This is an acceptable trade-off between memory usage and performance. With this approach, whenever we need to render a surface patch at an arbitrary level-of-detail all vertex and index data that is required will have been already computed and ready for use. All that is required then is to send the precomputed data to the GPU for rendering.

4.4.2 Level-of-Detail Selection

The most basic metric of selecting level-of-detail is the distance between the surface patch and the observer. It is a very simple yet effective metric for determining the correct level-of-detail. As the observer gets further away from the surface patch the length of each triangle edge gets shorter. The length of an edge is inversely proportional to the distance between the observer and the triangle. The area of each triangle is then approximately inversely proportional to the square of the distance between the observer and the triangle. Ideally, the areas of all triangles on the terrain surface should be equal such that the triangles that are further away are rendered larger. This is approximated in the presented level-of-detail approach. As the observer gets further away from a surface patch, the level-of-detail of the surface patch get lower and the number of triangles that

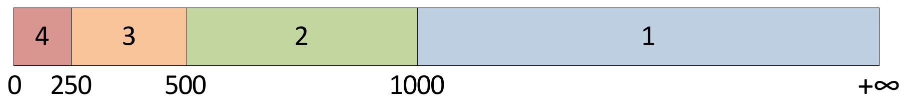


Figure 4.19: Sample level-of-detail ranges computed by Equation (4.13) where $k_{max} = 4$ and $d_{max} = 1000$.

cover the same surface area decreases and, consequently, the area covered by a single triangle increases. There are four times less triangles in each lower level-of-detail, meaning that the area of triangles in each lower level-of-detail is four times larger. If the distance between the observer and a surface patch is doubled, then the area of each triangle on that surface patch decreases to $\frac{1}{4}$ of its original, and if the level-of-detail of the surface patch is lowered, then the area stays approximately the same since $\frac{1}{4} \times 4 = 1$. Thus, the relation between the level-of-detail index k of a surface patch and its distance d from the observer is given by Equation (4.13) where d_{max} is the maximum distance beyond which the use of minimum level-of-detail is desired. It should be noted that the selected level-of-detail, k, decreases linearly as the distance d increases by powers-of-two (i.e., doubled), hence the use of logarithm in the relation between k and d. The value k computed by this equation must then be clamped to the interval $[1, k_{max}]$ where k_{max} is the maximum possible level-of-detail.

$$
\begin{aligned}
d_n &= \frac{d}{d_{max}} \\
k &= 1 - \log_2(d_n)
\end{aligned}
\tag{4.13}
$$

See Figure 4.19 for sample level-of-detail ranges computed by Equation (4.13) where $k_{max} = 4$ and $d_{max} = 1000$ values are chosen. In practice, k_{max} must be chosen considering the complexity of each surface patch and the targeted maximum detail for each surface patch, and d_{max} must be chosen depending on the far view range of the perspective camera that is used to render the scene.

Ideally, the distance d between the observer and a surface patch must be computed as the average of the distances between the observer and each triangle on the surface patch. This computation, however, requires the algorithm to go down to the triangle level which requires too much processing. The proposed approach, instead, approximates the actual distance by computing the distance

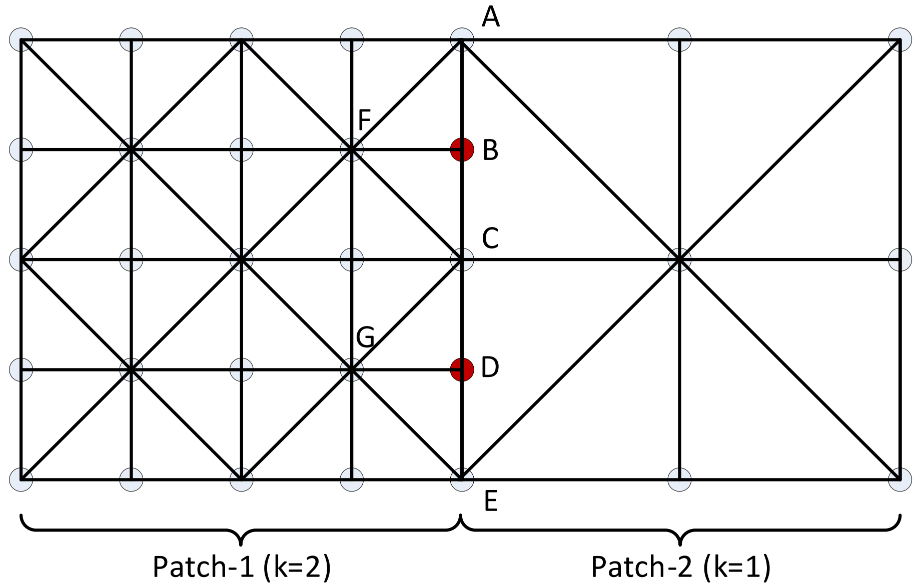


Figure 4.20: Adjacent surface patches are rendered at different levels-of-detail. Visual artifacts can occur as the red vertices are rendered on the border of surface patch-1 and not on surface patch-2.

between the observer and the bounding box of the surface patch. The center of the axis aligned bounding box volume is used as the reference location for all triangles on the surface patch. Please note that it is not the bounding box of the control points of the surface that is used in this computation, but rather the bounding box of the vertices that approximate the surface. If the vertices are not displaced (i.e., no heightmap is applied) then both bounding boxes are equal. They will differ in case the vertices are displaced, though, and the right one to use, in this case, is the bounding box of the actual vertices on the surface since, essentially, this is the bounding box of the surface patch triangles.

4.4.3 Level-of-Detail Artifacts

Rendering adjacent surface patches at different levels-of-detail can cause visual artifacts at places where the rendered border vertices of each patch do not exactly align. Figure 4.20 illustrates this situation. The vertices B and D, in this case, are rendered for surface patch-1 at level-of-detail index $k = 2$. These vertices, however, do not exist at level-of-detail index $k = 1$ and, hence, they are not rendered as a part of surface patch-2. The edges AB and BC are rendered for

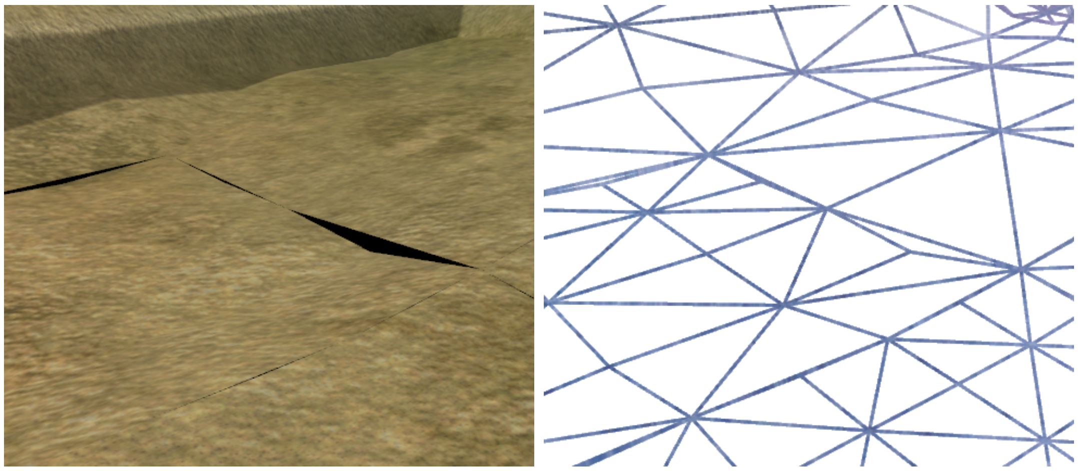


Figure 4.21: A sample visual artifact, a crack, caused by rendering adjacent surface patches at different levels-of-detail. Left: the actual rendering of the surface geometry, and right: the wireframe rendering of the same surface geometry.

surface patch-1 whereas the edge AC is rendered for surface patch-2, ignoring the vertex B in the middle due to the level-of-detail index of the surface patch. This can cause two similar but different problems:

- Visual artifacts called cracks can occur unless B is located on the edge AC (see Figure 4.21). This situation can easily occur in the proposed approach as each vertex can be independently displaced by the heightmap assigned to surface patches.

- Even if B is positioned on the edge AC, visual artifacts called T-joints may still occur. These visual artifacts are less disturbing than cracks, as they usually show up as a thin black line (e.g., like a tear on the surface) rather than a large hole on terrain surface. T-joint visual artifacts can occur because of floating-point rounding errors. In spite of the fact that the vertex B is on the edge AC it may still be rendered in a slightly different position due to floating-point rounding errors introduced during vertex transformations. Another possible cause of T-joints is the visual effects applied at the vertex level, e.g., per-vertex lighting computations. The attributes used for the per-vertex effect may be interpolated differently for the AB and BC edges compared to the AC edge. The surface normal for vertex B, for instance, may not be equal to the interpolated surface normal of the fragment at the location of vertex B on the edge AC.

The level-of-detail approach used by the proposed terrain representation constrains the levels of adjacent surface patches such that they cannot differ by more than one levels. That is, the difference between the levels of adjacent surface patches can be at most one. This constraint does not have any disadvantages in practice as the minimum distance between level-of-detail boundaries is usually much greater than the size of a single surface patch in which case it is already not possible for the levels of two adjacent surface patches to differ by more than one level. This minimum distance between level-of-detail boundaries $d_{min\_lod\_dist}$ can be computed by Equation (4.14). It is also not desired that adjacent surface patches can differ in levels by more than one since this would cause a sudden decrease in the level-of-detail perceived by the observer.

$$d_{min\_lod\_dist} = \frac{2^2 \times d_{max}}{2^{k_{max}}},\ \forall\ k_{max} > 1 \tag{4.14}$$

When a surface patch is adjacent to a lower-level one it is required to reorganize the triangles at the borders of these surface patches to get rid of the mentioned artifacts. The presented level-of-detail approach render the lower-level surface patch as it is and adapts the higher-level surface patch to the lower-level one at the common edge such that the artifacts are prevented. This is achieved by combining adjacent border triangles in the higher-level surface patch where a border triangle is defined as a triangle whose two vertices are border vertices. Figure 4.22 illustrates this, where two adjacent surface patches are rendered at different levels. This result is achieved simply by getting rid of the FB edge by combining the AFB and BFC triangles as they are shown in Figure 4.20. Instead of these two triangles, only the AFC triangle is rendered in this case. The same procedure is also applied for the CGE triangle. Please note that the previously mentioned artifacts caused by different levels-of-detail are prevented in this case since all A, C and E vertices that are used by both surface patches exist at the corresponding level of each surface patch. Therefore, no cracks or T-joints can form in this way.

