Technische Hochschule
Brandenburg
University of
Applied Sciences

Volumetric Terrain Rendering with WebGL

Raoul van Rüschen

23.12.2016

Master's Thesis
Master of Science
Brandenburg University of Applied Sciences
Department of Informatics and Media

Supervisor 1: Prof. Stefan Kim
Supervisor 2: Prof. Dr. Reiner Creutzburg

Declaration of Independent Work

I herewith declare that I wrote and composed this Master's thesis independently. I did not use any other sources, figures or resources than the ones stated in the bibliography, be they printed sources or sources off the Internet. This includes possible figures or tables. I marked all passages and sentences in my work that were taken from other sources clearly as such and named the exact source.

Furthermore I declare that – to my best knowledge – this work has never before been submitted by me or somebody else at this or any other university.

Date: _____ Signature: _____

Preface

This Master's thesis has partly been carried out in the context of an ERASMUS student exchange programme at the Norwegian University of Science and Technology (NTNU) and partly at the Brandenburg University of Applied Sciences (THB) during the summer semester 2016 and winter semester 2016/2017.

The idea of exploring volumetric terrain emerged from the project "GameLab" which is part of the computer science Master's degree programme at THB.

Readers are advised to have a basic understanding of conventional polygon-based real-time 3D rendering. The presented implementation uses the WebGL rasterisation API which is based on OpenGL/Vulkan (Khronos Group 2016). Knowledge about JavaScript or the OpenGL Shading Language (GLSL) is recommended, but not required as all source code that does appear in this thesis is explained in detail.

Acknowledgement

I would like to thank my parents, my brother and my friend Lucas for supporting me all this time.

Furthermore, I would like to thank my supervisors, Prof. Stefan Kim and Prof. Dr. Reiner Creutzburg for enabling me to conduct my research at the NTNU.

I would also like to thank my project supervisor Simon McCallum for his excellent guidance during my stay at the host university.

R.v.R.

Abstract

Since the introduction of WebGL in 2011, the web browser evolved into a new and promising platform for high-performance 3D games. One of the most common game elements is heightmap-based terrain, but due to the limited expressiveness of this approach, the need for an advanced alternative becomes more and more apparent. Many techniques exist that can convert an implicit surface into an approximated polygonal mesh. However, the actual application of such algorithms in a real-time environment, especially on mobile devices, where render time is of utmost importance has not been investigated sufficiently in the literature yet.

The present work outlines the implementation of a multithreaded volumetric terrain engine using the 3D rendering framework Three.js and sheds light on unexplored areas in the domain of real-time volumetric terrain rendering such as the management of large amounts of volume data and scheduling of volume modifications and surface extractions. The final system uses the Dual Contouring surface extraction technique and maintains discrete volume data in chunks which can be modified on the fly using Constructive Solid Geometry. Furthermore, the performance of the engine is evaluated to determine its suitability for mobile devices.

Seit der Einführung von WebGL in 2011 hat sich der Webbrowser in eine neue und vielversprechende Plattform für 3D-Spiele entwickelt. Eines der verbreitetsten Elemente in Computerspielen ist heightmap-basiertes Terrain. Diese Form von Terrain ist jedoch sehr eingeschränkt hinsichtlich der Modellierungsfreiheit und der Bedarf nach einer fortgeschritteneren Alternative steigt zunehmend. Es existieren viele Methoden mit denen implizite Oberflächen in polygonale Netze überführt werden können. Allerdings wird die Anwendung solcher Algorithmen unter Echtzeit-Bedingungen in der Literatur nur unzureichend behandelt.

Die vorliegende Arbeit beschreibt die Implementierung eines volumetrischen Terrainsystems unter Anwendung des 3D-Frameworks Three.js und beleuchtet unerforschte Bereiche des volumetrischen Terrainrenderings wie etwa das Verwalten von großen Mengen von Volumendaten und das Koordinieren von Volumenmodifizierungen und Oberflächenextraktionen. Das System verwendet den Dual Contouring Algorithmus zur Erstellung von Terrainoberflächen und organisiert Volumendaten in Blöcken, welche zu jeder Zeit frei bearbeitet werden können. Des Weiteren wird die Performanz des Systems genauer untersucht, um festzustellen, ob es sich für den Einsatz auf mobilen Geräten eignet.

Contents

Figures

Tables

Program Code

1 Introduction

1.1 Motivation

WebGL is a young and exciting technology used for the creation of web browser games that are almost on par with desktop games in terms of performance. The execution speed of JavaScript is currently the limiting factor in that regard. According to Eich (2015), the creator of the language, future versions of JavaScript will address this issue by supporting more low-level programming capabilities such as typed objects, parallel arrays and SIMD instructions. New updates to the language are planned to be considerably smaller and they will be released faster to allow browser vendors to implement the new features quicker.

Nevertheless, developing games with WebGL is already a feasible undertaking today, because the major JavaScript engines are heavily optimised. WebGL-based games run in the browser and can therefore run on mobile devices like smartphones and tablets, too. The fact that there are more mobile users than desktop users gives WebGL a big economic advantage. Furthermore, the Internet plays a significant role in today's life and allows easy and direct software deployment. This makes WebGL an even more appealing platform to work with.

A central part of many games is terrain and traditional approaches use 2D height data to elevate the vertices of a regularly subdivided plane mesh. As users begin to expect more of new games, the need for more sophisticated solutions becomes apparent. Modern terrain implementations use advanced algorithms to construct a surface based on volume data. Due to a lack of volumetric terrain engines for WebGL, the implementation of such a system might prove to be a valuable contribution to the open-source game development ecosystem.

Heightmap-based terrain is rather limited because each vertex in the terrain grid can only be moved up or down. It cannot be used to replicate caves, overhangs, bows and other interesting natural features. Volumetric terrain can replace the heightmap approach entirely and it provides much more freedom, but it inherently requires more memory as it operates on 3D data instead of 2D textures. Level of detail algorithms are also more complicated than the heightmap variants. It's important to note that there are no compute, geometry and tessellation shaders in WebGL, so a volumetric terrain must be implemented by mainly relying on the CPU.

NASA (2016) provides heightmaps of the visible earth, but these heightmaps are missing valuable information about the rich features of real terrain. A volumetric terrain engine combined with a basic volume editor could be used to build

upon these heightmaps to improve the data from NASA or from other sources. It could also be used for games, of course.

1.2 Objectives

The fact that there are currently no open-source terrain engines for WebGL, much less terrain editors, neither volumetric nor heightmap-based, could hint to the conclusion that such an implementation might not be worth the development effort despite the advantages mentioned earlier. Thus, the following central questions arise:

1. Is the performance of a volumetric terrain solution feasible?
2. Can such an implementation be used on mobile devices?

The main goal of this thesis is to implement a volumetric terrain engine with JavaScript and WebGL. The challenge in creating such an engine is that there are still unexplored aspects in the domain of real-time volumetric terrain rendering such as the management of large amounts of volume data. Additionally, the use of JavaScript requires a profound awareness of performance pitfalls as well as platform limitations. In the context of the aforementioned questions, the implementation details and design choices of the final system will be presented and discussed.

1.3 Structure

The following chapter outlines related work in the field of volume polygonisation and presents existing isosurface extraction techniques. Furthermore, a recap of the traditional heightmap-based terrain approach is given and a selection of games that already use volumetric terrain is presented. In the subsequent chapter an overview of the development environment is given, after which follows a technical description of the implemented volumetric terrain engine and its architecture. The thesis ends with a discussion about possible future work and the results of the project.

2 Related Work

2.1 Heightmap Terrain

The most prevalent approach to terrain rendering in games is to elevate the vertices of a regularly subdivided plane with height data stored in a greyscale texture. Such textures are typically called heightmaps or heightfields and efficiently store 2D height data which is kept separately from the terrain mesh. While it's possible to permanently apply the height values to the vertices, most terrain implementations read the height information per vertex during each render iteration to dynamically place the vertices at the correct position. This flexibility allows for advanced rendering techniques. For example, the subdivision of the terrain mesh could freely be performed in an adaptive way with a tessellation shader. However, WebGL supports neither geometry nor tessellation shaders and a JavaScript terrain implementation would have to rely on the Central Processing Unit (CPU). The left image in Figure 1 shows an example of a heightmap terrain created with Unity3D. Unfortunately, certain shortcomings of heightmap terrain limit its expressiveness. As already mentioned earlier, overhangs, cliffs and caves cannot easily be modelled with this approach. The right image in Figure 1 depicts a common workaround for caves where an additional model has been integrated in the terrain.

Since terrain oftentimes stretches far into the distance of a scene, a Level of Detail (LOD) mechanism must be deployed to avoid unnecessarily high concentrations of polygons in faraway places. The goal of LOD algorithms is to render distant geometry with a reduced amount of polygons while maintaining significant geometrical features. There are many ways to implement a LOD scheme for heightmap terrain. One example is the use of concentric rings consisting of multiple adjacent grids. Each grid has the same amount of polygons but is scaled depending on how far away it is from the centre, resulting in a lower effective

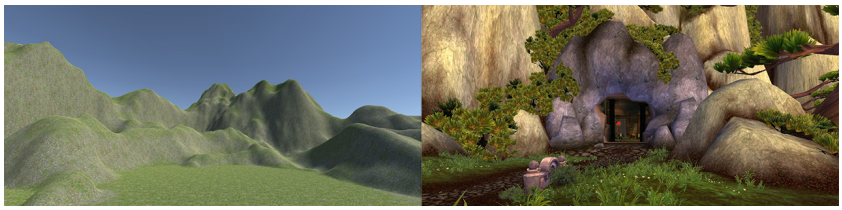

Figure 1: A simple heightmap-based terrain made with Unity3D (left) and a cave opening from the game World of Warcraft.

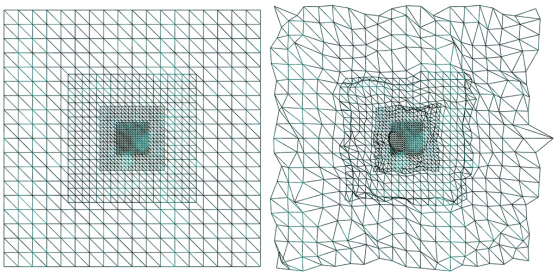

Figure 2: A top-down view of a basic LOD grid with 4 levels of detail. The right image shows the mesh with the heightmap applied to it.

resolution for distant rings. Figure 2 shows how such a setup could look like with 4 levels of detail. The camera is positioned in the middle of the grid to have the finest details close to the observer and the grid moves with the camera. Consequently, camera movements directly influence the world positions of the grid vertices which are used as heightmap sampling coordinates. As a result, the grid glides over the static height data and the terrain surface is updated automatically.

2.2 Volumetric Terrain in Games

Volumetric terrain is no novelty in the field of video games. The game Minecraft published by Mojang (2011), for example, uses a special form of volumetric terrain which simply consists of uniformly sized blocks that are stacked on top of each other. This allows for construction and destruction of the terrain in a block-by-block manner. Dungeon Keeper 2 is another older title published by Bullfrog Productions (1999) that lets players dig through the earth to create elaborate underground dungeons and tunnel systems. The destructible environment which can be seen in Figure 3 is organised in a blocky fashion similar to the terrain blocks in Minecraft. More recent games like Subnautica published by Unknown Worlds Entertainment (2016) and No Man's Sky published by Hello Games (2016) utilise advanced algorithms to extract a realistic terrain surface from volume data and also allow the player to actively influence the shape of the terrain in real-time. The left image in Figure 4 shows the underwater terrain from Subnautica and demonstrates that overhangs and tunnels can be modelled naturally. The image on the left shows a landscape from No Man's Sky that has smooth and sharp geometrical features.

While there are quite many volumetric terrain libraries, most of them are discontinued projects. The most notable active project is the Voxel Farm Engine which is maintained by Cepero (2016*b*). It's available for Unity3D and the Unreal Engine and licenses for this engine are currently offered for sale on the respective asset stores.

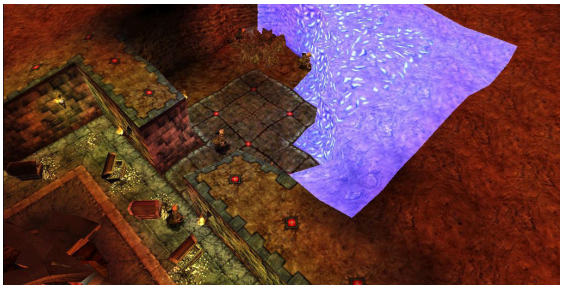

Figure 3: Dungeon expansion through manual digging in the game Dungeon Keeper 2.

Figure 4: Underwater terrain from the game Subnautica (left) and terrain with interesting geometrical features from the game No Man's Sky.

2.3 Signed Distance Functions

According to Osher & Fedkiw (2006), a Signed Distance Function (SDF) belongs to a subset of implicit surfaces and describes the signed Euclidean distance to the surface of a volume, effectively describing its density at every point in 3D space. It can be defined as $f\colon \mathbb{R}^3 \to \mathbb{R}$ and yields negative values for points that lie inside the volume and positive values for points outside. The value is zero at the exact boundary of the volume. These restrictions ensure that unpredictable results are avoided and the functions behave linearly.

2.4 Constructive Solid Geometry

Constructive Solid Geometry (CSG) is a design methodology for representing solids that is based on the mathematical set notation. It "offers simple, precise, and concise ways for humans and automata to define specific solid objects" (Requicha & Voelcker 1977, p. 10). In the context of implicit surfaces, the methodology is used to combine SDFs into complex descriptions of volumes. Figure 5 shows the effect of the three Boolean CSG operations Union $(\cup)$, Difference $(\setminus)$ and Intersection $(\cap)$.

"CSG schemes have a finite and usually small repertoire of compact solid primitives" (Requicha & Voelcker 1977, p. 12). As an example, the Persistence of Vision Raytracer only offers the following primitive solids: box, cone, cylinder, plane and torus. However, with these primitives alone it's possible to create

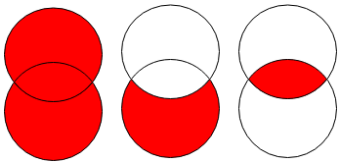

Figure 5: The effect of CSG operations from left to right: Union, Difference and Intersection.

highly complex solids using CSG. Another project that uses CSG is OpenCSG. This library follows an image-based rendering approach instead of ray tracing and relies on the depth and stencil buffer of the graphics hardware to render solids.

It's worth mentioning that the design of the terrain engine's CSG interface has been inspired by the JavaScript library csg.js which implements CSG operations on meshes using Binary Space Partitioning (BSP) trees. The approach that this library takes is particularly interesting as it doesn't rely on volume data and instead combines polygonal models. This, however, is an approach that is not going to be investigated in this thesis.

2.5 Isosurface Extraction Techniques

An isosurface represents the contour of an implicit surface $f(x, y, z) = c$ where c is a constant isovalue that denotes the boundaries of the SDF. Although it's possible to render implicit surfaces with a ray tracing approach, the performance penalty would be too high, especially for mobile devices. Since 3D hardware is optimised for conventional polygon-based rendering, the implicit surface must be converted into an explicit polygonal mesh that can be visualised efficiently.

Various isosurface contouring techniques exist that perform the conversion in different ways. One of the oldest and most prominent techniques is the Marching Cubes (MC) algorithm published by Lorensen & Cline (1987). It was originally created for medical visualisation purposes and translates the continuous values of an SDF into a discrete grid of uniformly distributed material indices. The amount of grid points and their proximity directly defines the resolution of the resulting polygonal mesh. This 3D grid is subdivided into voxel cells that use the grid points as their corner vertices. As the name of the technique suggests, MC marches over these cubic cells and evaluates the SDF at the world position of every cell corner. Depending on the density returned by the SDF, the grid point will either be set to air or to solid material. The information of all eight corners can collectively be stored in a single byte and the value of that byte as a whole is used as an identifier for the case at hand. With a finite number of possible material configurations per cell, each case can be mapped to a concrete triangle setup. To match the volume's surface as closely as possible, the vertices of the generated triangles are moved to the intersection points of their respective edges.