The presented solution to the visual artifacts is not difficult but the application of this solution in real-time is still slightly more involved as it requires operating on the triangle level, which is a performance killer. The solution to the performance problems, once again, is to precompute these triangle vertex indices for both cases where

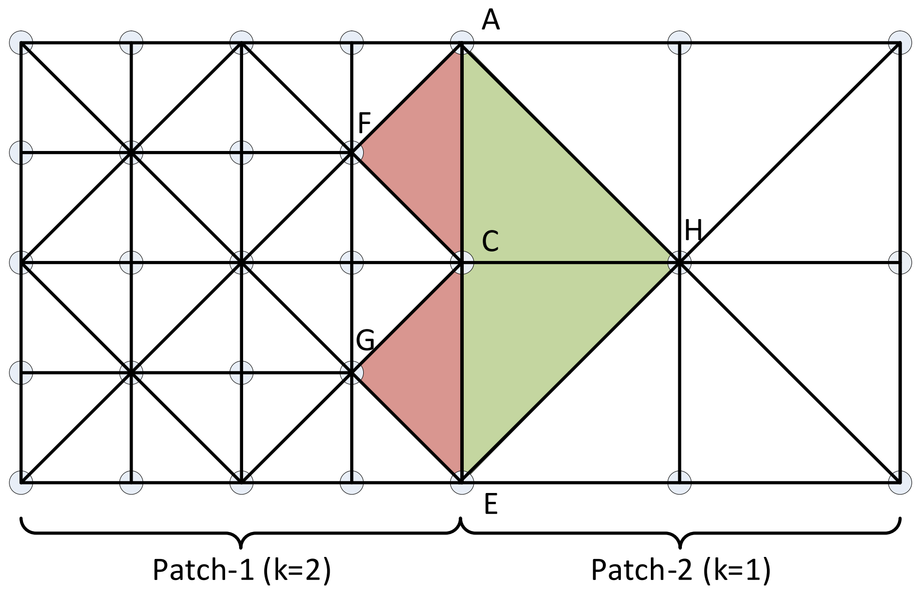


Figure 4.22: The rendering of two adjacent surface patches at different levels in order to prevent the artifacts caused by the level-of-detail difference.

- the adjacent surface patch is at the same level, and
- the adjacent surface patch is at the lower-level

It should be noted that one of these two cases is true for each edge of the surface patches independently. A surface patch may be adjacent to another surface patch at the same level at an edge and adjacent to a surface patch at the lower-level at another edge. Precomputing and storing the vertex indices for each of these $2^4 = 16$ possible cases increases the memory usage 16-times and it is quite impractical. Instead, the proposed approach divides the surface patch into five different regions and precomputes the vertex indices for each region independently:

- Four regions for each of the border vertices that belong to the four edges where a triangle is a border triangle if two of its vertices is located on an edge.
- One interior region where all but the border triangles are located.

The vertex indices of the interior region is static and these triangles are always rendered as they are regardless of the levels of the surrounding surface patches

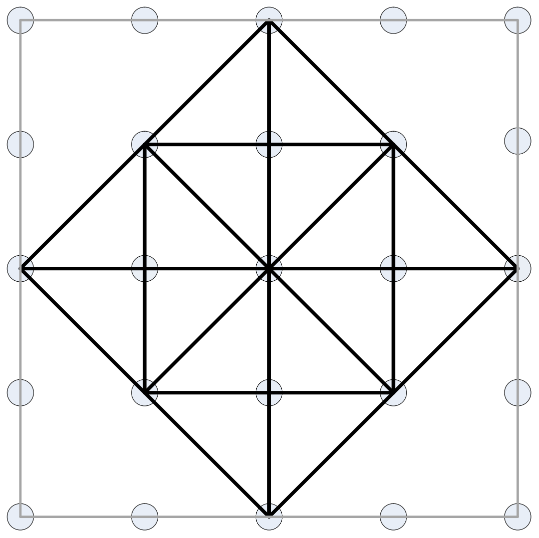


Figure 4.23: The triangles in the interior region of the surface patch.

(see Figure 4.23). The reason for this is the fact that all border vertices that are used by the triangles in the interior region exist at the lower level-of-detail as well. The rest of the surface patch consist of the border triangles which is a slightly more complex case. Each border triangle is associated with exactly one of the edges on which two of the vertices of the triangle is located (see Figure 4.24). The red vertices in this figure do not exist on the neighboring surface patches as they are at the lower-level, and we respond to this situation by getting rid of the edges that use these vertices. There are two vertex index batches for border triangles of each edge, one for the case where the adjacent surface patch is at the same level and the other for the case where the adjacent surface patch is at the lower-level. For the bottom edge, for instance, if the adjacent surface patch is at the same level as this surface patch the vertex index batch that contains E, F, G and H triangles is used. If the adjacent surface patch is at the lower-level, then another vertex index batch that contains T and U triangles is used.

Consequently, there are a total of nine vertex index batches that are precomputed and stored:

- One vertex index batch for interior triangles, and
- two vertex index batches for the border triangles of each of the four edges.

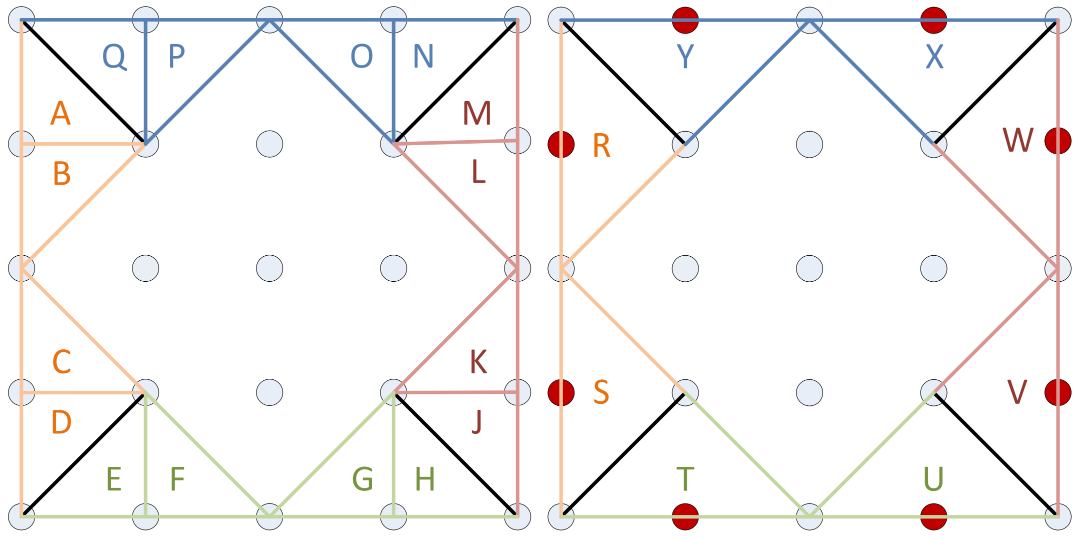


Figure 4.24: Left: the arrangement of border triangles where the level of each adjacent surface patch is equal to the level of this surface patch, and right: the arrangement of border triangles where the level of each adjacent surface patch is lower than the the level of this surface patch.

When a surface patch is to be rendered at a specific level, the interior triangles are always rendered as they are at that level-of-detail and the border triangles are selected depending on the level-of-detail of each surface patch that is adjacent to each of the four edges of the surface patch (see Figure 4.25). This is a very fast operation as all of these vertex index batches are precomputed and only need to be sent to the GPU for rendering.

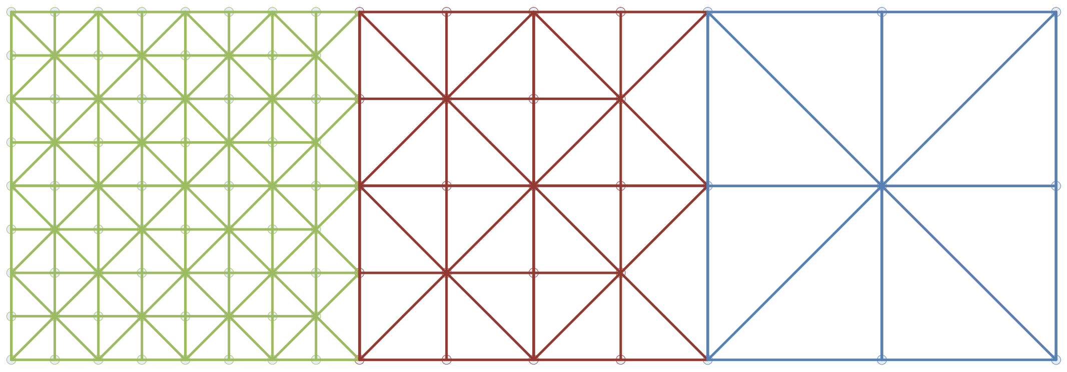


Figure 4.25: The rendering of three surface patches that are adjacent to each other and are at different levels-of-detail using the presented approach.

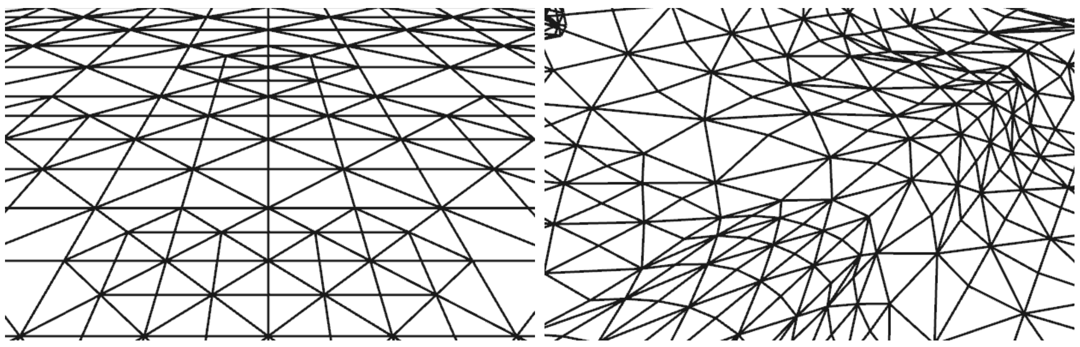


Figure 4.26: The geometry resulting from the application of level-of-detail. Top: a flat surface rendered with level-of-detail, and bottom: part of a sample terrain surface rendered with level-of-detail.

Sample geometry resulting from the application of the presented approach to 3D surfaces is shown in Figure 4.26. Please note that in a real application surface patches that are so close to each other do not have different levels-of-detail. In this case, however, d_{max} is deliberately chosen very small for demonstration purposes so that surface patches at different levels can be observed in a very small area.

4.4.4 Smooth Level-of-Detail Transitions

Besides the visual artifacts caused by level-of-detail, another related problem is the popping effect. Popping is evident when a the level-of-detail index of a surface patch changes as this event triggers a sudden change of the geometry. Moreover, in our approach, as in most other distance-based approaches, a group of surface patches tend to change level-of-detail at about the same time as each of their distances pass a preset threshold value. This is obviously not an issue for still image rendering. Our approach, on the other hand, targets real-time rendering applications where it is not unrealistic to expect an always-moving camera. This sudden change in the geometry has a serious negative impact on visual quality in animated renderings.

One solution to the popping geometry problem is to determine the surface patches that are about to change the level-of-detail and render them twice, once at a higher-level and then again at a lower level, and blend the resulting renderings depending on the distance. This approach is simple as it is independent from the representation and geometry of the terrain. It, however, requires rendering even more triangles than rendering the surface patch at the higher level. It, somewhat, diminishes the gains of level-of-detail. It also does not produce visually great results since it is an image-based method and basically blends two images of the surface patch at different levels.

The other solution is to gradually blend the geometry of the higher-level surface patch to the lower-level geometry. This approach can produce visually very satisfying results and it also does not require excessive rendering of triangles. The difficulty of this approach is that blending two different geometries together is a very difficult problem, one that is impractical for real-time rendering, if the terrain representation is not designed for such a task.

The proposed terrain representation constrains the geometry of the terrain in ways such that the resulting surface geometry is very similar to a regular flat heightmap grid. Consequently, we are able to employ the geometry-blending technique to solve the popping problem similar to its applications in heightmap-based terrains.