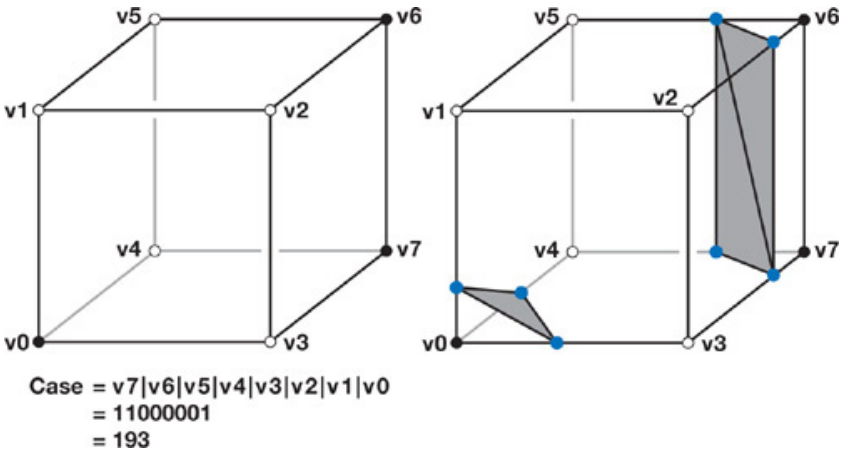

Figure 6: A Marching Cubes voxel cell. The materials at the eight corners are known; the right picture shows the generated polygons. Source: Nguyen (2007)

In a final step, all polygons are tied together. Figure 6 shows a voxel cell with an exemplary material configuration and the generated triangles next to it. The MC extraction method is fast but not without flaws: it often produces degenerate triangles and can't preserve sharp features. There are also a few ambiguous cases that require special treatment to avoid holes in the surface and the algorithm can't easily be extended with LOD functionality.

A solution to the LOD problem was presented by Lengyel (2010) in the form of the Transvoxel algorithm which introduces another set of polygon configurations for transition cells to connect meshes of different LODs.

The Extended Marching Cubes (EMC) algorithm presented by Kobbelt et al. (2001) introduces a mechanism for sharp feature preservation with Quadratic Error Functions (QEFs). For each voxel cell edge that exhibits a material change, the intersection with the implicit surface is approximated. Additionally, the normal vector of the surface is calculated at the identified point and then inspected to determine if a sharp feature exists in the cell. Together, the normals and intersection points describe planes that serve as input for a system of linear equations. Solving the system yields the intersection point of the planes which is, in fact, the sought feature point. However, an implicit surface might intersect with a voxel cell in such a way that there are less than three planes which causes the linear system to become underdetermined. In order to solve such a system, a QEF is used which finds a point inside the voxel that minimises the sum of the squares of the distances to the planes that are defined by the intersection points and normals. In case a sharp feature was detected, EMC solves the QEF to obtain a least squares solution, creates a triangle-fan at the identified feature point and connects it with the edge intersection points. Apart from that, the algorithm operates like MC.

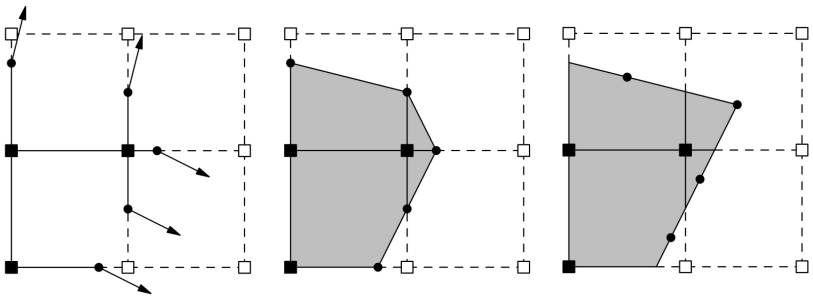

Figure 7: A comparison of triangle generation with Marching Cubes (centre) and Dual Contouring (right) in 2D. MC relies on the material indices and surface intersection points while DC also considers the intersection normals. Source: Ju et al. (2002)

Following the idea of preserving distinct details of the volume's surface, Ju et al. (2002) published the straight-forward contouring technique called Dual Contouring (DC) which takes after the approach of Surface Nets (SN) presented by Gibson (1999) and, like EMC, relies on feature points obtained with QEFs. However, unlike the previous methods, DC doesn't try to map voxel cell material configurations to certain triangle setups. Instead, it creates a single vertex per cell and connects it with vertices of neighbouring cells. Furthermore, the method uses an octree data structure to organise and traverse the voxel cells. A side-effect of this approach is that the algorithm supports LOD without any additional effort, because it allows voxels to be of any size.

Figure 7 shows a comparison of how MC and DC generate polygons and highlights the superiority of the latter. The leftmost image shows a section of a 2D volume grid consisting of equispaced material indices, surface intersection points at edges with a material change and surface normals originating from them. Such edge data is commonly referred to as Hermite data. MC can only approximate the surface roughly as shown in the central image while DC manages to preserve the sharp feature of the surface. In a later publication, Ju & Udeshi (2006) stated that "the surface produced by Dual Contouring is rarely intersection-free" and proposed a hybrid of MC and DC that uses triangle fans to produce intersection-free meshes at the cost of performance and increased complexity.

Another issue that DC shares with MC is that they both may produce non-manifold meshes. A topologically manifold mesh doesn't have holes and completely encloses a volume. In essence, every edge needs to be adjacent to two faces. Figure 8 shows an example of a non-manifold mesh which has edges that don't meet this requirement. Schaefer et al. (2007) addressed the issue by allowing multiple vertices per voxel cell and implementing a basic criterion for vertex clustering which, however, results in an increase in computational complexity. In an earlier publication, Schaefer & Warren (2004) also presented the Dual Marching Cubes (DMC) algorithm which introduced the concept of a dual grid for the

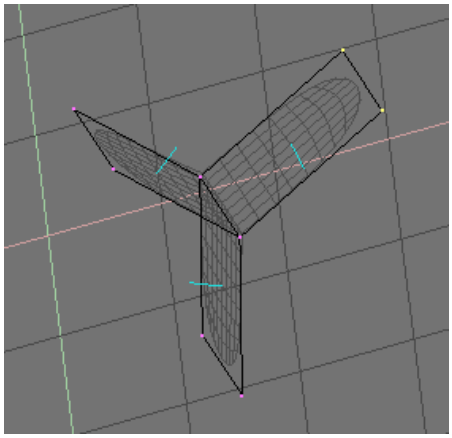

Figure 8: An example of a non-manifold mesh. Source: Stichting Blender Foundation (2005)

preservation of sharp features using MC.

Cubical Marching Squares (CMS) is another unique contouring method presented by Ho et al. (2005). It's based on MC, but works differently in that it unfolds the voxel cells and processes the cell faces with the simpler 2D Marching Squares (MS) algorithm to form lines. Hermite data is used to preserve sharp features and the algorithm guarantees topological consistency by dividing faces that have ambiguous edges. The faces are folded back into cubes which are then used to build the mesh.

2.6 Summary

MC, EMC, DMC and CMS are techniques that create independent triangles inside of voxel cells and can be classified as primal methods. These methods operate on isolated cells and are therefore perfectly parallel. On the other hand, DC and its derivatives are dual methods that create a single vertex per cell and connect it with vertices from neighbouring cells. This inter-cell dependency makes it harder to accelerate them with advanced parallelisation strategies such as General-purpose computing on Graphics Processing Units (GPGPU) because individual cells can't be processed in isolation.

Although CMS claims to solve all the problems that the other methods have, the resources regarding its implementation details are scarce. Rassovsky (2014) provides a loose implementation of the algorithm that proves the feasibility of the technique but lacks certain key features that are crucial for terrain rendering. DC preserves sharp features, supports LOD and uses the same data as CMS. Therefore, the original DC technique will be used in this thesis as it should be interchangeable with a CMS implementation at a later date unless, of course, an even better approach has been discovered by then.

2.7 Unexplored Areas

The presented contouring techniques don't touch upon the performance implications of primarily relying on SDFs and they assume that the underlying implicit surface is available at all times, however complex it may be. SDFs can be logically combined to form arbitrarily complex volumes, but with every added SDF, the density sampling performance decreases. In a system where performance is of utmost importance, it would be wasteful to always sample the density function whenever a new surface extraction takes place.

There are a few mentionable web blogs that provide inspiring information on the topic: Cepero (2016*a*) provides many short articles about the Voxel Farm Engine and Gildea (2014*a*) talks about the implementation of DC while Trettner (2013) explains volume generation in more detail. Furthermore, Lysenko (2012) provides basic JavaScript implementations of MC, Marching Tetrahedra (MT) and SN with a comparison of the three techniques in terms of performance and polygon counts. Unfortunately, the management of volume data for extensive game worlds remains largely undocumented and implementation details are scarce. This thesis sheds light on the application of contouring methods in a real-time environment where the volume frequently changes through user interaction. Considering the existence of recent desktop games such as Subnautica and No Man's Sky which already utilise volumetric terrain solutions, this thesis provides an implementation for WebGL.

3 Development Environment

In the previous chapter, the essential volume contouring methods have been reviewed and the need for a clarification of the specifics about real-time isosurface extraction has been highlighted.

Before the implementation is discussed, the development environment will be presented with a focus on the most important tools and their roles in this project. The chosen tools are the JavaScript runtime Node.js, the Node Package Manager (NPM), the task runner Grunt paired with the code bundler Rollup and the basic text editor Notepad++. The WebGL abstraction library Three.js is used for rendering the scenes. Furthermore, YUIDoc is used to generate the documentation for all code written in this project from documentation comments.

3.1 Node.js

JavaScript, also known as ECMAScript, has come a long way from being regarded as an irrelevant toy language. Now it's a highly optimised programming language that can even be used to implement WebGL games. In fact, it's the only option available. JavaScript used to only live inside the browser which made it hard to debug and maintain big projects. This problem was indirectly solved with the release of Node.js in 2009, a platform independent JavaScript runtime that incorporates Google Chrome's V8 engine which debuted in 2008. Node.js makes it possible to develop server systems with JavaScript. This, however, is only one of many possible scopes of application - it's more appropriate to see Node.js as a general-purpose tool for the creation of any kind of JavaScript-based software systems. A crucial aspect of Node.js is that it pushed the concept of modular software architecture by incorporating the CommonJS module standard that effectively embodies a workaround for JavaScript's lack of native package management facilities. This approach is still widely used in complex software systems, but is now obsolete because such a system was included in ECMAScript 6th Edition (ES2015).

3.2 NPM

The Node Package Manager is a tool that ships with Node.js. It's used for publishing JavaScript modules via the NPM registry and managing project dependencies. Dependencies are defined in a file named "package.json". This package configuration mainly holds meta information like the name of the module and the version number. Furthermore, scripts can be defined in this package configuration to automate specific tasks. While it's possible to rely entirely on this

very mechanism as a replacement for a build tool like Grunt, it's execution speed decreases quickly as more custom tasks are added. In this project, the scripts section of the package configuration only holds a start command for Grunt.

3.3 Grunt

Software development cycles always include a number of code processing steps from basic error checking to the compilation of the final product. JavaScript is no exception to this - all the components of a software system need to be checked for errors and should ultimately be combined into a single bundle file for optimal deployment on the Internet. The generated bundle can then, for example, be minified to reduce the file size. Grunt is a build tool like Gradle or Maven which enables developers to write concise project configurations that define project-specific task chains. Grunt isn't limited to JavaScript processing; it can also be used for other automated tasks like image compression or deployment to cloud computing web services. Alternatives to Grunt are, for example, Gulp and Broccoli. It's a matter of personal preference which one to chose, since they mainly differ in their respective configurations.

3.4 Rollup

As mentioned in Section 3.1, the ES2015 specification defines a native module import/export system that is static by design and therefore allows code analysis and optimisation of dependencies at compile time. Such dependencies can be internal as well as external software components that are usually called modules or packages regardless of their complexity. The now obsolete CommonJS module system loads software components during runtime which makes it impossible to determine their actual relevance before the whole program is executed. While this old approach made it possible to cleanly maintain a JavaScript project consisting of many separate files, it was only capable of importing entire modules. The new native system supports selective imports of parts of modules. Rollup takes advantage of the new static module system and accumulates only those parts of a project into a final bundle, that are actually being used in a meaningful way. This mechanism is commonly referred to as tree-shaking and generates truly minimal software products. It should be noted that Rollup is a fairly new and future-oriented tool that is still in an early development phase. ES2015 is, at the time of writing, still not fully implemented in the major browsers, but Rollup makes it possible to already write software with the new module system syntax. In addition to ES2015 bundles, Rollup can also construct traditional bundles from an ES2015 code base for backward compatibility.

3.5 WebGL

WebGL is the rasterisation Application Programming Interface (API) of the web and it allows developers to create animations inside the browser that run on

Table 1: A comparison of WebGL rendering engines

	ES2015	Documentation	Hello World
Three	yes	good	13 min
Babylon	yes	good	19 min
Scene	no	poor	14 min
PlayCanvas	no	good	9 min
Goo Create	no	good	9 min

the Graphics Processing Unit (GPU). The initial version of WebGL is based on OpenGL for Embedded Systems (OpenGL ES) 2.0 and supports a subset of the Open Graphics Library (OpenGL) API. As such, it has limited texture functionality support, no compute shaders, geometry shaders or tessellation shaders and many important features are only available in the form of extensions that need to be enabled explicitly. For example, depth textures are not enabled by default and support for texture sampling inside of vertex shaders is not guaranteed.

The upcoming second version of WebGL will be based on OpenGL ES 3.0 and introduces support for texture arrays which, according to Tavares (2016), allows access to hundreds of separate textures at the cost of a single sampler unit. This is especially useful for shaders that need to switch frequently between many different textures. At the present time, the only way to sample a wide range of different textures depending on a calculated index or offset is to use a carefully crafted texture atlas that combines all the required textures in one.

Future releases of WebGL are likely to incorporate features from Vulkan, the new generation graphics and compute API, but due to security concerns regarding its low level programming capabilities and extensive control over the hardware it is unlikely that Vulkan will replace WebGL.

In this project, WebGL 1.0 will be used as it is the current standard in all major web browsers.

3.6 Rendering Engine

Using the WebGL API directly to setup and render scenes requires a lot of code that can be generalised and reused. Abstraction libraries and frameworks such as Scene.js aim to simplify the management and rendering of complex 3D scenes. Five popular WebGL frameworks have been evaluated to find an appropriate engine for this project. All engines shown in Table 1 are free to use, easily extensible and support 3D rendering. None of them feature a terrain system. The documentation of Scene.js only consists of examples and tutorials whereas the other engines provide exhaustive documentations. Creating a basic scene with PlayCanvas and Goo Create can be achieved quickly as both of those engines provide a complete visual scene editor. However, Three.js and Babylon.js are

the only engines that already use ES2015 language elements and a scene editor doesn't necessarily provide an advantage when the main focus lies on programming. Three.js has been chosen for this project, although Babylon.js would've been an equally good choice. Consequently, the developed terrain engine will ultimately be designed to conform to the API of Three.js, although most of the internal software components will still be largely independent of the chosen rendering framework.

3.7 Shaders

The code of the terrain engine is primarily written in JavaScript, but it also includes a tri-planar texture mapping shader which is described in Section 4.13.

WebGL shaders are written in the OpenGL Shading Language. A shader program consists of one vertex shader and one fragment shader, each usually residing in a separate file, namely one with the extension ".vert" and the other with the extension ".frag". It's possible to keep the two in a single file with the more generic extension ".glsl", but having them in separate files improves maintainability. In addition to the pure shader code, a definition of dynamic uniforms and static macros is required. These variables act as user-defined parameters for the shader program. Three.js uses the concept of materials to wrap the shader code, uniforms and macros. Materials also specify whether the shader supports features like fog so that the renderer can prepare the material accordingly. Furthermore, the vertex shader can pass values to the fragment shader via varying variables. Geometry attributes such as vertex positions and normals, texture coordinates and vertex colours are only fed into the vertex shader.

To summarise the rendering pipeline, the vertex shader calculates the actual position of each input vertex. Then the vertices are used to form primitives which are usually triangles. In the subsequent rasterisation step, these primitives are used to determine which pixels need to be drawn. Finally, the fragment shader calculates the output colour of the identified pixels and the 2D result is written into a frame buffer which is typically the screen.