In the presented level-of-detail scheme, at each lower-level triangles from the higher-level are combined to obtain larger triangles, and as a result of this operation most of the vertices and edges that are rendered at the higher-level are not rendered in the lower-level. Figure 4.27 shows which vertices and edges disappear when the level of a surface patch decreases from two to one. It is possible to gradually transition the positions of the disappearing vertices before they disappear such that at the time they disappear there is visually no difference between their existence and the lack of their existence. The AB and BC edges, for instance, are rendered at level-2 and the AC edge is rendered at level-1. The transition is then visually undetectable if the vertex B is located on the AC edge during transition, because then rendering AB and BC separately essentially produces the same result as rendering just AC.

It is very important to note that every red vertex (i.e., the vertices that disappear at the lower-level) are on one-and-only-one black edge (i.e., an edge that do not disappear at the lower-level). Each of these black edges have blue vertices on each end of the edge. The blue vertices do not disappear at the lower-level as well, and that is simply why the black edges do not disappear. This means that there are exactly two blue vertices for each red vertex where we can use the locations of the blue vertices to compute where the red vertex should be

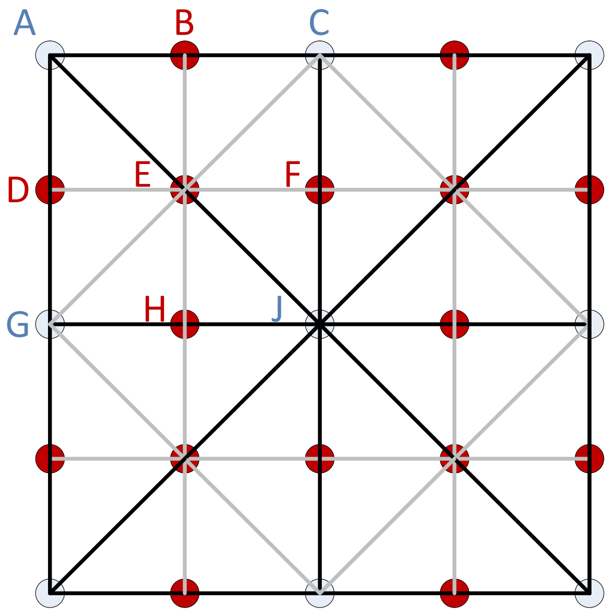


Figure 4.27: The vertices that disappear in a lower level-of-detail are colored red, and the disappearing edges are colored gray.

at the moment of transition. For instance,

- B must be at the center of AC,
- D must be at the center of AG,
- E must be at the center of AJ,
- F must be at the center of CJ, and
- H must be at the center of GJ.

The transition of the vertex position is desired to be gradual, however, continuous transition of vertex positions may as well reduce the visual quality. What we want is to have the transition start sufficiently earlier than the change of level and reach the desired position right at the moment when the level of the surface patch changes. The proposed approach computes the position of each vertex P by Equation (4.15) where

- k is the lowest level that this particular vertex can be rendered,

- d_{lod} is the length of the range in which the surface patches are rendered at level-k,
- l_t is the length of the transition measured in distance-unit rather than time,
- σ is a constant in the interval [0, 1] and it determines the length of the transition as a ratio of d_{lod},
- t_s and t_e are the distances at which the transition starts and ends, respectively,
- s_p is the size of a single surface patch,
- d_v is the distance of the vertex to the observer,
- d_n is the normalized distance of the vertex to the observer which is in the interval [0, 1],
- P_o is the original position of the vertex, and
- P_t is the position where the vertex is desired to be at the end of the transition.

$$\begin{aligned}
d_{lod} &= \frac{2^2 \times d_{max}}{2^k} \\
l_t &= d_{lod} \times \sigma \\
t_s &= 2 \times d_{lod} - l_t - \frac{s_p}{2} \\
t_e &= t_s + l_t \\
d_n &= max\{0, min\{1, \frac{d_v - t_s}{t_e - t_s}\}\} \\
\overrightarrow{P} &= \overrightarrow{P_o} \times (1 - d_n) + \overrightarrow{P_t} \times d_n \qquad (4.15)
\end{aligned}$$

Equation (4.15) results in linear translation of vertex positions. Our experiments have shown that using a quadratic or spline motion curve instead of a linear one does not result in any notably better results. Our experiments have also shown that choosing $\sigma = 0.3$ yields visually good results in most cases.

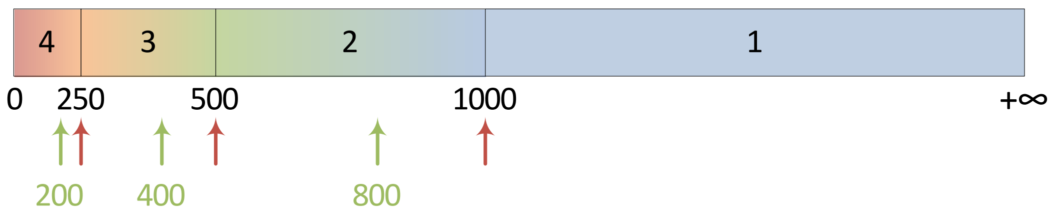


Figure 4.28: Sample level-of-detail range including approximate transition points where the green arrows show the points where the transition starts and red arrows show the points where the transition ends.

It should be noted that in the proposed approach the actual level-of-detail algorithm that determines the level-of-detail of each surface patch runs on the CPU. This algorithm cannot run at the vertex-level, hence we approximate the distance between the observer and each vertex of surface patch by computing the distance between the observer and the center of the bounding box of the vertices of that surface patch. The algorithm that computes the actual position of each vertex by using Equation (4.15), on the other hand, works on the GPU and does actually run at the vertex-level. In fact, the algorithm that runs on the GPU does not know anything about the surface patches, it only knows about the vertices. Thus, the distance computed by the transition algorithm is the actual distance of the vertex to the observer. There is a slight difference between the distances computed by two different algorithms, the on that runs on the CPU and the one that runs on the GPU. This can cause the level-of-detail index of a surface patch to change before each of its vertices completes the transition. This is the reason the half size of the surface patches $s_p/2$ is subtracted while computing t_s. By doing so, we guarantee that all vertices complete the transition at the point of level-of-detail change. The side-effect of this is that some vertices may complete their transition slightly earlier, but this is not a serious problem as it is visually undetectable. Figure 4.28 shows an example level-of-detail range including the transition points that are computed by Equation (4.15) where $k_{max} = 4$, $d_{max} = 1000$, and $\sigma = 0.4$. s_p is disregarded in this case as it is too small, in typical scenes it is equal to about $d_{max} \times 0.01$.

It should be noted that this transition prevents geometry popping, nevertheless it does not help with other effects unrelated to the positions of the vertices such as texturing and lighting. Texturing and lighting uses vertex normals and even though vertex positions are smoothly transitioned the vertex normals still change suddenly at the point of level-of-detail change. This causes the textures and lighting effects on the surface to change suddenly, which is sometimes even worse than geometry popping as it usually affects a greater area on the screen. Our proposed approach interpolates the surface normals as well to overcome this problem. It is essentially the same as vertex position transition, but instead the normals of the surrounding vertices are used to compute the transition normal.

The presented smooth level-of-detail transition algorithm is required to run during shadow map generation, too. Otherwise the shadows will continue to pop even though the actual geometry does not since the geometry that is used to compute the shadows is not transitioned. The proposed approach uses vertex position transition both at the shadow map generation stage and the actual rendering stage to get rid of shadow popping. The transition of surface normals, on the other hand, is redundant during shadow map generation as the surface normals are irrelevant to shadow computations.

Chapter 5

Implementation and Performance

During the course of this research, a sample application is designed and developed for

- creating and editing sample terrains to use during the experiments,
- analyzing the memory usage of the proposed terrain representation in typical use cases,
- analyzing the performance of the proposed algorithms, and
- evaluating the visual quality of the real-time terrain rendering obtained by using the proposed rendering pipeline.

This chapter discusses the implementation details of the sample application developed, the details of the sample scene as well as the hardware and software configuration of the test environment used to evaluate performance and memory usage, and the overall performance and efficiency of the proposed approach for typical use cases. A discussion about how the proposed approach performs in comparison to other approaches is also included at the end of this section.

5.1 Implementation Overview

The application is implemented entirely in the C++ programming language as a native application for the Microsoft Windows operating system using the Windows application programming interface (API), also known as the Win32 API. The implementation, though, is almost entirely platform independent and in theory it could easily be ported to other operating systems, e.g., to Linux. Specifically, the implementation of the proposed real-time terrain rendering system does not rely on any exclusive features of the underlying operating system, such as a paging system or an API that is exclusive to an operating system (e.g., Direct3D for 3D rendering). The application code is compiled and linked by the Microsoft Visual Studio 2010 development environment and its C++ compiler. An exhaustive list of the external libraries that are used are as follows:

Open Graphics Library (OpenGL) [40] is a standard specification that defines an API for developing applications that produce 2D and 3D computer graphics. OpenGL is used to implement the entire rendering pipeline that is presented in this thesis, using hardware-accelerated rendering capabilities of modern GPUs.

The OpenGL Extension Wrangler Library (GLEW) [41] is a cross-platform open-source C/C++ extension loading library for OpenGL extensions. Many advanced graphics features that are supported by modern GPUs are not added to the core OpenGL functionality until they become mainstream, since all the GPUs that support OpenGL would then have to implement them. These features are instead added as extensions to OpenGL. The presented rendering pipeline makes heavy use of such advanced features and GLEW makes it very easy to manage and use these extensions.

The Developer's Image Library (DevIL) [42], which is also known as the Open Image Library (OpenIL), is a cross-platform open-source image library with powerful image loading and saving capabilities. It is basically used to load image data for textures and heightmaps from files, and save image data to files, such as screen captures. It supports all of the image file formats used in the sample application, such as bitmap (BMP), portable network graphics (PNG), and tagged image file format (TIFF).

FreeType [43] is a cross-platform open-source software font engine that is used to produce high-quality glyph images of the symbols in fonts. It is used to generate textures for rendering text using OpenGL on the viewport. Text rendering is mainly used for debugging and performance evaluation purposes.

All of the code for the sample application, except for the libraries listed above, are written specifically for this project without the use of any other external libraries and dependencies.

5.2 Terrain Editor

One of the features of the developed sample application is its ability to create and edit terrain in real-time using the terrain deformation abilities of the proposed approach. The terrain creation process can be divided into two steps as follows:

1. Create the coarse volumetric representation of the terrain outlining all volumetric terrain features.
2. Edit terrain surface and apply fine details at an increased resolution over the coarse terrain model by modifying heightmaps that are applied to the surface patches.

5.2.1 Editing Coarse Terrain Model

The coarse terrain model is represented by the voxel representation. It is edited by turning each voxel on and off. The resolution is not static as an octree data structure is used to store the voxel representation. The coarse terrain model can be edited manually by selecting a specific voxel at a specific level of the octree and setting whether the voxel is filled or not. It can also be automatically created, entirely or partly, from a heightmap. After the heightmap is converted to the

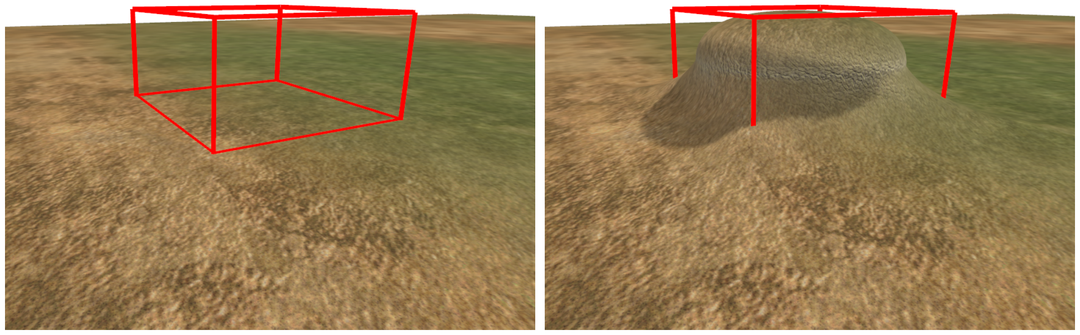


Figure 5.1: Manual editing of voxel model where the highlighted voxel is shown as a red box. Left: the highlighted voxel is empty, right: the highlighted voxel is filled.

voxel model, the voxel model can then be edited manually to create volumetric terrain features at the desired locations on the terrain.