Shader code is just text and it must at some point be made available to a JavaScript program in order to compile it via the WebGL API. In the case of Three.js this means that the shader code must simply be assigned to a custom shader material in the form of a single string. Three.js then takes care of the compilation. There are at least four different ways to achieve this:

1. Keep the shader code as a string inside the custom material at all times.
2. Read the text content of an HTML script tag containing the shader code through the DOM tree at runtime. The shaders must be embedded into an HTML document.
3. Request the external shader code with Ajax at runtime.
4. Read an external file at compile time and integrate it into the custom ma-

terial to include it in the final bundle.

For the sake of reusability, Three.js currently keeps many small shader code snippets in separate files which are converted into JavaScript strings at compile time. These snippets are then concatenated with additional static shader code strings inside a class called “ShaderLib” to construct the final shaders for the built-in materials. Hence, Three.js uses a mix of the first and fourth approach described above with the aim to reduce code repetition. As a result, the source code of the shaders is rather hard to read and modifications to the snippets affect many materials at once due to strong coupling. However, these snippets can be included in other custom shader materials which allows to build upon the built-in materials. Three.js was designed with ease of use as its main goal and extending built-in materials with new functionality is, in fact, not as easy as it sounds as there is a lot of special treatment in place for these materials which won’t automatically be applied to custom materials.

3.8 Shader Code Inlining

This project follows the fourth approach which is described in Section 3.7 by mainly relying on inlining. Moreover, custom shader materials are defined by using the ES2015 import syntax to read vertex and fragment shader code. A Rollup plug-in is used to inline the text file imports during the bundling process.

```
import fragment from "./glsl/shader.frag";
import vertex from "./glsl/shader.vert";
```

Listing 1: Shader code import.

The code shown in Listing 1 uses the native ES2015 import statement to include custom files. Although the syntax is valid JavaScript code, the referenced files need to be inlined because they aren’t JavaScript modules.

3.9 Web Workers

JavaScript is a single-threaded language which means that there is always only one function being executed at any given point in time. Multithreading capabilities haven’t been part of the language until the fifth version of HyperText Markup Language (HTML) was released in 2014 and computationally expensive tasks became more common in web applications. Today, it’s possible to spawn heavyweight threads in JavaScript via the Web Worker API which operate in isolation but can communicate with the main thread over message ports. Well-known

multithreading challenges like data synchronisation between threads are non-existent due to the fact that Web Workers can only communicate with the main thread through a strict, asynchronous message interface. According to Boesch (2016), Web Workers are supported by all browsers since 2014 and can be used on mobile devices.

It's worth noting that ES2015 introduced generator functions that can be paused in the middle of execution and resumed at a later time. Generators can be regarded as lightweight threads and are useful in various situations but they run on the main thread and can block the program flow quite easily, resulting in poor application responsiveness. Since the terrain engine needs to frequently run tasks that are computationally expensive and relatively long running, the use of generators is not an option and the tasks must truly be executed in parallel to the main thread.

Similar to shader code, the worker program needs to be started programmatically by the terrain engine and since the worker code needs to be loaded dynamically, the question arises how to integrate it into the final system. Since the worker is an independent JavaScript program, it needs to be bundled just like the main engine program. There are at least two possible ways to maintain the worker program bundle:

1. Keep the worker program in a separate file.
2. Inline the worker program at compile time to include it in the final bundle.

It is preferable to produce a single JavaScript file which can then be deployed to client web browsers efficiently. For that reason, the second approach is taken and the worker bundle is integrated into the terrain engine as plain text. The engine can then create a Binary large object (Blob) from this string during runtime and spawn worker threads with it.

Listing 2 shows how an internal data Uniform Resource Locator (URL) can be created for the worker program Blob which can then be specified during every worker instantiation. The drawback of inlining the worker code is that the bundle process of the worker needs to be completed before the engine can be built.

```
import worker from "./worker.tmp";

const workerURL = URL.createObjectURL(new Blob([worker], {
    type: "text/javascript"
}));

function spawnWorker() {

    return new Worker(workerURL);

}
```

Listing 2: Worker program import.

4 Implementation

In this chapter, the implementation of the terrain system is presented starting with an overview of the engine. After a description of the spatial partitioning strategy follows a definition of volume chunks and how volume data is generated, maintained and modified. The DC algorithm is presented in more detail and a LOD system is sketched which also touches upon the drawbacks of using volume chunks.

4.1 Engine Overview

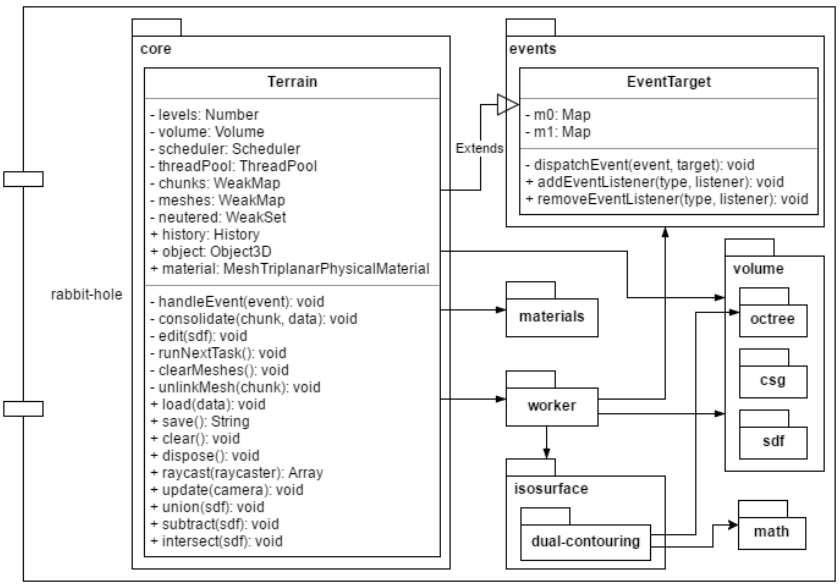

Figure 9: An overview of the engine components.

An engine is a system that tackles a specific group of closely related problems. In this case, the focus lies on problems that deal with volumetric terrain. An engine is also not much different from a framework or a library in that it provides a collection of classes that can be used to solve said problems. Frameworks and libraries often provide more general solutions and inherently require more programming effort to create software that is tailored towards more specific problems. The terrain engine that has been created in the context of this thesis incorporates multiple internal and external software libraries that take care of volume management and the generation of terrain geometry while also

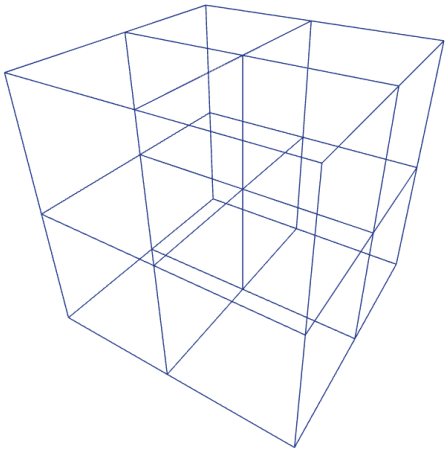

Figure 10: An example of an octree that has been subdivided once.

managing task scheduling and LOD calculations internally. The system operates independently and doesn't require the user to build software on top of it. Instead, it can simply be started and communicated with through a slim API.

Figure 9 shows an overview of the engine's architecture. The volume package contains classes that implement volume management and modification facilities and the isosurface package contains the DC surface extraction algorithm. Parts of this algorithm that deal with the construction of a special voxel octree have been moved into the volume package for a clearer separation of responsibilities. Math components from the original DC implementation provided by Gildea (2014*b*) have also been reworked and cleaned up. At the core of the engine lies the Terrain class which offers a selection of methods to the user. The goal during the design of the engine's API was simplicity and ease of use. Thus, only a few powerful methods are revealed to the user which provide the following essential functionality:

- Saving and loading of volume data.
- Terrain mesh picking through fast octree raycasting.
- On demand volume data construction and destruction.

In order to allow the terrain to be rendered with Three.js, all the generated terrain geometry is grouped in a single Object3D instance which can be added to any 3D scene created with Three.js. Additionally, the terrain implements the event target interface and dispatches two events for every modification and extraction task: one when the task starts and one when it has been completed.

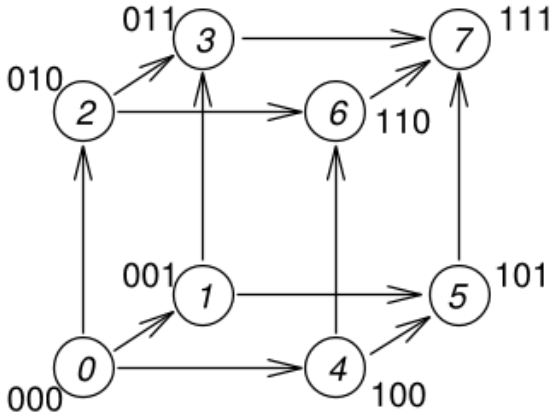

Figure 11: The depicted octant layout is crucial for positional assumptions during raycasting.

4.2 Space Partitioning

An octree is the 3D equivalent of a quadtree and it can be used to subdivide space in a hierarchical manner. Apart from accelerating spatial searches like camera frustum culling and raycasting, the octree data structure is a fundamental component of the DC algorithm. Figure 10 shows an octree after its initial subdivision into eight octants. The octree's root octant contains these eight smaller octants. Each octant is an Axis-Aligned Bounding Box (AABB) that is exactly half the size of its parent node. A search for data in 3D space can be limited to a subset of all octants with a simple intersection test. This step is repeated until a collection of leaf octants has been found which contain the sought data. Octant subdivision is controlled by user-defined criteria like a maximum depth, an upper limit for data entries per octant, a minimum octant size or other more complex conditions.

Due to the lack of a robust general-purpose octree module for JavaScript, an external stand-alone module has been created that provides the basic data structures and implements additional features that are beneficial for the terrain engine. All octrees used in this project are sparse which means that they may contain empty octants. Octants that aren't empty can either have children themselves or they can be leaf nodes that contain data. The alternative to a sparse octree is a complete octree which creates all possible octants down to a fixed tree depth regardless of whether they will actually be visited or populated with data. Complete octrees are useful for scenes with evenly distributed data where they require less memory than the sparse variant. "A full Octree of depth $D = 10$ consists of $N_T = 1227133513(1.2billion)$ nodes which consume around 9.14 GiB of memory" (Geier 2014*a*). A sparse octree is best suited for the volumetric terrain implementation since common game scenes will rarely be fully populated with volume data. In other words, a lot of space will typically remain empty.

Apart from deciding whether to build sparse or complete octrees, it's also

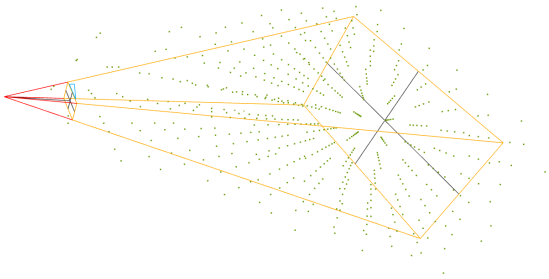

Figure 12: Octree culling with a camera frustum. The depicted dots represent the vertices of the octants that intersect with the frustum.

necessary to chose an internal representation for the octree nodes and how they are stored. There are two fundamentally different kinds of octrees to chose from: traditional pointer-based octrees and linear octrees. This project uses pointer-based octrees in which every octant keeps up to eight pointers to reference its respective children. Pointer-based octrees are very intuitive and allow easy and flexible octree modifications. The concept of linear octrees stems from linear quadtrees that were first proposed by Gargantini (1982) and they store all of their octants in a hash map. Individual octants can be identified by calculating a so called locational code. Linear octrees require less memory since no overhead needs to be stored. However, "Creating and deleting nodes at the top of hashed Octrees is very costly, because the locational code of all nodes below the new root node gets 3 bits longer and must be updated. Consequently, the hash map must be updated as well" (Geier 2014*b*).

The octree implementation that is used in the terrain engine incorporates an efficient raycasting technique that was originally proposed by Revelles et al. (2000). The algorithm capitalises on positional assumptions and therefore requires the octants to adhere to a common layout. This layout dictates the order of the eight children of an octant based on their relative position. Figure 11 shows the expected octant positions and maps each position to a unique binary identifier. The technique is a top-down parametric method that recursively analyses ray parameters to find the entry and exit planes of the octants that intersect with the ray. The advantage of using an octree for raycasting becomes apparent when comparing it to the brute force approach. Raycasting a point cloud consisting of 1048576 points with a naive approach may take upwards of 65 milliseconds while the octree approach culls a large amount of points and takes less than half a millisecond with a tree depth of $D = 5$. Moreover, the performance of octree raycasting scales well with larger amounts of data.

Another important feature of the octree implementation is culling which uses intersection tests to find octants that lie in a specific region. Such a region can be described mathematically in the form of a sphere, an AABB, a frustum or

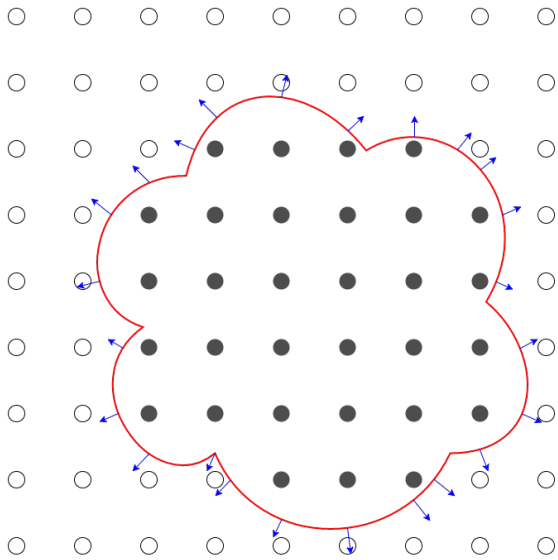

Figure 13: A 2D example of Hermite data with a chunk resolution of 8. Black dots are solid material indices. Blue arrows represent the surface intersection normals. The implicit surface is shown in red.

any other shape. Note that a point, a sphere with a radius of zero, can also be used for culling. Figure 12 shows an example of octree culling using a square frustum. The identified octants' vertices are visualised as green dots. Using a sphere or an AABB instead of a frustum for octree culling is less accurate but also faster because the intersection tests are simpler. The terrain engine relies on a sparse octree structure to organise volume data and it currently uses a camera view frustum to query lists of volume data chunks via culling.

4.3 Spatial Sampling of Density Data

Isosurface extraction methods produce a discrete approximation of a continuous SDF by superimposing a three-dimensional grid with a fixed amount of equispaced material indices. This material grid is essentially a 3D array of integers. Depending on whether the material indices lie inside or outside of the volume they are either set to air which is represented by a value of zero or to solid material which can be any other unsigned integer. The value of each material index is determined through sampling of the SDF at the respective grid point world positions. Following the description of the SDF from Section 2.3, a grid point lies inside the volume and represents solid material if the SDF returns a density value $d \leq 0$.

An edge between two adjacent grid points of which one is solid and the other is air exhibits a material change and contains the contour of the volume that is described by the SDF. Only these edges are important and need to be tagged with additional surface intersection data obtained from the SDF. Ju et al. (2002)

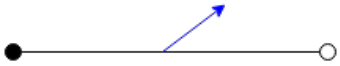

Figure 14: A grid edge that exhibits a material change. The normal vector is shown as a blue arrow originating from the surface intersection point on the edge.

refers to the ensemble of material indices and edge intersection data as Hermite data. Figure 13 depicts a 2D example grid with a chunk resolution of 8. In 3D, a chunk resolution n translates to n voxel cells and $n+1$ material indices in each dimension. Consequently, there are $(n+1)^3$ material indices and a total of $3 \times (n+1)^2 \times n$ edges, but the number of edges that actually contain the volume's surface is usually much lower. Edge intersection data is obtained through Zero Crossing approximation which is described in Section 4.4.

While a high resolution allows the surface extraction procedure to pick up more details of the implicit surface, it also results in an increased number of generated vertices and has a negative impact on processing time. Note that the world distance between the material indices describes the effective resolution of the chunk and the generated mesh.