The generation of the coarse model from heightmap data is quite simple. The heightmap is first divided into blocks depending on the desired resolution of the coarse terrain model. Then the average height of each block is computed. In this case, a set of voxels lined up along the y-axis corresponds to each heightmap block. The ones that are below the average height are then set as filled with matter while the other ones are left empty. This voxel model can then be carved manually to generate the volumetric terrain features, such as a cave.

The voxel editor mode is very simple, probably too simple for an artist. It is sufficient for our purposes, though. The editor lets the user to select the level of octree on which he wants to operate. Then the position of the cursor is used to compute a ray through the scene and the closest voxel at the selected level that intersects the ray is highlighted. The highlighted voxel can then be set as filled or empty (see Figure 5.1). Changing the fill status of a voxel at lower levels of the octree automatically increases the level of the octree at that branch by creating child in each higher level. Changing the fill status of a voxel at a higher level gets rid of its subtree in the octree for optimization, since that space can now be entirely represented by a single voxel.

The computation of the voxel selection ray is performed by a number of transformations applied to the 2D cursor position. Algorithm 5.1 is used to compute

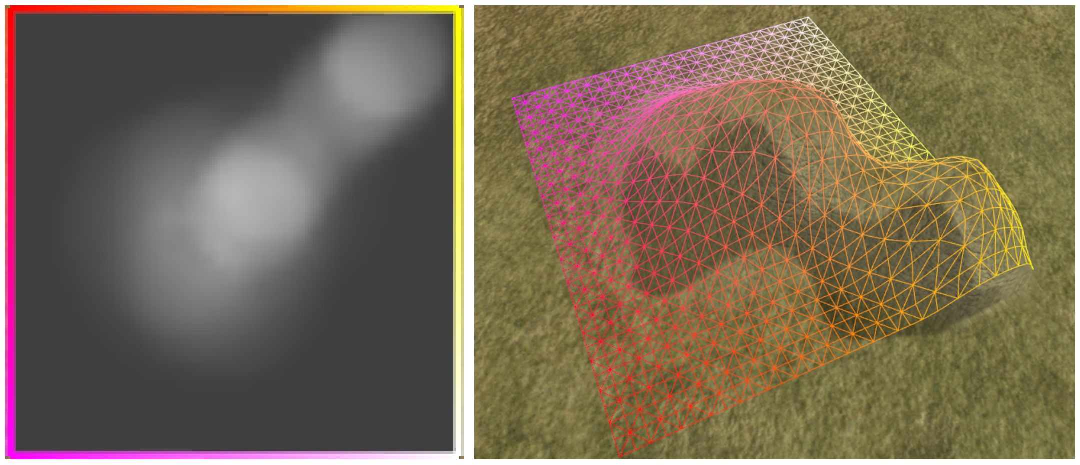


Figure 5.2: Editing terrain surface can be done by modifying the heightmap that is applied to the surface. Left: the heightmap applied to the surface patch, right: the surface patch being edited.

the voxel selection ray given the 2D coordinates of the cursor on the screen. This finite ray is then used to determine the closest intersecting voxel at the desired level of the octree.

5.2.2 Editing Terrain Surface

The terrain surface can be edited by applying deformations to the heightmaps. Each voxel is associated with a number of surface patches depending on the voxel configurations. Once a voxel is selected by the user, one of the surface patches that are associated with the selected voxel can be edited by editing the heightmap associated with that surface patch. A heightmap is always displayed as a 2D square texture on the user interface to allow simple editing operations. The surface patch on which the heightmap is applied, on the other hand, can be in various different shapes and orientations. The mapping between the heightmap and the surface patch can sometimes be unintuitive. In order to help user understand which part of the heightmap corresponds to which part of the surface patch, the edges of the heightmap and the wireframe skeleton rendered on the selected surface patch are colored accordingly (see Figure 5.2).

The heightmap can be directly edited by the user. The changed to the heightmap values are immediately reflected to the surface patch in real-time so

```
cursorPos: (input) 2D position of the cursor on application viewport
viewport : (input) Viewport attributes on which the rendering is done
camera   : (input) Attributes of the perspective camera that is used for
           rendering
rayStart : (output) Starting position of the ray in world coordinates
rayEnd   : (output) Ending position of the ray in world coordinates
begin
    // compute normalized cursor coordinates
    $normalizedCursorX = (cursorPos.X \,/\, viewport.width) \times 2 - 1$
    $normalizedCursorY = 1 - (cursorPos.Y \,/\, viewport.height) \times 2$
    if $viewport.width > viewport.height$ then
        $normalizedCursorX = normalizedCursorX \times viewport.width \,/\, viewport.height$;
    else
        $normalizedCursorY = normalizedCursorY \times viewport.height \,/\, viewport.width$;
    // the field of view is in radians
    $fovFactor = tan(camera.fieldOfView \times 0.5)$
    $normalizedCursorX = normalizedCursorX \times fovFactor$
    $normalizedCursorY = normalizedCursorY \times fovFactor$
    // compute the starting position of the ray
    $rayStart = \{$
        $normalizedCursorX \times camera.nearClipDistance$,
        $normalizedCursorY \times camera.nearClipDistance$,
        $-camera.nearClipDistance$, $1\}$
    // compute the ending position of the ray
    $rayEnd = \{$
        $normalizedCursorX \times camera.farClipDistance$,
        $normalizedCursorY \times camera.farClipDistance$,
        $-camera.farClipDistance$, $1\}$
    // transform ray definition into world space;
    $inverseViewMatrix = inverse(camera.viewMatrix)$;
    $rayStart = inverseViewMatrix \times rayStart$;
    $rayEnd = inverseViewMatrix \times rayEnd$;
```

Algorithm 5.1: Computing the start and end positions of the ray in world coordinates that is defined by the cursor

that the user can see the resulting effects of his actions. This immediate feedback is supported by the proposed terrain representation and rendering approach, since it supports terrain deformation in real-time, and this feature is certainly very valuable to terrain designers.

The heightmap can be edited by the user through the use of several different types of brushes. The sample application defines the following types of brushes for editing heightmaps:

Extrude brush is used to elevate corresponding parts of the surface patch depending on where the brush is applied of the heightmap. The brush has three parameters: radius, strength and displacement factor.

- Radius defines the effective radius of the brush such that the pixels of the heightmap that are outside this radius will not be affected by the brush.
- Strength determines how the effect of the brush will diminish as it is applied to the pixels further away from the brush application position.
- Displacement factor determines the magnitude of the displacement applied by the brush. If the displacement factor is positive than the vertices are displaced along the direction of their displacement normals, otherwise the vertices are displaced in the opposite direction.

It is usually desired that the effectiveness of the brush drops as it is applied to pixels that are further away from the center of the region where the brush is applied. This is useful to make smooth touches to the terrain surface. We use a sigmoid function to compute the effectiveness of the brush at a given distance. The effectiveness is computed according to Equation (5.1) where r is the radius of the brush, s is the strength and d is the distance of the pixel to the brush application position. The constant δ is used to convert the distance of normalized pixel coordinates on the image space to a distance comparable to brush radius. It is chosen $\delta = 20$ in the sample application. Figure 5.3 demonstrates the effectiveness functions for two sample brushes with different strengths.

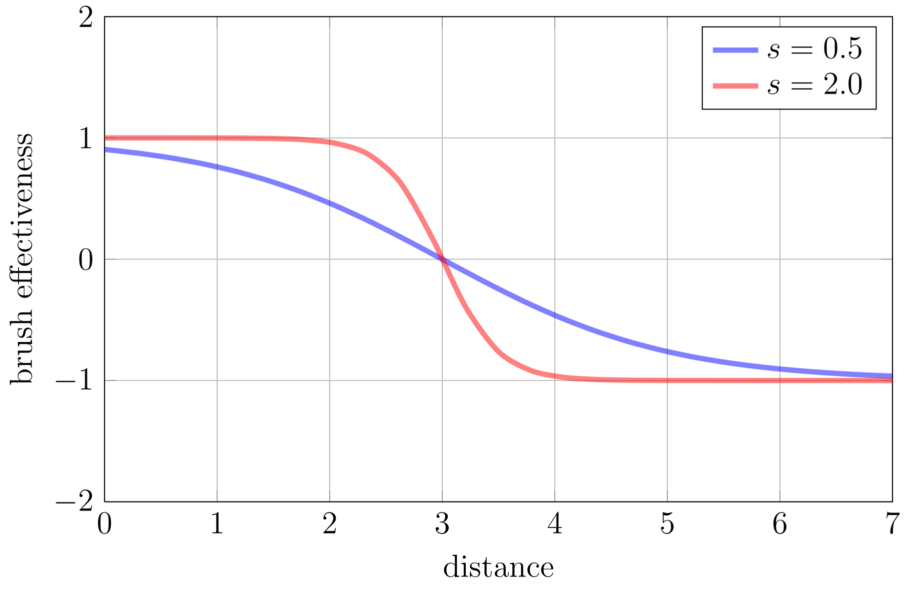


Figure 5.3: Sigmoid function that is used to compute the effectiveness of a brush at a particular distance. Blue: a brush with radius $r = 3$ and strength $s = 0.5$, red: another brush with radius $r = 3$ and strength $s = 2$.

$$tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$
$$f(d) = tanh\big((r - d \times \delta) \times s\big) \tag{5.1}$$

Smoothing brush is used to smooth parts of the terrain where the brush is applied. It has three parameters similar to the extrude brush: the radius, strength and the smoothing factor. The radius and strength are the same as the extrude brush. The smoothing factor determines how aggressively the smooth operation is desired to be performed. If the smoothing factor is large then the roughness of the surface patch is quickly smoothed out.

Noise brush is used to simply add fine detail in the form of noise on the heightmap. 3D Simplex noise [38, 39] is used to create noise where the input to the noise function is the actual world coordinates of the vertices that correspond to the edited pixels of the heightmap. This way, the continuity of the 3D surface is guaranteed after the application of the noise as the 3D world coordinates of the vertices are continuous.

5.2.3 Saving and Loading Terrain Data

The sample application is also able to save and load terrains. The saved terrain data only includes the coarse voxel model and the heightmaps for each surface patch. It does not store the entire content of the data structures in memory, such as vertices that approximate surface patches, shared vertex lists, displacement normals, and surface normals. The only data that needs to be persistent is the coarse voxel model and the heightmaps. Everything else is re-computed when the terrain data is loaded from file. First, the coarse voxel model is loaded and the surface extraction process is executed. Then, the surface generation process is executed, and finally the heightmaps are loaded from the file and applied to the corresponding surface patches. This greatly decreases the disk space required to store the terrain data at the cost of longer loading times, which is not that important for our purposes.