4.4 Zero Crossing Approximation

Edges that exhibit a material change from solid material to air or vice versa intersect with the isosurface of the volume. They need to be examined closely to find the Zero Crossing - the point where the SDF assumes the isovalue $c = 0$. Given that the world positions of an edge's starting and ending point are known, the problem can be reduced to a generic root finding problem of the form $f(x) = 0$ where f is the SDF and x is an unknown root. The problem can further be condensed into finding a value $t \in [0, 1]$ that represents the relative intersection point along the edge.

In practice, the SDF may be sampled in discrete steps along the edge to find a point where the isovalue is closest to zero. This naive method, however, limits the result to a very small set of possible values. Even with five sampling steps the Zero Crossing can only assume the values $\{0, 0.25, 0.5, 0.75, 1\}$ which is an unnecessary loss of information. Thus, a more advanced method is required that can approximate the intersection point more accurately and still offers a reasonably high performance. "One of the best, most effective methods for finding the real zeros of a continuous function is the bisection method" (Hamming 2012, p. 62). This method is also know as the binary search algorithm and cuts an initial interval $[x_1, x_2]$ in half by calculating the midpoint x_3. It then continues to search for the root in one of the two new sub-intervals based on the following condition:

$$f(x_1) \times f(x_3) = \begin{cases} < 0 & \text{then there is a sign change in } [x_1, x_3] \\ > 0 & \text{then there is a sign change in } [x_3, x_2] \\ = 0 & \text{then } x_3 \text{ is a zero} \end{cases}$$

The bisection is usually repeated as many times as necessary to find a function value that is equal to zero. A disadvantage of the bisection method is that it converges slowly towards the perfect solution. In fact, the method often reaches a satisfactory solution after less than eight steps and might actually never reach the perfect solution on a computer system due to rounding errors that are caused by the internal number representation.

The terrain engine uses three safeguards to allow early terminations of Zero Crossing approximations. First, the iteration count is limited to eight steps. If this limit is breached, the midpoint that was calculated last is used as the solution. Secondly, the size of the created sub-intervals is limited by a threshold of 1e−6. Lastly, a bias of 1e−2 is used for the density values returned by the SDF to accept solutions that are sufficiently accurate.

It's worth mentioning that the Zero Crossing approximation assumes that there is only one material change along the inspected edge. Therefore, this approach can only find one Zero Crossing per edge, but multiple surface intersections on a single edge are rather uncommon. Figure 14 shows an edge with a Zero Crossing value of roughly 0.5 and a surface normal that originates from it. When the intersection point is known, the surface normal can be approximated using a finite difference method or it can be calculated accurately using analytical derivation of the SDF. The terrain engine currently uses the central difference approach due to a lack of a mathematical expression system that could be used for automatic derivation. Let f be an SDF and let $\epsilon = 0.001$. The gradient at a specific point in space can then be approximated as follows:

$$\begin{aligned} nx &= f(x + \epsilon, y, z) - f(x - \epsilon, y, z) \\ ny &= f(x, y + \epsilon, z) - f(x, y - \epsilon, z) \\ nz &= f(x, y, z + \epsilon) - f(x, y, z - \epsilon) \end{aligned}$$

4.5 Volume Data

All previous presentations of isosurface extraction techniques use a single material grid that completely encloses the SDF. Creating a single grid of discrete volume data on the fly and discarding it as soon as the surface has been constructed is a justifiable option if the extent of the implicit surface is in a predictable margin and processing time is not a critical factor. In contrast to this, a terrain can

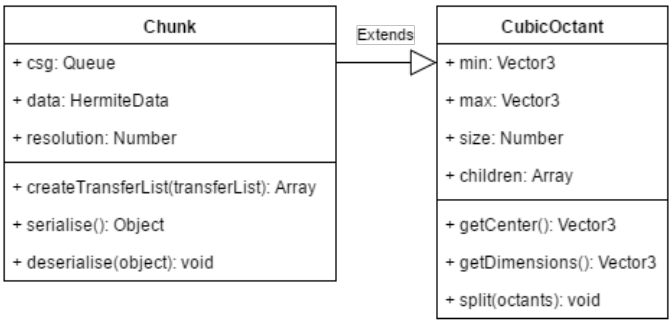

Figure 15: The Chunk class that extends the CubicOctant class to maintain Hermite data in an organised fashion.

push hardware limits in terms of data size and it must be rendered as quickly as possible due to its omnipresence in game worlds. Sampling an SDF over a large region with a low chunk resolution is not an option as this results in an oversimplified surface mesh. For that reason, the sampled volume data is kept in chunks and the SDFs are discarded as soon as the Hermite data has been built.

Volume chunks are organised in a sparse octree and maintain a description of the terrain at a fixed resolution. Figure 15 shows the Chunk class that extends the CubicOctant class provided by the external sparse octree module. Cubic octants only need to store their lower bounds in the form of a vector with three components plus a single size value. This reduces the memory consumption of the chunks and makes them easier to work with.

Note that JavaScript allows the definition of intelligent prototype properties in the form of Getters and Setters which are functions that behave like standard object properties. The CubicOctant class shown in Figure 15 uses this common technique for its max field which is only calculated on demand and doesn't occupy memory. It's important to be aware of the fact that this technique may cause a lot of object instantiations if used inappropriately. The upper bounds of cubic octants are therefore temporarily cached in certain parts of the terrain engine to reduce memory cluttering.

The benefit of using multiple clustered chunks to store the volume data is the ability to execute modifications and surface extractions in parallel. Additionally, the computational load of the system doesn't increase as the volume undergoes a multitude of consecutive modifications because all added SDFs simply transform the discrete volume data. Furthermore, the partitioned volume can be culled to focus computation power on portions of the volume that are in the field of view.

If the terrain engine only relied on SDFs to maintain a representation of the volume, it would be necessary to concatenate them with more SDFs for each new volume modification. Recurring extractions of the terrain surface from an SDF that becomes increasingly more complex would slow the system down further

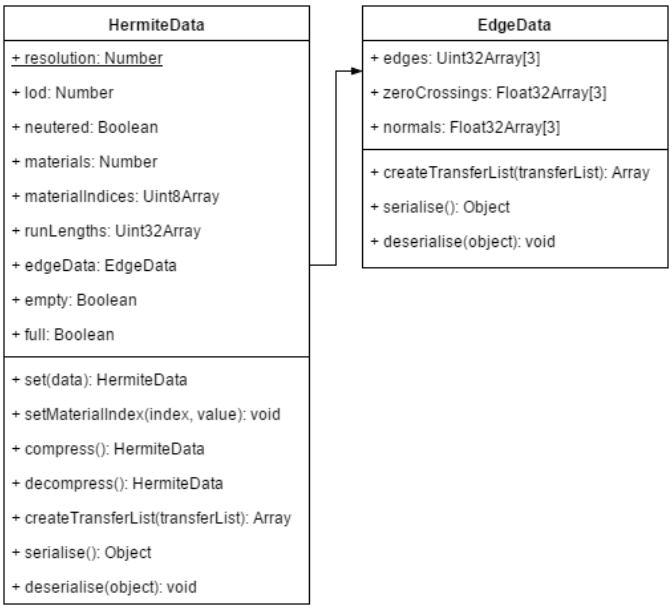

Figure 16: The HermiteData class and the EdgeData class.

and further. It's easy to see that this approach would quickly become impractical.

ES2015 supports raw binary data in the form of typed arrays which can be used to efficiently store a fixed amount of numerical values of a specific type. Compared to dynamic arrays, they perform much better in terms of read and write operations. Additionally, the typed arrays allow zero-copy data communication with Web Workers and are indispensable for the multithreaded approach that the terrain engine follows. Section 4.10 provides more details about the implemented multithreading strategy.

Figure 16 shows the HermiteData class and explains how the volume data is maintained. Material indices are kept in a one-dimensional typed array of 8-bit unsigned integers and the chunk resolution is constrained to powers of two ranging from 1 to 256. The three-dimensional arrangement of the grid points is preserved by translating their coordinates into a one-dimensional index. Let n be the chunk resolution and let x, y, z be integer grid coordinates. The flattened index of a specific material index grid point can then be calculated as follows: $z \times (n+1)^2 + y \times (n+1) + x$. As a result, the position of a material index in the array encodes its local position inside the material grid which can in turn be translated into a unique world position based on the lower bounds of the enclosing chunk's AABB. Consequently, the complete material grid needs to be available for modifications and surface extractions which conflicts with the idea of storing this data sparsely.

Edge data, on the other hand, can and should be maintained sparsely to save

space. The most obvious data structure for this undertaking would be a hash map and this could indeed be a feasible option in other programming languages, but native support for hash maps was only just recently added to JavaScript in ES2015. Hash maps in JavaScript are therefore still a rather young feature and not well supported. Considering the fact that an edge data hash map could easily contain more than a few thousand edges, the performance of hash maps could become unpredictable. An even more convincing aspect that speaks against the use of hash maps is that they are highly incompatible with Web Workers. Thus, typed arrays are used for the edge data, too.

In order to construct a data structure with typed arrays that simulates a hash map, a slightly more complex scheme must be developed. As shown in Figure 16, the EdgeData class stores edges, Zero Crossings and normals separately. Additionally, each of these three fields is further split into three arrays that hold the data for edges along the X-, Y- and Z-axis. Furthermore, all edges are stored as starting grid point indices in ascending order. This information combined with the dimension split is enough to uphold the association between edge data and pairs of adjacent grid points that exhibit a material change. The ending point indices are implicitly defined through the dimension split and the storage structure of the material grid: given a starting point index a, the ending point index b for the X-, Y- and Z-axis is defined as $a + 1$, $a + (n + 1)$ and $a + (n + 1)^2$ respectively where n is the chunk resolution. Each Zero Crossing value describes the relative surface intersection position on the respective edge. The values correspond to the order of the edges. Normal vectors are stored as (x, y, z) floating point triples and also correspond to the order of the edges.

4.6 Updating the Volume Octree

Before any volume data can be generated, the volume octree structure must be prepared in such a way that it accommodates the entirety of the terrain at all times. Since it's not possible to know the extent of the terrain in advance, the volume needs to be able to constantly adapt to changes.

At first, the volume consists of an empty root octant that has an initial size of four times the size of a volume chunk. This precondition paves the way for a robust volume expansion strategy. Furthermore, the system differentiates between two main cases:

1. New volume data will be added to the current volume.
2. The current volume will be reduced.

For the first case, the volume may have to be expanded and chunks that don't exist yet might need to be created. For the second case, it's only necessary to find existing chunks that may be affected by the volume reduction because chunks

that don't exist obviously can't become more empty.

The expansion of the volume octree is mainly guided by the AABB of the given SDF that is used for the volume modification. This AABB needs to describe the full reach of the SDF by taking the possibility into account that it may be a composite of several SDFs. For the first step of the expansion process, the absolute maximum of the AABB has to be determined. For example, the absolute maximum of an AABB with $\vec{min} = (-1, -9, -5)$ and $\vec{max} = (1, 2, 3)$ would be $m = 9$. If this value exceeds the current bounds of the volume octree, an appropriate target octree size must be found. This can be done by calculating $n = \lceil m \div \text{chunk size} \rceil \times \text{chunk size}$. Although it isn't necessary to constrain the octree size to powers of two, this practice simplifies calculations and may be beneficial for future optimisations. Therefore, the maximum value is rounded up to the next power of two with the following calculation: $n = 2^{\lceil \log_2 m \rceil}$.

In case the current volume is completely empty, the AABB of the octree's root octant can safely be adjusted directly according to the calculated target size by moving its lower bounds to $(-n, -n, -n)$ and setting its size to $2 \times n$. This ensures that the centre of the octree remains at the scene's origin $(0, 0, 0)$ which is beneficial for further operations. If the volume is not empty, it must be expanded carefully while preserving all existing volume chunks. For this, the size of the octrees' root is first divided by two which yields a value that represents the current reach of the octree in each direction. This value is then doubled repeatedly until the target size is reached. During each step, the root octant's AABB is expanded accordingly and becomes disconnected from its child octants. In order to reconnect the orphaned octants with the octree every time, the root octant is split which results in eight new children being created. Only those of the original children that actually contain deeper structures are then integrated into the new intermediate octants. Otherwise, the octree would often create empty subtrees. Due to the fact that the octree is expanded evenly in each direction, the original children always lie inside the newly created octants and can therefore easily be recycled by splitting the respective encompassing octants. Recall also that the initial octree size is four times the size of a volume chunk. This precondition is crucial for checking whether octants contain deeper structures because leaf octants don't contain children and must be excluded from this check. By repeatedly doubling the volume's size, the target size can be reached very quickly.

After the octree has been adjusted to accommodate the extent of the given SDF, the affected volume chunks need to be identified. By performing recursive octant intersection tests using the AABB of the SDF starting with the octree's root, a list of the affected octants can be obtained. However, some of them may not be leaf octants due to the sparse structure of the octree. Those intermediate octants must be split and the created children have to be checked against the AABB as well. This splitting process results in the creation of the octree structures down to the volume chunks that will hold the Hermite data. When the full list of affected

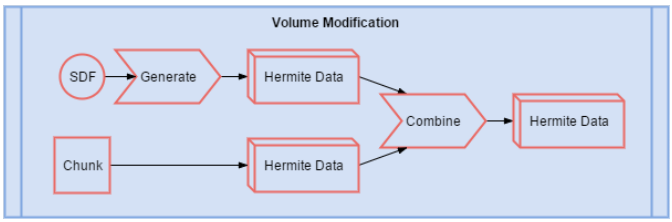

Figure 17: An overview of the volume modification.

leaf octants has been created, the actual volume modification takes place.

4.7 Volume Modification

Figure 17 provides an overview of the volume modification process and shows that its input consists of the SDF and an affected volume chunk. Each chunk of volume data is, in fact, modified in parallel and the result of every modification is another chunk of volume data which qualifies as input for further modifications.

SDFs are a fundamental part of the volume modification system. Like other CSG libraries such as csg.js, the terrain engine also provides a set of SDFs that represent primitive solids, namely a box, sphere and torus. Volume modifications strictly follow the CSG methodology in order to combine volumes in a structured and predictable way. Therefore, all SDFs can be linked together conveniently via the three chainable CSG methods union, subtract and intersect to construct arbitrarily complex SDF composites.

The terrain itself exposes the same three CSG operations that SDFs offer and calling them triggers an update of the volume octree. This active modification process also identifies volume chunks that are likely to be affected by the SDF. The actual modification of the volume data, however, is not executed immediately after. Instead, the SDF is simply added to the CSG operation queue of every affected chunk. This queue, which has previously been shown in the description of the Chunk class in Figure 15, manages items according to the First in, first out (FIFO) method. Additionally, the SDF is added to a global CSG operation history that keeps a chronological list of all executed SDFs. The invaluable advantage of letting every volume chunk maintain its own independent CSG queue instead of using a single central queue makes lazy modifications possible which are only executed when the associated chunk is actually seen by the viewer. Additionally, they simplify the overall management of CSG operations.

The use of a central queue is discouraged, as it has initially been used in this project and caused many inconveniences such as the need for complicated bookkeeping of associations between operations and chunks. This approach also doesn't provide a noticeable advantage in terms of memory usage because of the additional data structures that are necessary to uphold said associations. In fact, the independent CSG queues of the volume chunks only need to be kept

Table 2: An overview of the CSG operations applied to sets (A, B) and to Signed Distance Functions (f, g).

	Sets	Signed Distance Functions
Negation	$\neg A$	$-f$
Union	$A \cup B$	$\min(f, g)$
Difference	$A \setminus B$	$\max(f, -g)$
Intersection	$A \cap B$	$\max(f, g)$

in memory for as long as they contain items, which is usually a very short time span.

Based on the definition from Section 2.3, an SDF yields negative values for points that lie inside the volume and positive values for points outside. Thus, the combination of multiple SDFs via CSG can be formulated mathematically. Table 2 has been created according to a description by Trettner (2013) and shows how the semantics of the CSG operations can be translated to SDFs. Note that the negation of a set $(\neg A)$ directly translates to negating the function value of an SDF. Suppose, for example, that an SDF $f(x)$ returns a negative value for a specific point x which implies that the point lies inside the volume. By negating the returned function value, the point would consequently be considered outside. Combining SDFs according to these rules is straight-forward and provides a robust strategy for combining SDFs into complex implicit surfaces. However, the terrain engine doesn't rely on SDFs to describe the terrain's volume. Instead, it uses chunks of discrete Hermite data and generates and transforms them with the help of SDFs. Thus, a more complex strategy must be devised that focuses on the structured and organised combination of discrete data sets.