5.3 Rendering Pipeline

The rendering pipeline of the proposed approach is implemented using OpenGL version 3.2. Fixed function pipeline is completely removed from OpenGL in this version. The rendering pipeline presented in this section is implemented completely by programming the GPU using custom vertex and fragment shaders for rendering. The shader programs are written in the GL Shading Language (GLSL) [44].

Figure 5.4 shows an overview of the rendering pipeline. The rendering pipeline can be divided into four major steps:

1. Vertex buffer updates (green parts on Figure 5.4),
2. index buffer updates (red parts on Figure 5.4),
3. generating shadow maps (blue parts on Figure 5.4), and
4. terrain surface rendering (orange parts on Figure 5.4).

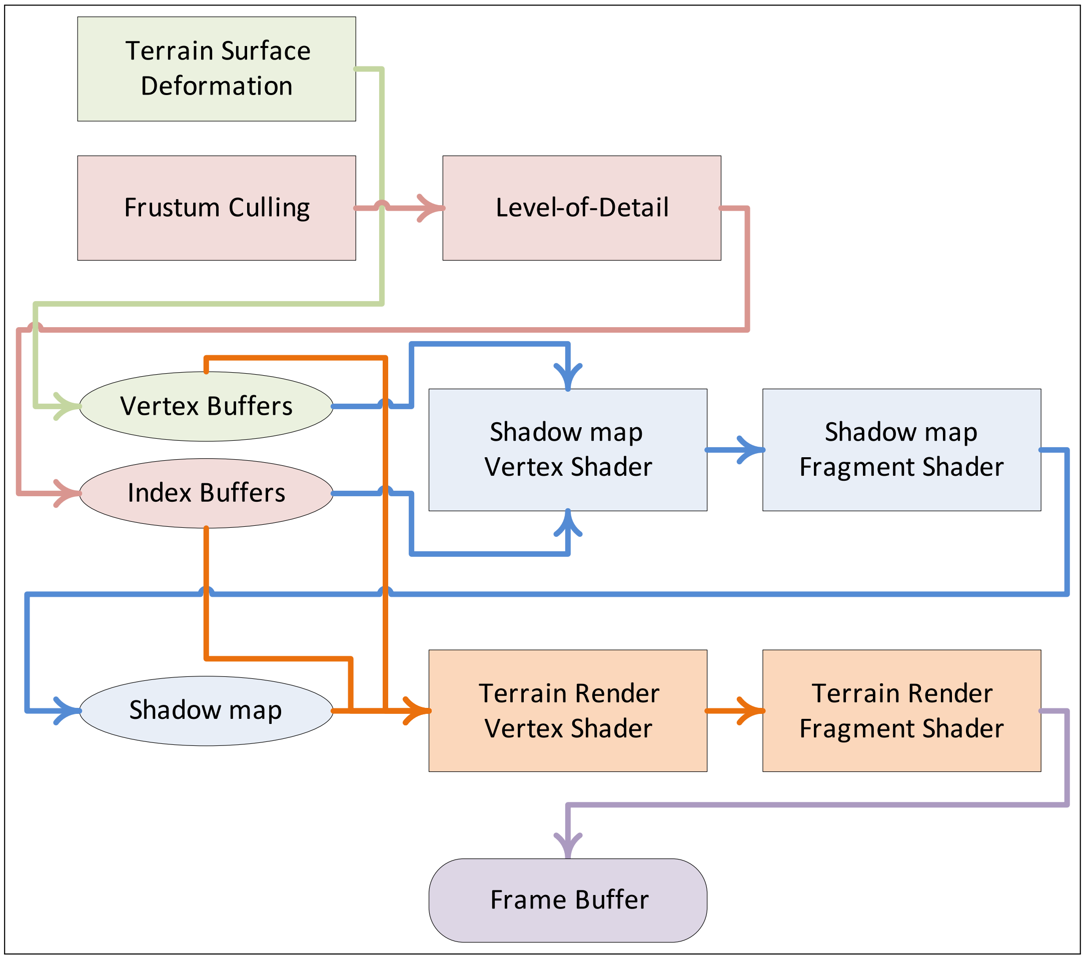

Terrain Surface Deformation
Frustum Culling
Level-of-Detail
Vertex Buffers
Index Buffers
Shadow map Vertex Shader
Shadow map Fragment Shader
Shadow map
Terrain Render Vertex Shader
Terrain Render Fragment Shader
Frame Buffer

Figure 5.4: Rendering pipeline that is used in the sample application.

5.3.1 Vertex Buffer Updates

The proposed approach uses vertex buffers to store vertex attributes, such as the vertex position, vertex normal, and geometry morphing parameters. The advantage of vertex buffers is that they are stored directly on the video memory. Consequently,

- GPU access to data stored in vertex buffers is extremely fast, and
- data that is stored in the main memory do not need to be transferred to the GPU in each frame, causing a possible CPU-to-GPU data transfer bottleneck.

The vertex buffers are filled with data once the terrain surface generation is completed and the terrain surface is ready to be rendered for the first time. As long as the terrain surface is not deformed, it is not needed to update vertex buffers since all vertex data that is needed for rendering is already in the video memory. Whenever some part of the terrain surface is deformed, the attributes of the vertices that are affected by the deformation are updated. These updates are initially performed in the main memory by the algorithms discussed in Section 3.5.5. The relevant parts of the vertex buffer must then be updated with the data in the main memory.

5.3.2 Index Buffer Updates

Index buffers are also stored in the video memory, similar to vertex buffers, but they store the connection information for vertices, that is, how they should be connected to form triangles. Index buffer is the actual list of primitives to be rendered by the GPU. If the index buffer is empty, the GPU renders nothing as it does not know what to make of the vertices in the vertex buffer. Therefore, the number of primitives represented by the index buffer actually determines the rendering performance. Because of this, the index buffer is updated at each frame as the frustum culling and level-of-detail functions are executed.

The algorithm that updates the index buffer operates on the octree. Each node of the octree is first checked to see if its terrain surface bounding box is completely outside the frustum. It it is, then the subtree of that node is completely eliminated from further processing as it does not contribute to the rendering process in any way and any further processing would be redundant as a consequence. The level-of-detail of the surface patches that are associated with the voxels that pass the frustum test are then computed. The corresponding triangle vertex index data of these surface patches are copied to the index buffers as it is described in Section 4.4.3.

5.3.2.1 Frustum Culling

Frustum culling is a method that is used to improve rendering performance. Its main goal is to discard large portions of the scene with simple occlusion checks to prevent redundant processing of geometry that is not supposed to contribute to the rendering. An occlusion check basically tells whether a particular geometry is inside the view frustum or not. Performing a frustum check on a complex 3D object is very expensive. Thus, frustum checks are usually performed on bounding boxes of objects where each corner of the bounding box is tested against the frustum planes. If every corner of the bounding box is on the outside of at least one of the frustum planes then the bounding box is definitely not inside the frustum, not even partially, and the object represented by the bounding box can safely be discarded.

It should be noted that in the proposed terrain representation it is not possible to use the voxels in frustum checks since the displaced terrain surfaces may overflow voxel limits. For frustum culling, therefore, a hierarchy of surface bounding boxes are computed and stored in the octree. These bounding boxes are computed in a bottom-up manner. The leaf nodes in the octree are assigned bounding boxes that are computed by the bounding boxes of the surface patches assigned to them. Then, the bounding box of each higher-level node is computed by taking the union of the bounding boxes of each child node. When a surface patch is deformed, its bounding box is updated as well. In this case, the bounding boxes of all octree nodes on the path from the root to the leaf node that contains the deformed surface patch are updated, again in a bottom-up direction. This is not an expensive operation as the depth of an octree is not very high in the proposed representation, since the octree only defines a coarse representation of the terrain.

Please note that frustum culling must be repeated whenever the view frustum changes, meaning that when four cascaded shadow maps are used, frustum culling is repeated four times for rendering the shadow maps and another time for the rendering of the actual terrain surface at each frame. Frustum culling, however, helps improve rendering performance even when it is repeated multiple times per frame since, depending on the viewpoint, usually a very small portion of the terrain is in the frustum.

5.3.3 Generating Shadow Maps

The terrain surface geometry is rendered once for each cascaded shadow map as it is described in Section 4.3 in detail. The vertex and fragment shaders used to render shadow maps are relatively simple.

The vertex shader basically computes the distance of the vertex to the observer and then applies geometry morphing to the vertex as it is described in Section 4.4.4 in detail. It then multiplies the computed world coordinate with the model-view-projection matrix of the light to compute the projected fragment coordinate. The fragment shader then takes the z-component of the fragment coordinate, adds an offset to it to prevent z-fighting, and writes the computed value to the shadow map.

5.3.4 Terrain Surface Rendering

Once the shadow maps are generated, the actual rendering can be done using the terrain surface geometry, shadow maps and the other textures. The vertex and fragment shaders that are used at this stage are more complex.

The tasks performed by the vertex shader is as follows:

- The vertex shader normalizes the surface normal since it is not normalized by the CPU. Normalizing surface normals by the CPU requires another pass over all vertices which is quite expensive, but it is required to be done only once unless the surface geometry is updated. The cost of normalization by the GPU is almost zero but it has to be performed at each frame.

The reference implementation normalizes surface normals on the GPU as it improves performance when the terrain surface is constantly updated (e.g., during terrain editing).

- The distance of the vertex to the observer is computed.
- The level-of-detail transition position of the vertex is computed using the distance to the observer.
- The level-of-detail transition normal of the vertex is computed using the distance to the observer.
- The world coordinate of the vertex is projected into the eye space by multiplying by the view-projection matrix of the perspective camera.

The tasks performed by the fragment shader is as follows:

- The fragment shader normalizes the interpolated vertex normal since interpolating between two normalized vertex normals does not always result in a normalized vertex normal.
- Tri-planar texture coordinates are computed as it is described in Section 4.2.1 in detail.
- Simplex noise values are computed for use in multi-texturing.
- Multi-texturing and texture splatting is applied as it is described in Sections 4.2.2 and 4.2.3 in detail.
- Per-pixel directional lighting computations are performed.
- Per-pixel point light computations are performed for each point light in the scene.
- Shadow maps are sampled several times and blended using Gaussian filtering.
- Output color values of the texturing, lighting and shadowing phases are blended to compute the final pixel color.

5.4 Performance and Memory Usage

This section discusses the performance and memory usage of the reference implementation.

5.4.1 Test Environment

The hardware and software features of the reference implementation test environment are specified in Table 5.1. Please note that the most influential element on rendering performance is the GPU and the GPU that is used in our tests is about four years old at the time of writing this thesis. It still supports most of the advanced techniques and is considered as a modern GPU. The performance results presented in this section, nevertheless, do not imply an upper limit as the GPU used in these tests are not state of the art. Using the most powerful consumer GPU available today could easily double, triple, and even quadruple the performance figures presented. The proposed approach and the reference implementation is still efficient enough to render large terrains in real-time, though. Using a more powerful GPU, on the other hand, could easily allow real-time rendering of much larger terrains with better visual quality.

Operating system	Windows 7 Professional x64, SP1
CPU	Intel Core i7-2600K 3.4 GHz
# CPU cores	4 + 4 (Hyper-Threading)
Main memory	8 GB DDR3
GPU	nVidia GeForce GTX260
GPU driver version	301.42
GPU graphics clock	650 MHz
GPU processor clock	1400 MHz
GPU memory clock	1000 MHz
Video memory	896 MB GDDR3
CPU-GPU data bus	PCI Express x8 Gen2

Table 5.1: Test environment of the reference implementation.

Although many efficiency related improvements have been made in the reference implementation, the implementation is not in a production-ready state. There are many possible optimizations that can be applied to different stages of the rendering pipeline. The implementation does not use vendor-specific and hardware-specific extensions available in OpenGL. These extensions could easily improve the performance further. Since multi-threading is not used in the reference implementation, only a single core of the eight-core CPU is utilized. The sample application uses CPU merely for terrain rendering and the CPU intensity of the proposed approach is very low, even when the terrain deformations are applied in real-time. Consequently, the CPU, stalls most of the time waiting for the GPU to finish its job. More CPU tasks could be set up to run in the idle-time, such as occlusion culling, artificial intelligence, and physics computations, without having a negative impact on the rendering performance.

5.4.2 Timers

High-resolution timers are used to measure the time it takes to complete an operation. Measuring time spent by the CPU is relatively easy. A high-resolution Windows-specific timer [45] is used for this task. The granularity of the CPU timers are on the order of nanoseconds.

Measuring time spent by the GPU is slightly more complex. The GPU has a job queue, and the OpenGL commands are converted to GPU tasks by the GPU driver and added to the queue. The GPU then executes these tasks asynchronously. Measuring the CPU time only gives the time it takes to add the command to the job queue of the GPU, and is therefore meaningless. We use GPU timer queries [46] provided by the OpenGL specification. These timer queries are added to the job queue of the GPU by the CPU before and after a particular command is executed. The GPU, consequently, executes the timer tasks before and after that particular command is executed. The results of the timer queries are asynchronously received by the CPU after the execution of the corresponding commands are completed by the GPU. Although the resolution of these GPU timers are hardware-dependent, the GPU we used for these tests have a timer granularity in the order of nanoseconds.

5.4.3 Test Scene

Statistics of the sample terrain that is created for performance evaluation is given in Table 5.2. The number of nontrivial voxel intersection volumes are those for which one or more surface patches are generated.

# filled voxels	24239
# nontrivial voxel intersection volumes	2544
# surface patches (without static surface culling)	7481
# surface patches (with static surface culling)	3560

Table 5.2: Coarse terrain model statistics of the test scene.

The terrain surface is generated with different parameters throughout the testing. The statistics of the terrain surface are given in Table 5.3 for different parameters.

maximum lod	# patch vertices	# patch triangles	# total vertices	# total triangles
1	3×3	8	32K	28K
2	5×5	32	89K	114K
3	9×9	128	288K	456K
4	17×17	512	1M	1.8M
5	33×33	2048	3.9M	7.3M
6	65×65	8192	15M	29.2M

Table 5.3: Terrain surface statistics of the test scene for different maximum levels-of-detail.

Figure 5.5 show several screen captures of the test scene from different viewpoints.

5.4.4 Memory Usage

Most of the memory usage is required to store the vertex attributes, both on the main memory and on the video memory. The vertex attribute data structure that is stored for each vertex on the main memory is shown in Table 5.4.

Vertex attributes are stored in a similar data structure on the video memory as shown in Table 5.5. This data structure is filled using the attributes that are stored on the main memory and then loaded to the video memory.

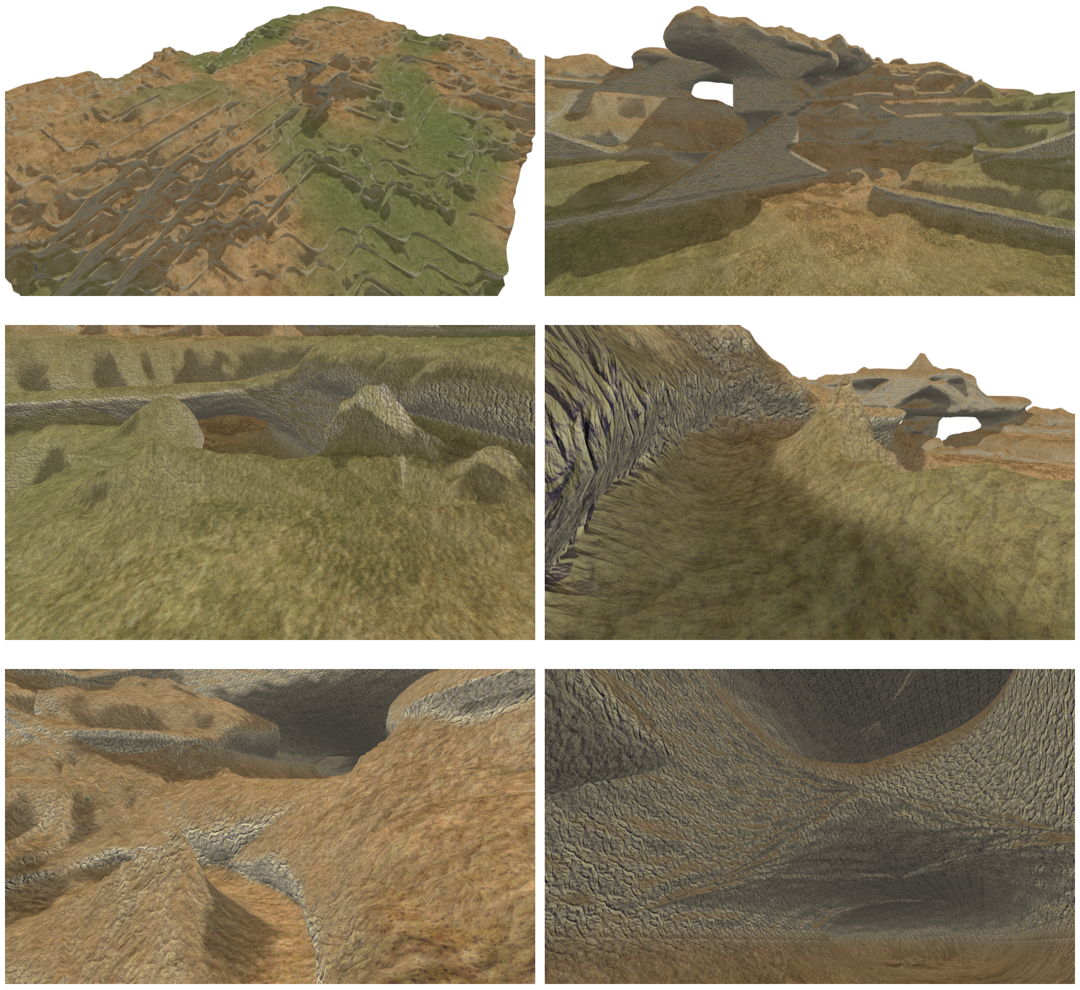


Figure 5.5: Screen captures of the test scene.

vertex attribute	data type	size
original position	float [3]	12-bytes
displacement normal	float [3]	12-bytes
actual normal	float [3]	12-bytes
displacement	float	4-bytes
lower level-of-detail neighbor [2]	pointer [2]	8-bytes
level-of-detail index	unsigned int	3-bits
re-compute normals	unsigned int	1-bit
vertex buffer index	unsigned int	28-bits

Table 5.4: Vertex attributes that are stored on the main memory.

Storing the attributes of each vertex requires 52-bytes both on the main memory and on the video memory. It should be noted that it is required to store the vertex attributes on main memory so that when the terrain is deformed the vertex attributes can be updated. The vertex attributes in the main memory can be completely discarded if real-time terrain deformations are not needed.

vertex attribute	data type	size
position	float [3]	12-bytes
normal	float [3]	12-bytes
geomorph transition threshold	float	4-bytes
geomorph transition position	float [3]	12-bytes
geomorph normal	float [3]	12-bytes

Table 5.5: Vertex attributes that are stored on the video memory (i.e., in vertex buffers).

The main memory usage of the reference implementation for the test scene is shown in Table 5.6. Memory space required to store the vertex attributes and triangle vertex indices are the most significant, rest of the memory usage is negligible.

maximum lod	vertex attributes	shared vertices	index data	heightmaps
1	0.3 MB	0.67 MB	512 KB	0.1 MB
2	1.6 MB	1.5 MB	1.7 MB	0.4 MB
3	9 MB	3.2 MB	7.3 MB	1.2 MB
4	40 MB	6.5 MB	29 MB	4.1 MB
5	175 MB	13.3 MB	117 MB	15.5 MB

Table 5.6: Main memory usage of the reference implementation for the test scene.

The video memory usage of the reference implementation for the test scene is shown in Table 5.7.

maximum lod	vertex attributes	index data (avg.)	textures
1	1 MB	60 KB	6 MB
2	3 MB	125 KB	6 MB
3	12 MB	250 KB	6 MB
4	46 MB	1 MB	6 MB
5	188 MB	4 MB	6 MB

Table 5.7: Video memory usage of the reference implementation for the test scene.

Video memory space required to store shadow maps are not included in Table 5.7 since it depends on the number of cascaded shadow maps used. Table 5.8 shows the memory usage of the shadow maps on video memory. Please note that shadow maps used in the reference implementation have 24-bit depth.

# cascaded shadow maps	resolution	size
1	1024×1024	3 MB
	2048×2048	12 MB
	4096×4096	48 MB
2	1024×1024	6 MB
	2048×2048	24 MB
	4096×4096	96 MB
3	1024×1024	9 MB
	2048×2048	36 MB
	4096×4096	144 MB
4	1024×1024	12 MB
	2048×2048	48 MB
	4096×4096	192 MB

Table 5.8: Video memory required to store shadow maps.

The reference implementation uses four cascaded shadow maps of resolution 2048×2048, hence the video memory required is 48 MB.

5.4.5 Performance

This section discusses the runtime performance of the reference implementation.

5.4.5.1 Terrain Surface Generation Performance

The performance-related statistics of the terrain surface generation process is shown in Table 5.9.

maximum lod	surface extraction	surface approximation	shared vertices	normals	indices
1	40 ms	15 ms	55 ms	6 ms	4 ms
2	40 ms	20 ms	70 ms	22 ms	13 ms
3	40 ms	40 ms	110 ms	60 ms	50 ms
4	40 ms	90 ms	230 ms	240 ms	150 ms
5	40 ms	210 ms	660 ms	890 ms	460 ms

Table 5.9: Performance of the terrain surface generation process.

The columns in Table 5.9 represent the time required to

- extract the surface of the coarse voxel model,
- approximate the surface patches by creating vertices,
- find shared vertices between surface patches,
- compute displacement and surface normals, and
- compute triangle vertex indices for each surface patch and different levels-of-detail.

The surface extraction step does not depend on the maximum level-of-detail as it is not affected by the number of vertices each surface patch is approximated with. Please note that these steps are performed only once, at the terrain creation or loading step.

5.4.5.2 Terrain Surface Deformation Performance

When the terrain is deformed in real-time, the vertex displacements are computed, surface normals are re-computed, and updated vertex attributes in the main memory are copied to vertex buffers. Table 5.10 lists the execution time of updating vertex displacements, re-computing surface normals, and updating vertex buffers on video memory.

maximum lod	updating vertex displacements	re-computing normals	updating vertex buffers
1	0.6 μs	6 μs	16 μs
2	2 μs	25 μs	22 μs
3	6 μs	0.1 ms	74 μs
4	24 μs	0.3 ms	0.28 ms
5	0.1 ms	1.2 ms	1.1 ms

Table 5.10: Performance of real-time terrain surface deformation process.