The main purpose of an SDF is to define a three-dimensional shape. In order to preserve this distinct role, SDFs are wrapped in CSG operations which define combination logic on top of them. An example of a conversion from an SDF composite into a CSG expression is shown in Listing 3. The variables a to e in this example are references to SDF instances.

```
a.union(b.intersect(c)).union(d).subtract(e)
    => Difference(Union(a, Intersection(b, c), d), e)
```

Listing 3: Conversion from an SDF composite into a CSG expression.

When an SDF is added to another, it becomes a child of the target SDF and it gets tagged with a CSG operation type. This allows the creation of composites through nesting. During the automatic conversion into a CSG expression, every

SDF of a composite is wrapped in a special Density Function CSG operation that doesn't provide any volume combination logic, but instead defines methods that use the attached SDF to generate volume data. Union, Difference and Intersection operations, on the other hand, have no access to the SDFs and only define how existing volume data is combined. The core strategy for the modification of discrete Hermite data using SDFs can be described as follows:

1. Fully execute the given SDF to generate a single independent set of Hermite data that captures the implicit surface inside the current chunk's boundaries.
2. Combine the generated data with the existing data according to the chosen CSG operation type.

What is meant by fully executing an SDF is that it needs to be evaluated completely, including its children if it is a composite. The whole process of modifying volume data is always limited to the AABB of the current chunk at hand.

Relying on the mathematical approach to combine SDFs during the generation of volume data is a feasible option, but this introduces a drawback. Each SDF may define a custom material index for the volume that it describes. Merging multiple SDFs would result in the loss of this information. Therefore, each component of the SDF composite is executed individually which results in the creation of several sets of Hermite data that are ultimately combined into one.

It could be argued that most SDF composites would rarely use different materials for their components and that treating the components separately just to preserve the material information is unnecessary. However, it could also be argued that SDFs that are used for terrain modifications are rarely complex and that the extra work is therefore usually within limits.

In any case, combining a data set with another one remains the central problem of the volume modification process. Furthermore, SDFs may be executed using both approaches depending on whether they contain components with varying materials. The terrain engine solely relies on the combination of data sets and executes the components of SDF composites one after another to accumulate the complete data set. The mathematical combination of SDFs is faster and requires less memory because it only creates one data set. Thus, it can be considered an optional optimisation.

The generation of volume data starts with the creation of a blank set of Hermite data containing no edge data and a material grid in which all material indices are set to air. After that, the material grid and edge data is updated by evaluating the given SDF. It's important to realise that each CSG operation, be it a Union or a Difference operation, starts with the generation of a discrete data set. The generated material indices and edges describe the given SDF in a form

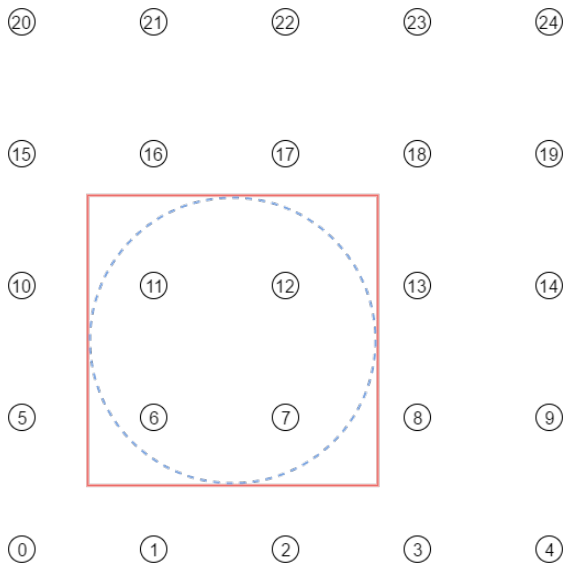

Figure 18: Identification of affected grid points using the operation's reach. In many cases, only a subset of the grid points needs to be processed.

that is compatible with the existing terrain volume data. In case no terrain data exists yet, the generated data may directly be adopted depending on the operation type. Otherwise, the generated data is merged with the existing data in a subsequent combination process; material index by material index, edge by edge.

With a worst-case time complexity of $\mathcal{O}(n^3)$, the generation and combination of material indices is quite costly. However, both processes can be sped up significantly by limiting the work to grid points that lie inside of the SDF's AABB. Since the generation of volume data is always performed on a blank data set, the unaffected grid points can safely be ignored. The combination of generated data with existing data, however, can only be accelerated with this method for Union and Difference operations since these two don't depend on existing data. Intersection operations, on the other hand, do depend on existing data and always influence the entire volume. Hence, all solid material indices that lie outside the AABB of an Intersection operation's SDF need to be set to air. Due to the destructive nature of this operation, it's rarely used for terrain modifications. For example, a single Intersection operation could easily delete most of the existing data and must therefore be used with caution. SDF composites are more likely to contain Intersection operations since their effect is then local to the composite.

Figure 18 shows a 2D material grid with a chunk resolution of 4 which is used to capture an SDF that describes a circle. Note that in this case, the grid completely covers the extent of the SDF. In practice, most SDFs would typically straddle multiple grids. The figure also shows how the grid points are numbered.

In a 3D material grid, the numbering would continue at the bottom row of the next layer. Furthermore, the SDF itself is shown in blue while its AABB is shown in red. The AABB of the implicit surface can directly be used to identify the grid points that need to be updated. Recall that the identification of affected chunks relies on the complete AABB of the SDF. This means that not all of the identified chunks have to be affected by all components of an SDF composite. Since the components are executed separately, their individual AABBs can be used to limit the respective grid updates. An intersection test with the chunk's AABB reveals whether the component needs to be executed for the current chunk. Again, this doesn't apply when an Intersection operation is used for the combination of volume data because all chunks are affected in this case.

Provided that the conditions for the grid update limitation are met and that the current SDF intersects with the chunk's AABB, the grid point index bounds $\vec{min}$ and $\vec{max}$ can be calculated as shown in Listing 4.

```
min.copy(sdf.aabb.min).max(chunk.min).sub(chunk.min);

min.x = Math.ceil(min.x * resolution / size);
min.y = Math.ceil(min.y * resolution / size);
min.z = Math.ceil(min.z * resolution / size);

max.copy(sdf.aabb.max).min(chunk.max).sub(chunk.min);

max.x = Math.floor(max.x * resolution / size);
max.y = Math.floor(max.y * resolution / size);
max.z = Math.floor(max.z * resolution / size);
```

Listing 4: Identification of affected grid points.

The calculated bounds are then used to iterate over a portion of the material grid. For example, the AABB shown in Figure 18 contains the grid points $\{6, 7, 11, 12\}$ and translates to the lower bounds $(1, 1)$ and the upper bounds $(2, 2)$. Using these bounds, the iteration would start at 1 and end at 2 for both the X- and Y-axis. According to the index calculation scheme from Section 4.5, the one-dimensional index of the grid point with the iteration coordinates $(1, 2)$ would, for example, be $2 \times (4 + 1) + 1 = 11$.

For the generation of volume data, it's also necessary to calculate the world position of each affected grid point. Let s be the size of the volume chunk and let n be the resolution. The local offset of a grid point can then be calculated based on the iteration indices:

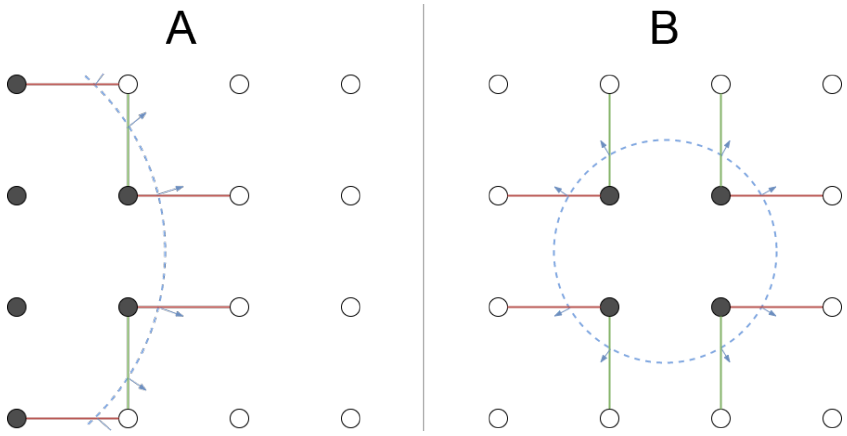

Figure 19: Two exemplary sets of 2D Hermite data.

$$\vec{\text{offset}} = \left(\frac{x \times s}{n}, \frac{y \times s}{n}, \frac{z \times s}{n} \right)$$

Adding this offset to the chunk's $\vec{min}$ position yields the world position of the grid point which can be used to sample the SDF and to determine whether the respective material index should be set to solid material or to air.

The process of generating edge data has a worst-case time complexity of $\mathcal{O}(3 \times n^3)$ and is even more costly than the generation of material indices. In order to handle this task efficiently, a divide and conquer approach is used. As stated in Section 4.5, edges are stored separately for each axis. This stems from the fact that edges are first processed along the X-axis, then Y and finally Z.

The goal of the edge generation process is to generate and store surface intersection data for edges that exhibit a material change and thus contain the contour of the implicit surface. According to the description of the material grid's structure from Section 4.5, the ending grid point index b of an edge can easily be determined by adding a fixed offset to the starting grid point index a. Using the example from Figure 18 and assuming that edges are currently being processed along the Y-axis, the ending grid point index $b = a + (n + 1)$ for the starting grid point index $a = 11$ would be $11 + (4 + 1) = 16$. When both the starting and ending grid point indices are known, the respective material indices can be checked to see if the edge exhibits a material change. If it does, the grid points are translated into world positions according to the offset calculation described above. The edge is then processed according to the Zero Crossing approximation described in Section 4.4 to obtain the surface intersection data.

Furthermore, it's important to adjust the grid index bounds for the generation and combination of edge data in order to include edges that straddle the AABB

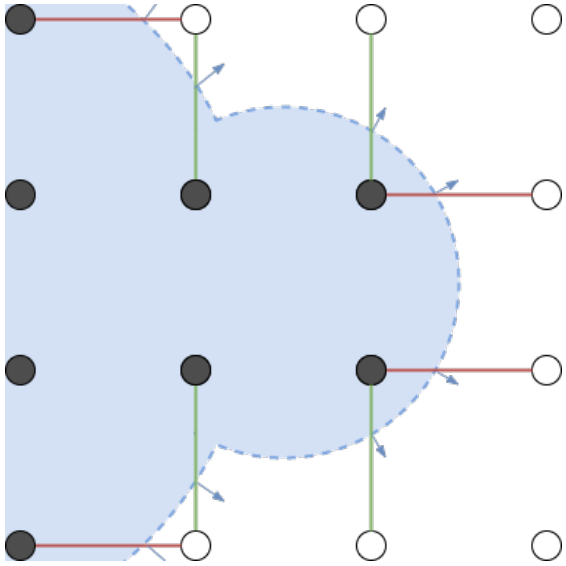

Figure 20: An example of a CSG Union operation.

of the SDF and to avoid processing of non-existing edges at the grid borders. Depending on the axis, the respective x, y or z component of both the $\vec{min}$ and $\vec{max}$ vector must be updated as shown in Listing 5.

```
min[axis] = Math.max(min[axis] - 1, 0);
max[axis] = Math.min(max[axis], resolution - 1);
```

Listing 5: Adjustment of grid point index bounds for edge processing.

Although the number of edges that contain the implicit surface is usually very low, the potential maximum amount of edges must always be accounted for. Thus, the arrays that are used to store the starting grid point indices, intersection normals and Zero Crossings all need to be initialised with the maximum size. More sophisticated strategies may be applied to reduce the space complexity at the risk of having to perform costly array resizing. For the combination process, the array size can be limited to the sum of the existing and generated edges. Empty space that remains can safely be cut off afterwards in both cases.

After the SDF has fully been executed, the generated data can be combined with the existing data. The process of combining a data set A with another set B consists of updating affected material indices and deciding which edges to keep. While the generation and combination of material indices both have the same time complexity, the combination of edge data only has a time complexity of

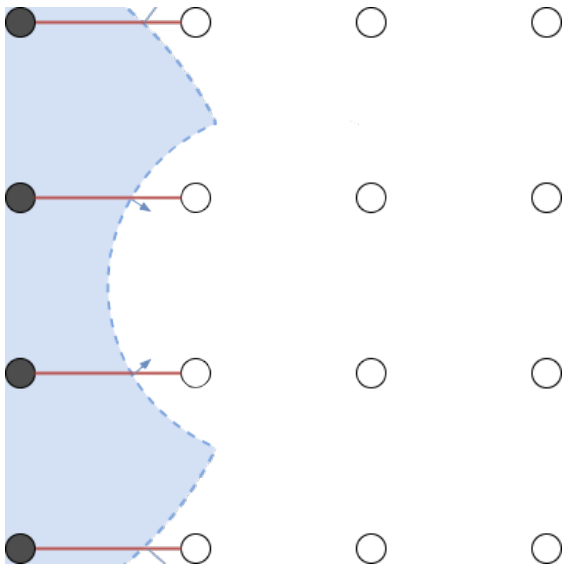

Figure 21: An example of a CSG Difference operation.

$\mathcal{O}(n)$. To provide a clear description of the actual combination process, a visual example is given for each CSG operation type. Figure 19 shows two exemplary sets of Hermite data. Set A represents an existing chunk of volume data while set B represents a predominant set of generated data. Solid material indices are depicted as black dots while empty material indices are coloured white. Edges along the X-axis are coloured red and edges along the Y-axis are coloured green. The surface intersection normals are shown as blue arrows.

"For $A \cup B$, all non-air materials of B override the corresponding material in A" (Trettner 2013). Edges that exhibit a material change are updated accordingly. For $A \cup B$, all edges of B override the corresponding edge in A if their respective Zero Crossing position is closer to the air grid point. Ignoring this important constraint could lead to an undesired reduction of the volume.

Furthermore, all edges in A that no longer exhibit a material change are removed, or rather discarded since the edge selection process actually collects the final edges in a new edge data instance. The effect of the Union operation can be seen in Figure 20. Notice how the green edges from set A have been selected instead of the conflicting edges from set B.

"For $A \setminus B$, all non-air materials of B result in air" (Trettner 2013). Similarly, the edges of B override the corresponding edge in A, but only if they still connect different materials in A and their respective Zero Crossing position is closer to the non-air grid point. "Otherwise, the difference operation could wrongly increase the volume" (Trettner 2013). Additionally, the intersection normals of edges that were adopted from B must be inverted to keep the description of the surface consistent. Figure 21 shows the result of the Difference operation and

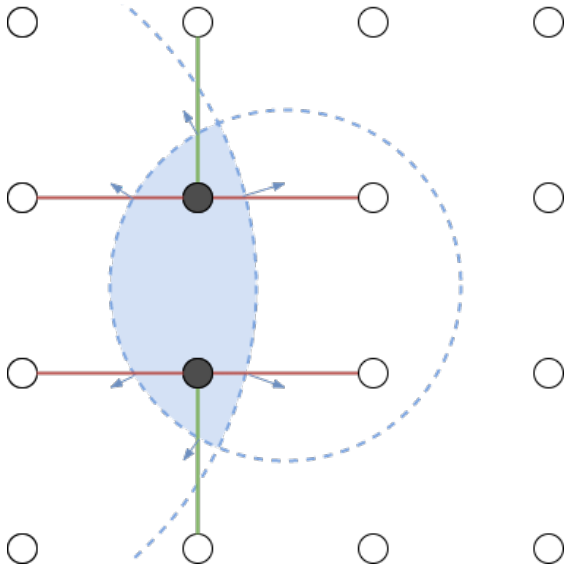

Figure 22: An example of a CSG Intersection operation.

demonstrates that the normals from B have been inverted.