5.4.5.3 Rendering Performance

The rendering pipeline renders the scene geometry in $N + 1$ passes, where N is the number of cascaded shadow maps. The first N passes render each one of the N shadow maps, and the final rendering pass renders the actual terrain surface using the shadow maps and applying various per-pixel effects, such as lighting and texturing. The performance-related statistics of the shadow map rendering passes for shadow maps of resolution 2048×2048 are listed in Table 5.11.

maximum lod	# shadow maps	frustum culling	index buffer updates	rendering
4	1	0.05 ms	0.04 ms	0.6 ms
	2	0.1 ms	0.14 ms	1.1 ms
	4	0.2 ms	0.3 ms	2.3 ms
5	1	0.05 ms	0.3 ms	2.1 ms
	2	0.1 ms	0.6 ms	4.3 ms
	4	0.2 ms	1.1 ms	8.5 ms

Table 5.11: Performance of shadow map rendering passes.

Table 5.12 shows the overall rendering performance of the reference implementation. The last column indicates the number frames-per-second rendered by the rendering pipeline.

5.5 Discussion

In this section, we will discuss how the proposed real-time terrain representation and rendering approach performs when compared to the heightmap- and voxel-based approaches. The comparison is split into subsections for the sake of clarity.

maximum lod	# shadow maps	frustum culling	index buffer updates	rendering	FPS
4	1	0.05 ms	0.15 ms	3.9 ms	235
	2	0.1 ms	0.25 ms	4.4 ms	205
	4	0.25 ms	0.4 ms	5.4 ms	160
5	1	0.05 ms	0.7 ms	10 ms	91
	2	0.1 ms	1 ms	12.1 ms	74
	4	0.25 ms	1.5 ms	16 ms	57

Table 5.12: Overall performance of the proposed rendering pipeline.

5.5.1 Expressiveness

Expressiveness is defined as the ability to create interesting terrain features using a terrain representation. A heightmap samples the terrain in 2D, hence it has a very limited expressive power. It is not possible to represent volumetric terrain features using a heightmap-based representation, such as caves, overhangs, arches, and even vertical cliffs.

Voxel-based representations, on the other hand, sample the true 3D space surrounding the terrain. Thus, the voxel-based representations are extremely good in terms of expressiveness. In fact, assuming an infinitesimal voxel size such that the resolution is extremely high, it is possible to represent all possible kinds of terrain, and any other 3D object for that matter. This is not the case for heightmaps. Heightmaps cannot represent volumetric features even if the resolution is infinitely high. Increasing voxel resolution, though, has serious practical implications such as extremely high memory usage. Doubling the voxel resolution in each axis causes memory usage to increase eightfold.

The proposed hybrid approach basically provides a trade-off between voxel and heightmap representations in terms of expressiveness. It converges to a voxel-representation as the resolution of the coarse voxel model increases, and converges to a heightmap-based representation as it decreases. The practical expressiveness of the proposed approach is less than that of a voxel representation, because as the coarse model gets higher resolution the efficiency of the representation gets worse. However, it is not possible to render very high resolution voxel models in real-time anyway. The proposed approach is able to provide high voxel resolution where volumetric features are dense and complex, and can benefit from the simplicity of the heightmap-based approaches where it is redundant in order to increase the efficiency overall.

5.5.2 Simplicity

By simplicity, the simplicity of the algorithms that operate on the terrain data is meant. Heightmaps are very simple representations and the algorithms that operate on them are also simple. Voxel representation is also quite simple, as a form of volumetric representation. It is possible to write simple algorithms that work on voxel data. It is, however, not plausible to use them in real-time applications. Voxel data is usually huge which increases the complexity of the algorithms that now have to deal with paging data in and out of memory, generating relevant parts of the surface from the voxel model at every frame, compressing and decompressing voxel data, etc.

The proposed representation tries to make use of heightmaps and regular grid surfaces as much as possible. The algorithms, such level-of-detail system, is quite simple thanks to this feature of the proposed representation. It is basically a heightmap representation applied not on a uniform and flat 2D plane, but on a coarse voxel model. This makes it possible to benefit from existing approaches for heightmap-based regular grid representations with minimal changes required for adaptation to the proposed representation.

5.5.3 Efficiency

Efficiency basically covers the memory usage and the processing power demanded by the real-time terrain rendering approach. Heightmaps are quite unbeatable in this respect. It is a very simple, regular and usually a uniform representation of the terrain surface. Its memory requirements are extremely low, even for extremely large terrains. Almost all vertex attributes are implicit in heightmap-based approaches making it redundant to even store the vertices at all. It is, in fact, so simple that given a heightmap image the GPU can create the entire geometry including attributes such as surface normals in an extremely efficient way. This is the very reason why heightmap-based approaches are so dominant in the industry.

One of the design goals of our approach was to benefit from the simplicity and efficiency of heightmap representations. The terrain surface in the proposed approach is quite similar to a regular grid heightmap surface. The proposed approach is obviously not as efficient as heightmap-based representations. This is, however, the cost of the expressiveness that our approach provides. Our approach is, in terms of both memory usage and demanded processing power, still efficient enough to be used in a real-time terrain rendering application meeting with our design goals.

Voxel-based representations, on the other hand, usually require extremely high amounts of memory when they are stored uncompressed and at very high resolution. The memory efficiency can be increased at the cost of processing power by compressing and decompressing the voxel representation as needed. This is, however, very difficult in a real-time rendering application. Voxel representations cannot be directly used for rendering and, unlike heightmaps, it is not straightforward to generate the polygonal surface of a voxel model. Doing this in real-time and at every frame, such as for level-of-detail and culling algorithms, is even more difficult to manage. As a result, voxel-based terrain representations are not very popular in real-time applications, except for technology demos and experimental work.

5.5.4 Visual Quality

Visual quality is extremely dependent on the methods used for lighting, texturing, shadowing etc. What we mean by visual quality, in this context, is the visual quality of the terrain surface generated from the internal representation of the terrain. The surface geometry of a heightmap is quite simple. Therefore, it is easy to generate the terrain surface geometry of a heightmap without visual artifacts. Level-of-detail management of a heightmap surface geometry is also very easy as the terrain surface is greatly constrained and the connections between vertices are well-defined and simple.

Generating a polygonal surface for a volumetric representation is not an easy task. Relatively simple methods, such as the original marching cubes algorithm, generate visual artifacts in some cases. This can be prevented with the use of better and more complex algorithms, though. Level-of-detail is also problematic with voxel representations because most of the surface extraction methods do not support a sampling grid with different resolutions in different areas. Without level-of-detail it is not plausible to render large and detailed terrains. Even methods that can use some kind of level-of-detail approach with a volumetric surface extraction, such as [35], cannot support smooth level-of-detail transitions, which significantly degrades the visual quality.

The proposed surface extraction method is able to generate a smooth terrain surface for the coarse volumetric representation of the terrain without artifacts. The terrain surface consists of surface patches which are very similar to regular grid heightmap terrain surfaces. It is, therefore, possible to adapt simple level-of-detail approaches for use with the proposed approach. We have also presented a level-of-detail approach for the proposed terrain representation which also supports smooth level-of-detail transitions by geometry morphing.

5.5.5 Content Creation

Ease of content creation is also important for a terrain representation. Content creation can be performed by procedural techniques, manual editing by artists, or a combination of both. A quasi-random terrain data is usually created by procedural techniques, which can be controlled by a set of parameters to some extent, and then fine details are added on top of it by artists.

Successful procedural techniques for heightmap creation are available as it is a relatively simple process. A heightmap is a 2D image and it is possible to create sufficiently realistic terrains in a controllable way using noise functions at several octaves. Manually editing this heightmap is also easy as it can be done with an image editor as well as a specialized 3D heightmap terrain editor. It is easy to create and edit terrains with heightmaps, but it is impossible to create interesting terrain features regardless of how hard you try.

Procedural content creation techniques are also available for volumetric terrains. It is easy to create interesting terrain models with lots of volumetric features that are not very similar to the ones in the real world. It is more difficult to control these techniques to create meaningful terrain features, though, such as mountains, hills, caves and arches as they would appear in reality. Manually editing voxel terrains, on the other hand, is very easy. A single voxel defines whether the volume defined by that voxel is filled with matter or not. It is a simple and straightforward definition making it easy to edit the terrain simply by turning voxels on or off.

The proposed representation makes use of both voxel and heightmap representations. Therefore, it is possible to use procedural techniques for volumetric terrains to create the coarse model of the terrain and procedural techniques for heightmaps to create the terrain surface details. Manual editing is easy as in both voxel and heightmap representations. It can, however, sometimes be unintuitive due to displacement normals of the terrain surface. This is a disadvantage that can be neutralized by using a terrain editor that is specifically designed for editing terrains represented with the proposed terrain representation.

5.5.6 Physics and Interaction

Many applications that use real-time terrain rendering do not use the terrain representation merely for rendering purposes. Other elements of the virtual world, such as creatures and vehicles, can be desired to be in interaction with the terrain model. This requires physics computations on the terrain data, such as collision detection. Furthermore, depending on the type and purpose of application, it is occasionally desired to modify the terrain in runtime according to these interactions.

Voxel terrain models are very easy to edit intuitively by simply turning voxels on and off. Modification of the voxel model may require regeneration of the entire surface, though, since most surface extraction approaches do not support local updates. Simple queries are usually efficient because checking whether a voxel is empty or not merely requires a very fast lookup. The performance of more advanced physics computations depend on the data structure used to represent volumetric data.

Editing heightmaps is also easy but there is less control over the updates as only the height of a vertex can be changed. Local updates are also easy as it only requires changing the height of corresponding vertices and does not require changing vertex connections. Heightmap terrains, therefore, can easily be deformed in real-time but the level of control over deformations is very limited. Not all types of deformations can be applied to a heightmap as the vertices can only be moved in one axis. The performance of physics queries, such as collision detection, is simple since a given position in the space maps one to four samples, at most, on the heightmap. The fact that heightmap representation allows very limited terrain configurations also makes physics computations simpler as there are fewer problematic cases.

The proposed terrain representation is deformable at the surface patch level in real-time. That is, the coarse representation of the terrain cannot be modified in real-time. The deformations applied to surface patches are handled very efficiently and only cause local changes on the terrain surface. Coarse voxel model can be used to answer some queries but it is not very dependable as it does not regard heightmap displacements applied to surface patches. We believe that it is possible to create a voxel representation of the terrain in the proposed approach for the sole purpose of answering physics queries efficiently. In this case, the performance of the proposed approach and other voxel-based approaches become equivalent. The resolution of this voxel representation can be determined according to the desired level of precision in computations. Since this representation is used only for physics computations, it does not require storing of visualization attributes of voxels which makes it possible to store the voxel representation efficiently. This topic, however, is considered outside the scope of this thesis and can be an interesting topic for future research.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

We proposed a new hybrid terrain representation that combines voxel- and heightmap-based approaches. The proposed representation models volumetric terrain features, such as caves, overhangs, arches and vertical cliffs, using a coarse voxel-based approach. The resolution of the terrain surface is then increased further to add surface detail by applying heightmap displacement to the terrain surface. In order to be able to do this, a method for extracting terrain surface from the voxel model is used, where the extracted terrain surface consists of surface patches similar to the blocks of a 2D regular grid heightmap-based approach.

The proposed approach provides a useful trade-off between the simplicity and efficiency of the heightmap-based approaches, and the expressiveness of the voxel-based approaches. Since the terrain surface consists of surface patches that can easily be mapped to 2D planar coordinates, it is possible to use many existing algorithms that are designed to work on regular grid terrain surfaces with minimal changes required for adaptation.

Several visualization techniques, such as lighting, texturing and shadowing, are discussed throughout the thesis that can be used with the proposed terrain representation for high quality real-time terrain rendering. A level-of-detail scheme is presented to allow the real-time rendering of much larger terrains and geometry morphing is used to make smooth transitions between levels-of-detail possible.