Lastly, $A \cap B$ sets all materials of A to air except for materials that are solid in both A and B in which case the material of B is chosen. Similarly, all edges in A that no longer exhibit a material change are discarded and the edges of B override the corresponding edge in A if they exhibit a material change in A and their respective Zero Crossing position is closer to the non-air grid point. The effect of the Intersection operation is shown in Figure 22.

On a technical level, the combination of edge data is mainly driven by the generated edge data. Recall that only the starting grid point indices of the edges are stored and that they are sorted in ascending order. This allows the edge combination algorithm to capitalise on a few assumptions in order to collect all relevant edges in one fell swoop.

While iterating over the set of generated edges, the starting and ending grid point indices of each edge are used to check if there is still a material change on the edge. If there is none, the edge can be discarded and the iteration continues. However, if there is a material change, the algorithm enters an inner loop to process existing edges up to the current generated edge. This catch up mechanism picks up existing edges that also exhibit a material change and have been skipped by the outer loop. If this loop happens to reach an existing edge that has the same starting grid point index as the current generated edge, then there is a conflict which needs to be solved by selecting an edge to keep based on the CSG operation type. Furthermore, the inner loop is not reset so that it may continue where it left off. After the generated edges have all been processed, the remaining existing edges are collected to complete the process.

Table 3: An example of volume data memory usage. The presented data is an accumulation of 152 volume chunks with a resolution of 64.

	Material Indices	**Edges**
Maximum Count	41743000	123302400
Solid Materials	2163224	-
Actual Count	169856	375336
Max. Mem. Usage	39.81 MB	2351.81 MB
Actual Mem. Usage	829.38 KB	7.16 MB
Compression Ratio	49.15	328.51
Space Savings	97.97%	99.70%

4.8 Data Compression

Memory consumption is one of the biggest concerns when it comes to maintaining large amounts of volume data. Table 3 provides insight into the actual memory usage of the engine for an exemplary scene in which a large torus with an inner radius of 200 and a tube radius of 8 occupies a total of 152 volume chunks. These chunks all have a size of 32 and a resolution of 64. The maximum amount of material indices of a single chunk with a resolution of 64 is $(64 + 1)^3 = 274625$. With 152 chunks this amounts to 41743000 potential material indices. Assuming that each material index is stored as an 8-bit unsigned integer, the total memory usage for the maximum number of material indices is roughly 39.81 MB.

A closer inspection of the material data reveals that its structure is predestined for data compression. Solid and empty material indices are often stored as uniform sequences that tend to be fairly long, resulting in a strikingly low variety of data. Every material index is important and can't just be truncated, but the identified structure promises high potential for compression. A prominent compression approach that exploits data repetition effectively is the Run-Length Encoding (RLE) algorithm. It belongs to the group of entropy encoders and quickly compresses data in a lossless way. "The idea behind this approach to data compression is this: If a data item d occurs n consecutive times in the input stream, replace the n occurrences with the single pair nd." (Salomon 2004, p. 20). In the terrain engine, this approach is applied to numerical material index arrays. Since material indices are stored in a one-dimensional array, it's easy to count repeating occurrences. A streak of repeating values is called a run and the number of occurrences in a run is called a run-length. For example, an array containing the following values: $\{0, 0, 0, 0, 0, 0, 1, 1, 1\}$ would result in the compressed data: $\{0, 1\}$ plus the run-lengths: $\{6, 3\}$. This shows that the effectiveness of RLE highly depends on the structure of the data.

As previously shown in Figure 16, the run-lengths array of the HermiteData class uses 32-bit unsigned integers. This is due to the fact that a single run-length

must be able to hold the maximum number of material indices which cannot be achieved with only 16-bit. If a chunk of Hermite data contains only solid material indices it is considered full. This is the case if the chunk lies completely inside the terrain's volume. Compressing chunks that are full outputs only one material index and one run-length that holds the total material index count. While empty chunks can safely be discarded, full chunks must be maintained as they still contain meaningful information.

Table 3 shows that by applying the RLE algorithm, the amount of material indices in the example scene can been reduced to 169856 plus 169856 run-lengths. Since material indices use 8-bit and run-lengths 32-bit, each run-length value counts as four material indices. Together, this amounts to 849280 8-bit long values which is only 2.03% of the maximum material index count. The actual space requirement of the compressed data is roughly 829.38 KB which proves that RLE is well suited for the data at hand.

The maximum possible number of edges for a chunk with the same resolution is $3 \times (64+1)^2 \times 64 = 811200$. Multiplied by 152, this results in 123302400 potential edges in total. Each edge requires a 32-bit unsigned integer to store the index of its starting grid point, an additional 32-bit floating point value for its Zero Crossing position and three 32-bit floating point values for its normal vector. The maximum space requirement for the edge data of 152 chunks is roughly 2351.81 MB. Storing this much data just for the terrain would be disadvantageous for a game which has to keep many other assets in memory. Thankfully, the actual memory usage is much lower than these estimated numbers. As can be seen in Table 3, the amount of edges in the test case is 375336 which is only 0.3% of the maximum count. A chunk could only ever be fully populated with edges if it contained an implicit surface that returned evenly distributed noise and it is unlikely that such a function would be used for terrain. With a total of 7.16 Megabyte for the tested scene, the space requirement for edge data can be considered manageable and it seems rather unnecessary to compress this data further.

4.9 Surface Extraction

Similar to the volume modification process, all surface extractions are executed in parallel for individual volume chunks. Contrary to the previous presentation of isosurface extraction techniques, the input of the terrain engine's extraction process is a single chunk of discrete volume data instead of an SDF. This means that the extraction process doesn't need to evaluate the SDF on the fly as the necessary data is already directly available.

Furthermore, the terrain engine uses the DC algorithm to create polygonal meshes from volume data. For this technique, the raw volume data needs to be converted into a disposable octree of voxel cells. All of the created voxel cells are constructed on top of the material grid; the corner vertices of the cells match

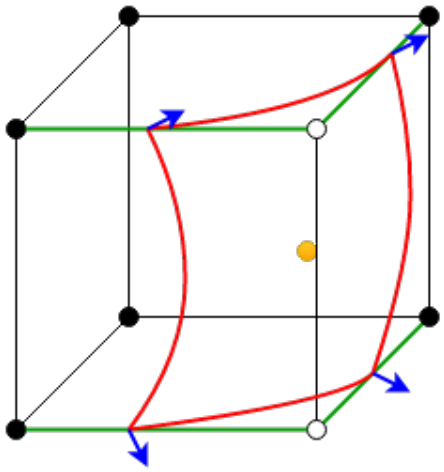

Figure 23: A voxel cell that contains a part of the implicit surface.

with the position of the material index grid points. Consequently, the material information and edge data is shared by adjacent voxel cells. Furthermore, the maximum amount of voxel cells is defined as n^3 where n is the resolution of the volume chunk.

A voxel cell contains QEF data which is an accumulation of the edge data associated with the cell. To be precise, the surface intersection positions that are described by the Zero Crossing interpolation values and the respective intersection normals are used to describe a linear system of intersecting planes. By solving this system, a single point can be determined that approximates the isosurface of the volume for that particular cell. Moreover, the generated feature point becomes a vertex of the final polygonal mesh. Additionally, the respective vertex normal is calculated by taking the average of the involved surface intersection normals. Figure 23 shows a single voxel cell of which four edges contain the surface shown in red. The surface intersection normals at the Zero Crossing positions are depicted as blue arrows and the computed vertex is shown as a yellow dot.

"Given a plane π, defined by a point P and a normal n, all points X on the plane satisfy the equation $n \cdot (X - P) = 0$ (that is, the vector from P to X is perpendicular to n)" (Ericson 2004, p. 126). Finding the intersection point x of three planes can thus be formulated as the linear system:

$$
\begin{aligned}
n_1 \cdot (x - P_1) &= 0 \\
n_2 \cdot (x - P_2) &= 0 \\
n_3 \cdot (x - P_3) &= 0
\end{aligned}
$$

Solving this system is only possible as long as the edge data describes at least three intersecting planes. However, the implicit surface may intersect with a voxel cell in such a way that the intersection points and normals describe only two planes and sometimes only one. Since the intersection of two planes is a line, it's not possible to find a single point of intersection. In this case, the linear system is called underdetermined which means that there is not enough information to find the exact feature point. Therefore, a least squares solution is computed using the accumulated QEF data. According to Tzur (2003), the exact intersection point is approximated using the following equation:

$$E(x) = x - n_i \cdot P_i$$

The obtained point minimises the distances to all planes involved. With this approach, the QEF solver will occasionally compute a point that lies outside the voxel cell. For this case, the solution falls back to the average of the intersection positions which is called the mass point of the voxel.

Furthermore, each voxel cell contains one byte that stores the combined materials of its eight cell corners. Unlike MC which uses this information to identify a polygon configuration, DC only uses it to quickly check if edges exhibit a material change and to find out if the QEF of the current cell is underdetermined.

The contouring algorithm itself is solely responsible for the creation of polygons based on the vertices stored in the voxel cells. The DC algorithm traverses the voxel octree by executing multiple recursive procedures which were provided in the original DC implementation and constructs the final polygonal mesh by tying the vertices of adjacent voxel cells together.

Before the surface is constructed, the octree is run through a simplification procedure in an attempt to merge groups of eight voxel cells. It combines their QEF data, solves the QEF and checks if the error of the computed position is below a certain threshold. Groups that fail this check are left untouched. The simplification operates in a bottom-up fashion and only combines cells of the same size that are either leaf octants or clustered intermediate octants that contain the information of multiple merged voxel cells. This process can reduce the final vertex count of the mesh significantly by preventing unnecessary tessellation along almost straight lines.

As mentioned above, the volume data needs to be converted into a temporary voxel octree which serves as input for the contouring algorithm. If the volume data had to be created at this point by sampling an SDF, the strategy for building the octree would be to create all voxel cells and to sample the SDF at the eight corners of all of the created cells. The fact that the complete volume data is already available allows for a more sophisticated strategy.

Since all edges that exhibit a material change are known, the process of building a voxel octree can rely on the edge data to determine which voxel cells need to be created. Due to the way the edges are stored, the process is performed in

three steps for the X-, Y- and Z-axis. Each edge is uniquely described by its starting grid point index which can be decomposed into the local grid coordinates. Let n be the resolution of the chunk and let i be the index of the edge's starting grid point. The x, y and z coordinates can then be calculated as follows:

$$
\begin{aligned}
x &= i \mod (n+1) \\
y &= \lfloor \left(i \mod (n+1)^2\right) \div (n+1) \rfloor \\
z &= \lfloor i \div (n+1)^2 \rfloor
\end{aligned}
$$

After translating the obtained coordinates into world positions, the Zero Crossing interpolation value of the edge can be used to compute the actual intersection position along the edge. This position will be used during the accumulation of the QEF data later on. Since adjacent voxel cells share their corner grid points and edges, a single edge may belong to up to four voxel cells. The strategy that has been devised for the creation of these potential voxel cells is to rotate around the edge.

Table 4: A list of octant offsets used for the creation of voxel cells.

	X	**Y**	**Z**
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Table 5: A lookup table for voxel cell offsets.

	0	**1**	**2**	**3**
X	0	1	2	3
Y	0	1	4	5
Z	0	2	4	6

Table 4 shows a list of octant offsets that are crucial for the rotation process. In fact, this table also describes the global octant layout of all octrees that are

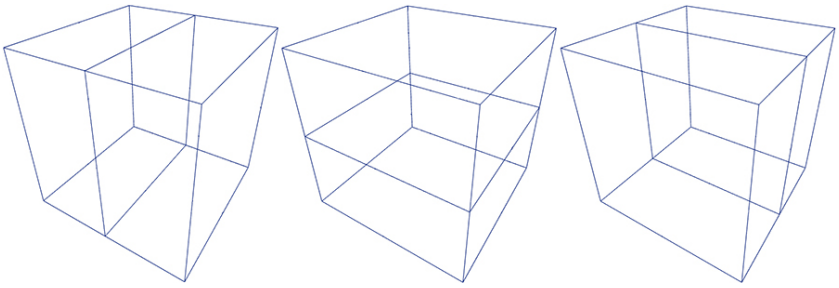

Figure 24: The three-step octant selection. Octant sectors from left to right: YZ, XZ, XY.

used in this project and is merely borrowed for the voxel cell creation. Table 5 describes which of the offsets from Table 4 should be used to identify the four potential voxel cells for each axis. For example, the offset that identifies the third voxel cell belonging to an edge that is aligned with the Y-axis is $(1, 0, 0)$. Adding the offset to the local coordinates of the edge's starting grid point yields a position by which the voxel cell can be identified. However, if the adjusted coordinates no longer lie inside the bounds of the material grid, then there is also no voxel cell that needs to be processed.

Granted that the coordinates still lie inside the bounds of the grid, the next step is to retrieve the voxel cell. In case the cell doesn't exist yet, it must be created which also implies that intermediate octants that don't exist yet must be created, too.

The implemented traversal algorithm which is shown in Listing 6 creates intermediate voxel cells down to the leaf octant that is described by the given local grid coordinates and returns it. Moreover, it starts at the root of the voxel octree and identifies the next child octant using a three-step octant selection scheme that relies on the structure of the material grid and the fact that the voxel cells use the grid points as their corner vertices. The initial value of the parameter n is the resolution of the chunk. Traversing the octree down to the leaf octant only takes a very small amount of steps. With a resolution of $n = 64$, for example, the algorithm would return the requested voxel cell after only six steps.

Figure 24 shows an octant that has been divided into two sectors for the YZ-, XZ- and XY-plane and provides a visualisation of the stepwise octant selection. Depending on the given x, y and z coordinates, the algorithm chooses one of the two sectors for each plane starting with the YZ-plane and ending with the XY-plane. These three binary decisions influence an index that is initially zero. At the beginning, the index can potentially identify any of the octant's children. After each decision, the list of remaining candidates is cut in half. The final value of the index identifies the next child octant that should be traversed. Furthermore, it's important to adjust the coordinates according to the chosen sector so that they stay local.

```
function getCell(cell, n, x, y, z) {

        let i = 0;

        for(n = n >> 1; n > 0; n >>= 1, i = 0) {

                if(x >= n) { i += 4; x -= n; }
                if(y >= n) { i += 2; y -= n; }
                if(z >= n) { i += 1; z -= n; }

                if(cell.children === null) { cell.split(); }

                cell = cell.children[i];

        }

        return cell;

}
```

Listing 6: Voxel cell creation and traversal.

Note that the existence of an edge guarantees that all associated voxel cells contain the implicit surface. Thus, all of them must be created and populated with voxel data. For this, the material information of the eight corners of the cell is packed into a single byte. Additionally, the material information is used to find out how many edges of the voxel cell intersect with the implicit surface. This is necessary because there is no other way to know when the voxel has been populated with all the existing data that belongs to it. As soon as the voxel is ready, it can be finalised by solving its accumulated QEF data and thus generating its vertex and the vertex normal.

A visualisation of the voxel cells that are created during the surface extraction process and serve as input for the DC algorithm is shown in Figure 25. Intermediate octants are not shown. The algorithm that has been developed for the creation of the voxel octree operates very effectively and only creates octant structures that are absolutely necessary.

Figure 26 shows a comparison of a typical box mesh consisting of six planes and a box that has been generated with the DC algorithm. Due to the fact that each voxel cell contains exactly one vertex and one averaged normal, the generated box mesh has only 8 vertices instead of 24 which can be considered a

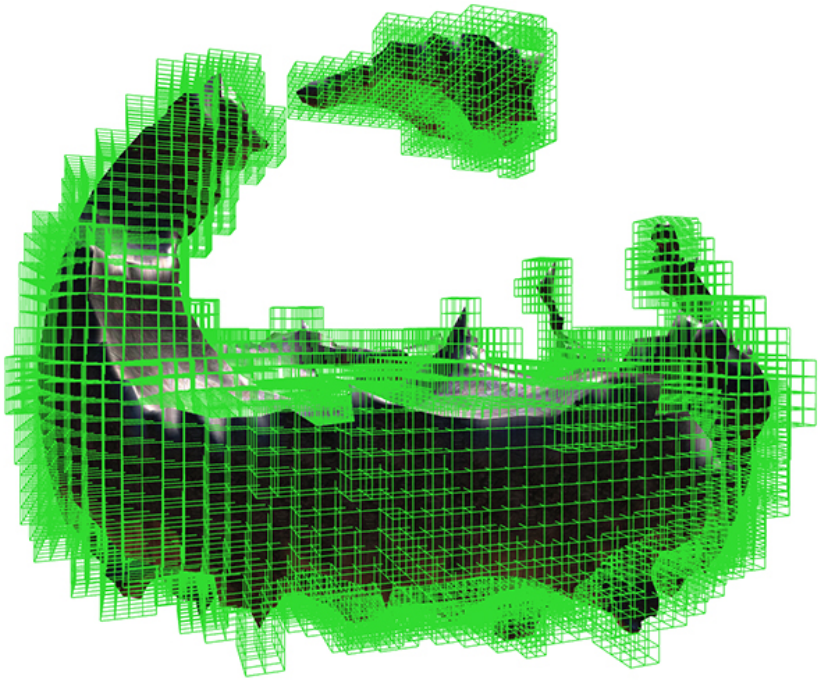

Figure 25: A visualisation of the voxel cells that are created during the surface extraction process.

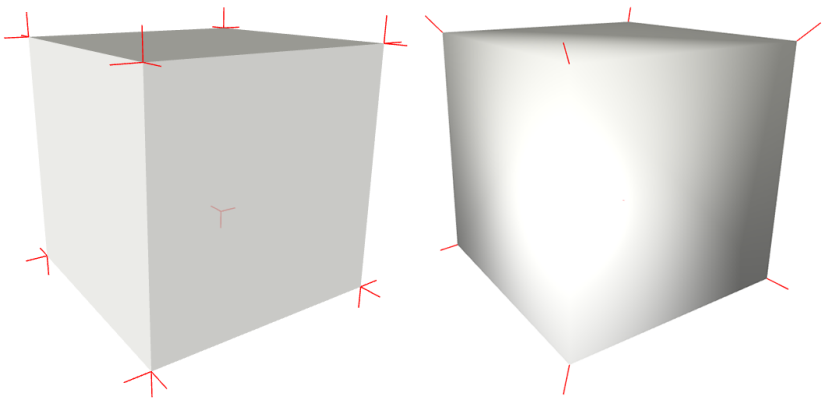

Figure 26: A comparison of a typical box mesh (left) and a box that was generated with Dual Contouring. The red lines represent the vertex normals.

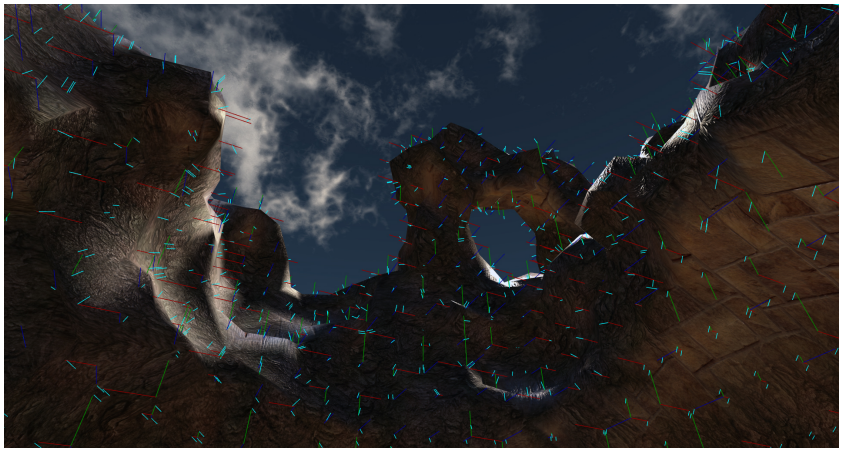

Figure 27: An exemplary terrain that has been created with the developed engine. The Hermite data is visualised in the form of coloured edges. The short blue lines represent the normals of the implicit surface and originate from the Zero Crossing position.

drawback for smooth shading of meshes with very sharp edges. While it would be possible to duplicate vertices during the polygon creation to directly use the actual intersection normals instead of a single averaged one, it wouldn't be trivial to implement such a workaround. Primal methods like CMS, on the other hand, might have an advantage in this regard as they create facets inside of voxel cells without reusing vertices from neighbouring cells.

An exemplary terrain that has been created with the developed engine is shown in Figure 27. Additionally, the Hermite data that describes the terrain's volume is visualised in the form of coloured edges. Red edges are aligned with the X-axis, green edges with the Y-axis and blue edges with the Z-axis. The short blue lines represent the normals of the volume and they originate from the Zero Crossing position. Material grid points are not explicitly shown but the relevant material indices are implicitly visualised through the edges.

4.10 Multithreading

Dedicated Web Workers are used to execute modifications and extractions for individual volume chunks in parallel. The amount of active worker threads is limited by the number of logical processors which can be queried from the client's browser. Without multithreading, the terrain engine would have to execute volume modifications and surface extractions on the main thread which would cause the system to freeze frequently. The use of Web Workers is crucial for the performance of the engine because it has to process large amounts of data which is bound to be computationally expensive. However, the use of Web Workers also comes with a few quirks that need to be accounted for.

The Web Workers API defines an asynchronous message protocol that is used

to send data from the main thread to a worker and vice versa. For the sake of thread-safety, the data is automatically copied using the structured clone algorithm which is similar to JSON. Consequently, sending large amounts of data is fairly slow. However, objects that implement the Transferable interface can be moved from one execution context to another with a zero-copy operation. Typed arrays implement this interface and can therefore be transferred efficiently.

Besides the performance advantage of using typed arrays, the ability to transfer data ownership is the main reason why the volume data needs to be stored in typed arrays. Storing edge data in a hash map would probably be more convenient, but the performance penalty of copying the entire data for every task would be too high.

A problem arises for volume modifications as SDFs define a sampling method which cannot be copied by the structured clone algorithm. Therefore, the SDFs needs to be serialised on the main thread and then revived by the worker. As a result, there is currently no support for custom SDFs. This could be improved by introducing another way for SDFs to define their sampling logic through data which could then be send to the worker.

4.11 The Engine Update Cycle

The terrain must manually be updated with the observing camera, usually every frame, to find volume chunks that intersect with the camera's frustum. The update cycle actively advances task scheduling and represents the driving mechanism of the engine. The main activities of the engine are shown in Figure 28.

During the update process, an internal scheduler checks whether there are any scheduled tasks for the chunks in the field of view that are not currently being processed by a worker. In case a task exists for a given chunk, the priority of the task is examined. Modifications take precedence and therefore have the highest priority which means that nothing needs to be done if such a task is at hand. When the task has a lower priority or if there is no task at all, the chunk is checked for pending CSG operations and if there are any, a new modification task is scheduled for that chunk. Failing this, the chunk's volume data is examined closer. In case the chunk contains data that is not full, the current LOD value of the chunk is calculated based on its distance to the camera and a new extraction task is scheduled if the new LOD differs from the previous value.

Every time a new task is scheduled inside of the update cycle, the engine attempts to execute a pending task immediately after. For this, a worker thread is requested from a thread pool and the next task is requested from the scheduler. Due to the fact that the scheduler itself acts like a priority queue, any task may be returned, but it's safe to assume that the order in which the tasks are returned corresponds to the chronological order in which they were added.

Modification tasks are executed by polling an SDF from the associated chunk's CSG operation queue and then sending the chunk and the SDF to the worker.

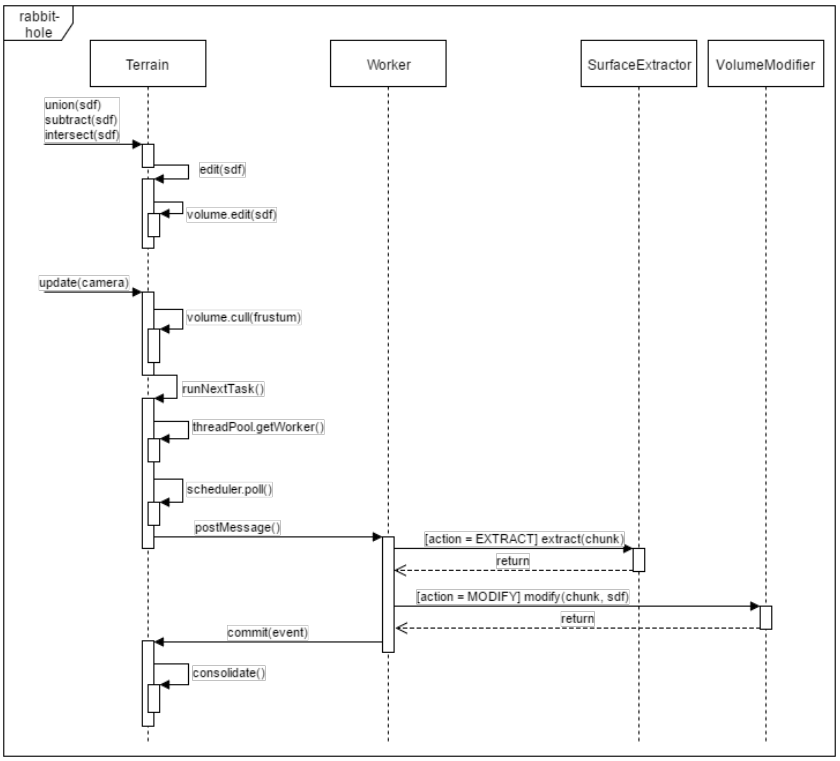

Figure 28: The engine activities.

Note that this removes the task from the scheduler albeit the CSG queue of the chunk may contain more items. Assuming the queue has not been drained yet, the next SDF will be polled from the chunk's operation queue as soon as the worker has finished the modification task which means that there is no delay. In this manner, a new task can internally be scheduled independently from the active update cycle and other tasks won't get blocked. As a result, the execution of tasks is not bound to the update interval and is only limited by the execution speed of the worker threads.

Extraction tasks can't be partly completed and are therefore easier to handle than modification tasks. Granted that there are no pending modification tasks left, the scheduler will return extraction tasks in descending order of priority. The priority of extraction tasks is based on the respective chunk's LOD value; a smaller value results in a higher priority. After a worker has finished the extraction task, the generated surface mesh is added to the terrain.

Due to the fact that the main purpose of the manual update is to focus processing power on volume chunks that lie in the given camera's field of view, other volume chunks will not be updated. This may cause issues when multiple cameras are used to create certain effects like a mirror or water reflection. In such

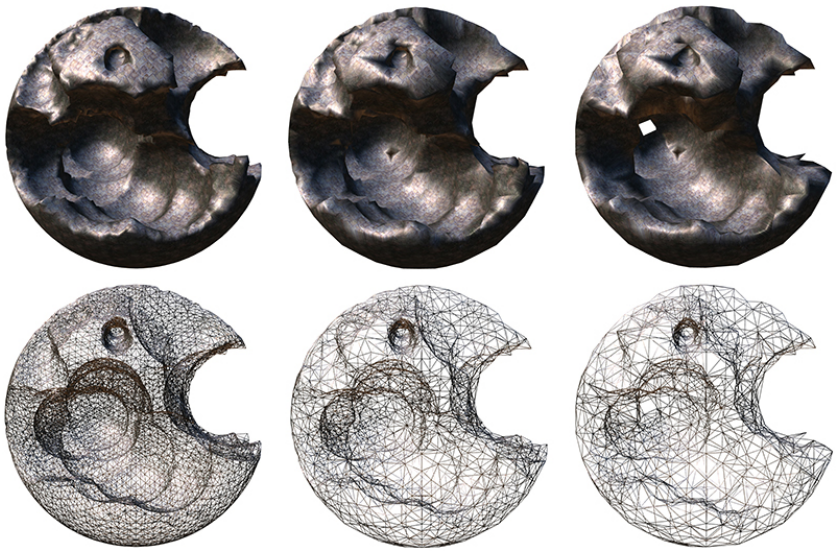

Figure 29: Isosurface extraction using different LODs. The three meshes have been extracted from the same volume data. The respective wireframe mesh is shown underneath each model. LODs from left to right: 0, 1 and 2.

cases there is no need to perform multiple manual updates for each camera. Instead, all cameras can collectively be passed to the update method. The camera frustums will then be combined for the volume octree culling.

4.12 Level of Detail

Rendering a large terrain can be a very expensive task and one way to improve the performance is to reduce the amount of vertices. The farther away an object is from the viewer, the less vertices are needed to sufficiently describe its shape. The terrain engine supports LOD by automatically reducing the amount of generated vertices for volume chunks that are farther away from the camera position. Listing 7 shows how the distance-based LOD value is calculated for each chunk.

```
distance = chunk.distanceToPoint(camera.position);

lod = Math.min(
    Math.trunc((distance / camera.far) * levels),
    levels - 1
);
```

Listing 7: A linear LOD calculation.

Note that the distance from the camera to a chunk has to be zero if the camera is inside of it. Otherwise, the LOD values are sometimes incorrect and cause undesired mesh simplifications. Calculating the distance to the centre of the chunk is therefore not advised.

The LOD value of a chunk is used during the simplification of the voxel octree. While the main purpose of the simplification is to eliminate unnecessarily high detail, the process can also be used to simplify the mesh more aggressively by raising the QEF error threshold. This allows more voxel cells to collapse which results in a reduced vertex count. Alternatively, the octree could be simplified to a specific octree depth while ignoring the QEF error altogether, but this would destroy many surface features indiscriminately. In the current implementation, the LOD value is raised to the third power and then added to the base error threshold. This simple method results in a very effective vertex count reduction for distant chunks while chunks that are close to the camera maintain their full detail.

An example of the LOD mechanism is shown in figure 29. The three depicted meshes have been extracted from the same volume data, but were using different LODs. Notice how the detail decreases from left to right with the LOD values 0, 1 and finally 2 where 0 represents the finest detail. The respective wireframe mesh is shown underneath each model to better illustrate the effect of the simplification. Table 6 shows the vertex and face count for the mesh at each LOD. It can be seen that the simplification process managed to reduce the amount of vertices significantly from LOD 0 to LOD 2 while keeping its shape intact.

Table 6: Vertices and faces for three different levels of detail.

	LOD0	**LOD1**	**LOD2**
Vertices	61266	15474	8322
Faces	20422	5158	2774

The terrain engine allows any number of detail levels to be defined which are then distributed evenly across the view distance. By default, the number of detail levels is $\log_2 n$ where n is the resolution of the volume chunks.

Since the volume data is maintained in chunks, the generated terrain also consists of multiple adjacent meshes. Splitting up large objects is usually beneficial for performance because the parts of the object that aren't in the field of view can be culled. However, too many meshes result in reduced performance due to an increased number of draw calls that often submit too little data to the GPU.

There's certainly more than one way to implement a LOD system. For example, the VoxelFarm engine uses a linear octree representation which allows the selection of octants based on a 64-bit integer key that consists of a computed LOD value and 3D coordinates. Higher LOD values select parent nodes which

Figure 30: Stretched textures on steep terrain geometry in the game Tera.

then sample the volume data of their child octants at a lower resolution. Unfortunately, JavaScript doesn't have 64-bit integers, but it might still be possible to implement a similar system in the future.

4.13 Tri-Planar Texture Mapping

The polygonal meshes that are generated by the surface extraction process have no texture coordinates. These coordinates are commonly known as UV coordinates and describe how textures should be projected onto the surface of a three-dimensional model. In the absence of texture coordinates, "it is common to simply drop the z coordinate and scale the (x, y) coordinates" (Lengyel 2010, p. 46). Alternatively, the y and z coordinates may be swapped. Although this approach is often used for heightmap-based terrain, it is not a robust texturing technique since it causes severe texture distortion on steep geometry as shown in Figure 30.

"A simple way to resolve this is to use triplanar texturing, or three different planar projections, one along each of the three primary axes (x, y, and z). At any given point, we use the projection that offers the least distortion (stretching) at that point" (Nguyen 2007). In the fragment shader, the three planar projections can be obtained by using the world position of the fragment to sample the texture. For the X-axis projection, the (y, z) coordinates are used. For the Y-axis projection, (z, x) is used and for the Z-axis projection, (x, y) is used. The three samples are then blended together based on the surface normal to obtain the final texture blend.