6.2 Future Work

The terrain representation proposed in this thesis give birth to a number of interesting research problems. A new terrain representation means a whole new set of algorithms to operate on that representation, possibly some adapted from previous work and some designed from scratch. A number of interesting extensions and improvements that can be built on the proposed approach are listed in this section.

Vector field displacement can be used to apply displacement on surface patches along all directions rather than just the direction of the displacement normal. A similar approach is proposed by [28]. This extension could increase the expressiveness of the representation greatly.

Hardware-accelerated tessellation is one of the newest methods supported by modern GPUs. It can be used to tessellate terrain surface patches so as to increase the terrain surface resolution much higher.

Surface extraction can be improved to handle voxels at different resolutions. The proposed surface extraction algorithm cannot handle voxels at different resolutions. This limits the maximum resolution of the coarse voxel representation.

Collision detection algorithms can be designed to work with the proposed terrain representation. Many types of real-time terrain rendering applications require some form of collision detection, even if a very simple one. This could definitely be a useful extension to the proposed approach. We believe that it is possible to create a voxel representation of the terrain in the proposed approach for the sole purpose of answering physics queries efficiently. The resolution of this voxel representation can be determined according to the desired level of precision in computations. Since this representation is used only for physics computations, it does not require storing of visualization attributes of voxels which makes it possible to store the voxel representation efficiently.

Portals and occlusion culling algorithms can be designed to work with the proposed terrain representation to cull parts of the terrain that are blocked by other parts that are closer to the observer. The use of occlusion culling can significantly increase the rendering speed depending on the terrain model. Portals are usually used for occlusion culling of in-door scenes that are divided to rooms. We believe that it is also applicable to the proposed representation as it can represent caves with similar structure to such scenes.

Terrain synthesis techniques can be developed to convert a high resolution voxel representation to the proposed terrain representation. This could be performed by first lowering the resolution of the given voxel representation depending on a desired resolution or an error metric. This low resolution voxel representation would be the coarse terrain representation in the proposed approach. The displacement values of each surface patch could then be computed to approximate the actual high resolution voxel representation with displacement of surface patch vertices. This can be very useful for content creation as it is much easier and intuitive to create and edit terrains in voxel representation.

Ambient occlusion technique can be adapted for use with the proposed approach for more realistic and better lighting.

Blend maps can be used to apply more realistic texturing to the terrain surface. The current approach to multi-texturing adopted by the proposed approach blends all pixels of the textures to compute the final color. Blend maps define how to blend two textures together in a more realistic way as the blend map is created specifically for each texture.

Parallax mapping can be used to add fine detail at a very high resolution. This could greatly improve the visual quality of the terrain rendering, especially on cave walls, ceilings, cliffs, and rocks.

Bibliography

[1] T. K. Peucker, R. J. Fowler, and J. J. Little, "The Triangulated Irregular Network," in *Proceedings of the ASP Digital Terrain Models (DTM) Symposium*, (St. Louis, Missouri), Oct. 1978.

[2] M. P. Kumler, "An Intensive Comparison of Triangulated Irregular Networks (TINs) and Digital Elevation Models (DEMs)," *Cartographica: The International Journal for Geographic Information and Geovisualization*, vol. 31, pp. 1–99, Oct. 1994.

[3] B. N. Delaunay, "Sur la Sphére Vide," *Izvestia Akademii Nauk SSSR, Otdelenie Matematicheskikh i Estestvennykh Nauk*, vol. 7, pp. 793–800, 1934.

[4] M. Fan, M. Tang, and J. Dong, "A Review of Real-time Terrain Rendering Techniques," in *Proceedings of the 8th International Conference on Computer Supported Cooperative Work in Design*, vol. 1, pp. 685–691, may 2004.

[5] M. Garland and P. S. Heckbert, "Fast Polygonal Approximation of Terrains and Height Fields," Tech. Rep. CMU-CS-95-181, School of Computer Science, Carnegie Mellon University, 1995.

[6] M. H. Gross, R. Gatti, and O. Staadt, "Fast Multiresolution Surface Meshing," in *Proceedings of the 6th IEEE Conference on Visualization (VIS '95)*, pp. 135–142, 446, Oct. 29 - Nov. 3 1995.

[7] D. Daniel Cohen-Or and Y. Levanoni, "Temporal Continuity of Levels of Detail in Delaunay Triangulated Terrain," in *Proceedings of the 7th IEEE Conference on Visualization (VIS '96)*, pp. 37–42, Oct. 27 - Nov. 1 1996.

[8] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. A. Turner, "Real-time, Continuous Level of Detail Rendering of Height Fields,"

in *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '96)*, (New York, NY, USA), pp. 109–118, ACM, 1996.

[9] H. Hoppe, "Progressive Meshes," in *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '96)*, (New York, NY, USA), pp. 99–108, ACM, 1996.

[10] H. Hoppe, "View-Dependent Refinement of Progressive Meshes," in *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '97)*, (New York, NY, USA), pp. 189–198, ACM Press/Addison-Wesley Publishing Co., 1997.

[11] L. Hu, P. V. Sander, and H. Hoppe, "Parallel View-Dependent Refinement of Progressive Meshes," in *Proceedings of the Symposium on Interactive 3D Graphics and Games (I3D '09)*, (New York, NY, USA), pp. 169–176, ACM, 2009.

[12] H. Hoppe, "Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering," in *Proceedings of the IEEE Conference on Visualization (VIS'98)*, (Los Alamitos, CA, USA), pp. 35–42, IEEE Computer Society Press, 1998.

[13] W. Evans, D. Kirkpatrick, and G. Townsend, "Right-Triangulated Irregular Networks," *Algorithmica*, vol. 30, pp. 264–286, 2001. 10.1007/s00453-001-0006-x.

[14] M. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein, "ROAMing Terrain: Real-time Optimally Adapting Meshes," in *Proceedings of the 8th IEEE Conference on Visualization (VIS '97)*, pp. 81–88, Oct. 1997.

[15] M. White, "Real-time Optimally Adapting Meshes: Terrain Visualization in Games," *International Journal of Computer Games Technology*, vol. 2008, Jan. 2008.

[16] T. Ulrich, "Rendering Massive Terrains Using Chunked Level of Detail Control," in *Super-size It! Scaling up to Massive Virtual Worlds, ACM SIGGRAPH '02 Course Notes, No. 35*, (San Antonio, TX, USA), ACM, 2002.

[17] W. H. de Boer, "Fast Terrain Rendering Using Geometrical Mipmapping." flipCode Featured Articles, http://www.flipcode.com/archives/article_geomipmaps.pdf, Oct. 2000.

[18] J. Levenberg, "Fast View-Dependent Level-of-Detail Rendering Using Cached Geometry," in *Proceedings of the IEEE Conference on Visualization (VIS'02)*, pp. 259–265, Nov. 2002.

[19] A. A. Pomeranz, "ROAM Using Surface Triangle Clusters (RUSTiC)," Master's thesis, University of California at Davis, 2000.

[20] S. Röttger, W. Heidrich, and H.-P. Seidel, "Real-Time Generation of Continuous Levels of Detail for Height Fields," in *Proceedings of the Sixth International Conference in Central Europe on Computer Graphics and Visualization (WSCG '98)* (V. Skala, ed.), pp. 315–322, 1998.

[21] C. C. Tanner, C. J. Migdal, and M. T. Jones, "The Clipmap: A Virtual Mipmap," in *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '98)*, (New York, NY, USA), pp. 151–158, ACM, 1998.

[22] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno, "BDAM – Batched Dynamic Adaptive Meshes for High Performance Terrain Visualization," *Computer Graphics Forum*, vol. 22, pp. 505–514, September 2003. Proc. Eurographics 2003 – Second Best Paper Award.

[23] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno, "Planet-Sized Batched Dynamic Adaptive Meshes (P-BDAM)," in *Proceedings of the 14th IEEE Conference on Visualization (VIS'03)*, (Washington, DC, USA), pp. 147–154, IEEE Computer Society, Oct. 19-24 2003.

[24] F. Losasso and H. Hoppe, "Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids," *ACM Transactions on Graphics (TOG) (Proceedings of ACM SIGGRAPH '04)*, vol. 26, no. 3, pp. 769–776, 2004.

[25] A. Asirvatham and H. Hoppe, "Terrain Rendering Using GPU-Based Geometry Clipmaps," in *GPU Gems 2*, pp. 46–53, Addison-Wesley, 2005.

[26] S. Mantler and S. Jeschke, "Interactive Landscape Visualization Using GPU Ray Casting," in *Proceedings of the 4th International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia (GRAPHITE '06)*, (New York, NY, USA), pp. 117–126, ACM, 2006.

[27] A. Kolb and C. Rezk-Salama, "Efficient Empty Space Skipping for Per-Pixel Displacement Mapping," in *Proceedings of International Workshop on Vision, Modeling and Visualization (VMV'05)*, pp. 407–414, 2005.

[28] C. McAnlis, "Halo Wars: The Terrain of Next Gen," in *Game Developers Conference*, March 2009.

[29] D. A. K. McRoberts, "Real-time Rendering of Synthetic Terrain," Master's thesis, University of Johannesburg, July 2011.

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

[31] W. E. Lorensen and 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), pp. 163–169, ACM, 1987.

[32] D. S. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, and S. Worley, *Texturing & Modeling: A Procedural Approach*. San Diego, CA, USA: Elsevier Science, 3 ed., 2003.

[33] S. Forstmann and J. Ohya, "Visualization of Large ISO-Surfaces Based on Nested Clip-Boxes," in *ACM SIGGRAPH 2005 Posters*, (New York, NY, USA), ACM, 2005.

[34] B. Gregorski, M. Duchaineau, P. Lindstrom, V. Pascucci, and K. I. Joy, "Interactive View-Dependent Rendering of Large Iisosurfaces," in *Proceedings of the IEEE Conference on Visualization (VIS'02)*, (Washington, DC, USA), pp. 475–484, IEEE Computer Society, 2002.

[35] E. Lengyel, *Voxel-Based Terrain for Real-Time Virtual Simulations*. PhD thesis, University of California at Davis, 2010.

[36] P. Bézier, *Mathematical and Practical Possibilities of UNISURF*. New York: Academic Press, 1972.

[37] L. Piegl and W. Tiller, *The NURBS Book.* Berlin: Springer-Verlag, 2 ed., 1997.

[38] K. Perlin, "Noise Hardware," in *Real-Time Shading, ACM SIGGRAPH '01 Course Notes, No. 24*, (Los Angeles, CA, USA), ACM, 2001.

[39] S. Gustavson, "Simplex Noise Demystified." `http://webstaff.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf`, March 2005.

[40] Silicon Graphics, Inc., "OpenGL: Open Graphics Library." http://www.opengl.org/, 2012.

[41] M. Ikits and M. Magallon, "GLEW: The OpenGL Extension Wrangler Library." http://glew.sourceforge.net/, 2012.

[42] D. Woods, N. Weber, and M. Dario, "DevIL - A Full Featured Cross-platform Image Library." `http://webstaff.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf`, 2012.

[43] S. Gustavson, "The FreeType Project - A Free, High-Quality, and Portable Font Engine," 2012.

[44] Silicon Graphics, Inc., "GLSL: OpenGL Shading Language." http://www.opengl.org/documentation/glsl/, 2012.

[45] Microsoft Corp., "MS Windows, QueryPerformanceCounter function," 2012.

[46] Silicon Graphics, Inc., "OpenGL SDK, glGetQueryObject function," 2012.