It would be optimal if the terrain mesh could use a slightly modified version of one of the built-in materials that Three.js has to offer. The material would have to be changed so that tri-planar texture mapping is used where textures are

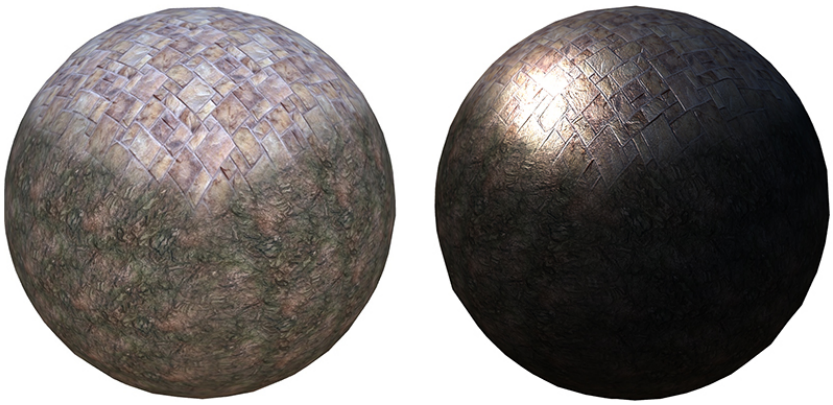

Figure 31: Tri-planar normal mapping. The sphere on the right uses tri-planar normal mapping while the sphere on the left doesn't.

sampled with UV coordinates. However, extending a built-in material in Three.js is rather difficult and causes various problems due to the fact that the library treats these materials as special cases.

Three.js uses the concept of shader chunks to construct its built-in materials. These shader code snippets can also be used by custom materials. Thus, a custom material is created that uses the same shader code as one of the library's built-in materials. The advantage of extending a built-in material this way is that lighting calculations, fog and other features are already implemented. Shader chunks that rely on UV coordinates for texture sampling are copied and modified to use tri-planar texture mapping. By registering these modified chunks as additional shader chunks, they can be used by the custom material.

The tri-planar texture mapping implementation used in the terrain engine supports separate textures for the X-, Y- and Z-axis projection. This allows the user to set a specific texture for steep walls and another texture for the floor. Furthermore, the shader supports normal mapping as shown in Figure 31.

The material information that is stored in the material index grid is currently not used by the shader due to the fact that WebGL has rather limited support for textures. However, WebGL 2 will introduce texture arrays that could be used to sample one of many textures based on the material index associated with a vertex.

4.14 Performance

The performance of volume modifications has been tested with the same example scene that has been used in Section 4.8. In this scene, a large torus with an inner radius of 200 and a tube radius of 8 occupies a total of 152 volume chunks which all have a resolution of 64. The AABB of this implicit surface, however, causes 2043 volume chunks to be created which all need to be fully processed to

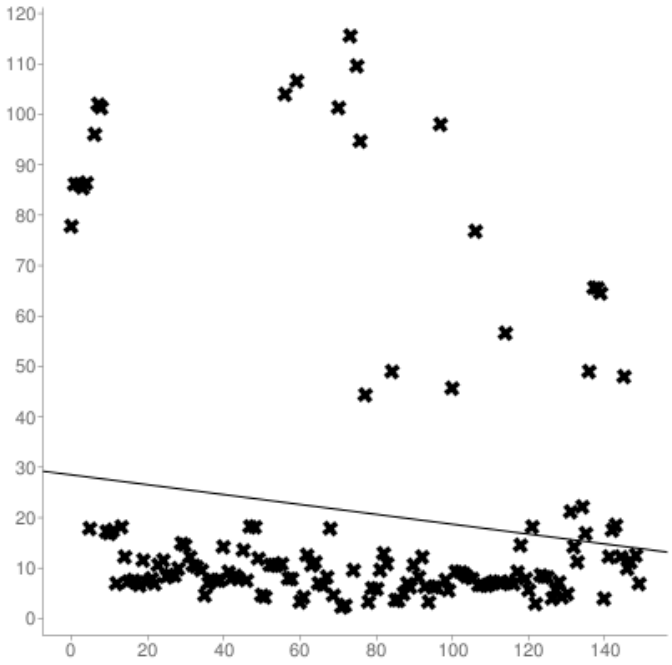

Figure 32: Execution times of volume modifications.

check if they actually contain the surface.

Table 32 shows the measured execution times of the first 150 volume modifications in milliseconds. With a mean time of 21.19 milliseconds, the volume modification process operates quickly. Notice how the time needed for a modification task slowly decreases the more tasks have been run. This is due to the fact that the web browser that was used in this test optimises JavaScript programs during runtime. The longest modification took approximately 115 milliseconds which is still rather fast considering the amount of data that has to be processed.

The execution times of the surface extractions for the same example scene are shown in Table 33. All surface extractions were run after the modifications. With a mean time of 62.66 milliseconds, the surface extractions are approximately three times slower than the volume modifications. Again, the measured execution times decrease as more tasks are executed due to the browser's runtime optimisations. The longest surface extraction took approximately 343 milliseconds which isn't fast but can still be considered reasonable.

All performance tests have been conducted on an Asus laptop with an Intel Core i7-3630QM CPU running at 2.4Ghz and 8GB RAM. Google Chrome Version 55.0.2883.87 m (64-bit) has been used to run the tests.

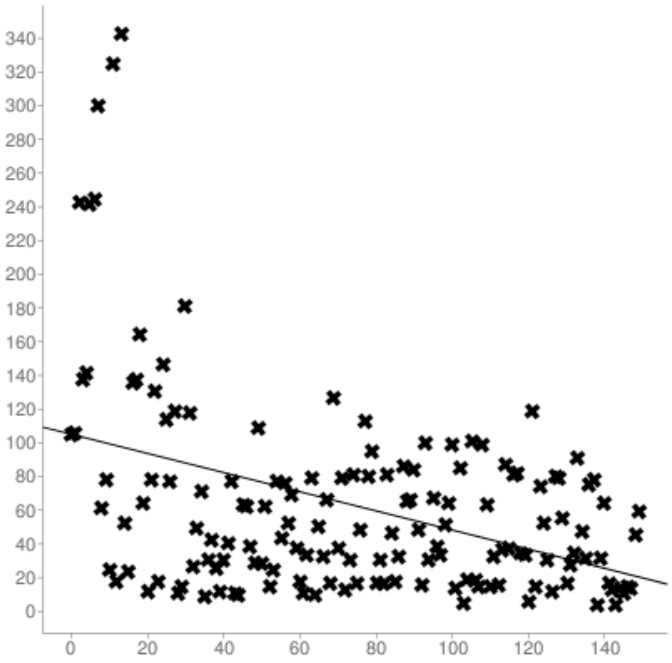

Figure 33: Execution times of surface extractions.

5 Discussion

5.1 Conclusion

The implementation of a terrain engine with JavaScript and WebGL was a success. The final system allows dynamic terrain modifications in real-time and manages large amounts of volume data in a multithreaded fashion. Volume modifications can easily be executed through a CSG interface that has been designed with simplicity and efficiency in mind. The generated meshes are textured using a tri-planar texture mapping shader that also supports normal mapping.

Furthermore, the engine can be used on mobile devices as long as WebGL and the Web Worker API is supported. It could also be shown that the performance of the volumetric terrain solution is indeed feasible both in terms of computational load and in terms of memory consumption. Since the terrain engine allows the user to manually set the resolution of the volume data, it's also possible to trade detail for performance. It should also be noted that the developed software is still in an early stage and that there is a lot of room for improvement.

Octree raycasting enhances terrain mesh picking and provides the base for efficient terrain editing. Moreover, the engine runs in the web browser with no setup required. Additionally, a minimalistic terrain editor has been built on top of the engine. Figure 35 shows a terrain that has been created with that editor.

The source code of the project can be found on GitHub:

https://github.com/vanruesc/rabbit-hole

5.2 Challenges

Volume data is stored in chunks because of its potentially infinite size. Chunks can be maintained in a structured and organised way, but they introduce a problem that is difficult to solve. Since the chunks of volume data represent a collective volume, there exists a strong dependency between adjacent chunks that becomes apparent after extracting the isosurface from a chunk. Performing the extraction on a single chunk in isolation means that the generated surface will be separated from the rest of the volume's surface.

Figure 34 shows a box that occupies eight adjacent volume chunks, causing gaps in the generated mesh. The DC algorithm can tie voxel cells of any size together, but keeping volume data in chunks always causes gaps between generated meshes. Although the chunks share their faces with their neighbours, the DC algorithm never gets the chance to connect the voxel cells with those from the neighbouring chunks. Even primal methods face this problem.

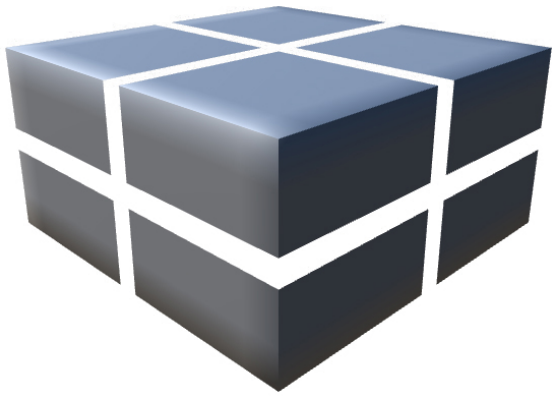

Figure 34: A box that occupies eight adjacent volume chunks, causing gaps in the generated mesh.

It wasn't possible to find a satisfying solution to this problem during the time of this project. However, it's safe to say that the issue lies with the presentation of volume data and that it is not a problem with the contouring algorithm. Finding a sound solution to the problem is a challenge that is left for the future.

5.3 Ethics

The presented terrain rendering engine doesn't extend the preceding implementations in terms of ethical questions. Thus, it seems unlikely that the implemented engine will become the subject of an ethical dispute. The replication of natural landscapes, planets or even galaxies is already possible to a certain degree as can be seen in recent games like No Man's Sky.

5.4 Future Work

The terrain engine currently uses the DC algorithm for the isosurface extraction process and produces polygonal meshes with geometrical errors. Even though the results are acceptable, switching to a better extraction algorithm such as CMS would be an improvement. Another aspect of the current system that can be improved is the calculation of surface normals at edge intersection points; instead of using a finite difference method, analytical derivation could be used which would yield accurate normals. Although it could be shown that volume data can be compressed very effectively, it still uses a lot of memory if the terrain is large. Most of the volume data remains in memory even if it is unused for long periods of time. Thus, it would be a good idea to store unused data persistently until it's needed again. Inside the browser, this can be achieved with the IndexedDB API which is also available in web workers.

With a robust terrain editor, the workflow of editing volumetric terrain could be compared to the traditional heightmap-based approach where additional 3D objects are used to create more complex terrain features. Furthermore, advan-

tages and challenges of using a Leap Motion controller for terrain editing could be examined.

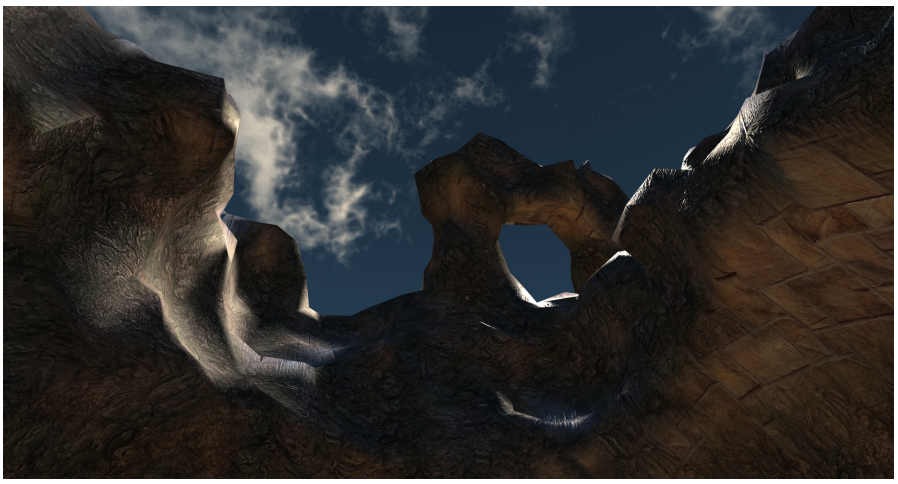

Figure 35: An exemplary terrain that has been created with the engine.

Bibliography

Boesch, F. (2016), 'Webgl stats', http://webglstats.com/ (Visited 20.12.2016).

Bullfrog Productions (1999), 'Dungeon Keeper 2', [PC CD-ROM]. Video Game.

Cepero, M. (2016*a*), 'Procedural world', http://procworld.blogspot.de/ (Visited 20.12.2016).

Cepero, M. (2016*b*), 'Voxel farm engine - reference', http://docs.voxelfarm.com/reference (Visited 20.12.2016).

Eich, B. (2015), 'Ecmascript harmony: Rise of the compilers', https://brendaneich.com (Visited 20.12.2016).

Ericson, C. (2004), *Real-time collision detection*, CRC Press.

Gargantini, I. (1982), 'An effective way to represent quadtrees', *Communications of the ACM* **25**, 905–910.

Geier, D. (2014*a*), 'Advanced octrees 2: node representations', https://geidav.wordpress.com/2014/08/18/advanced-octrees-2-node-representations (Visited 20.12.2016).

Geier, D. (2014*b*), 'Advanced octrees 3: non-static octrees', https://geidav.wordpress.com/2014/11/18/advanced-octrees-3-non-static-octrees/ (Visited 20.12.2016).

Gibson, S. F. F. (1999), 'Constrained elastic surfacenets: Generating smooth models from binary segmented data', *TR99* **24**.

Gildea, N. (2014*a*), 'Dual contouring: Seams & lod for chunked terrain', http://ngildea.blogspot.no/2014/09/dual-contouring-chunked-terrain.html (Visited 20.12.2016).

Gildea, N. (2014*b*), 'Implementing dual contouring', http://ngildea.blogspot.no/2014/11/implementing-dual-contouring.html (Visited 20.12.2016).

Hamming, R. (2012), *Numerical methods for scientists and engineers*, Courier Corporation.

Hello Games (2016), 'No Man's Sky', http://www.no-mans-sky.com (Visited 20.12.2016). Video Game.

Ho, C., Wu, F.-C., Chen, B.-Y., Chuang, Y.-Y., Ouhyoung, M. et al. (2005), Cubical marching squares: Adaptive feature preserving surface extraction from volume data, *in* 'Computer graphics forum', Vol. 24, Wiley Online Library, pp. 537–545.

Ju, T., Losasso, F., Schaefer, S. & Warren, J. (2002), Dual contouring of hermite data, *in* 'ACM Transactions on Graphics (TOG)', Vol. 21, ACM, pp. 339–346.

Ju, T. & Udeshi, T. (2006), Intersection-free contouring on an octree grid, *in* 'Proceedings of the 14th Pacific Conference on Computer Graphics and Applications', Vol. 3.

Khronos Group (2016), 'Vulkan', https://www.khronos.org/vulkan (Visited 20.12.2016).

Kobbelt, L. P., Botsch, M., Schwanecke, U. & Seidel, H.-P. (2001), Feature sensitive surface extraction from volume data, *in* 'Proceedings of the 28th annual conference on Computer graphics and interactive techniques', ACM, pp. 57–66.

Lengyel, E. S. (2010), Voxel-based terrain for real-time virtual simulations, PhD thesis, Citeseer.

Lorensen, W. E. & Cline, H. E. (1987), Marching cubes: A high resolution 3d surface construction algorithm, *in* 'ACM siggraph computer graphics', Vol. 21, ACM, pp. 163–169.

Lysenko, M. (2012), '0 fps', https://0fps.net/category/programming/voxels/ (Visited 20.12.2016).

Mojang (2011), 'Minecraft', https://minecraft.net (Visited 20.12.2016). Video Game.

NASA (2016), 'Visible earth: Topography', http://visibleearth.nasa.gov/view.php?id=73934 (Visited 20.12.2016).

Nguyen, H. (2007), *Gpu gems 3*, Addison-Wesley Professional.

Osher, S. & Fedkiw, R. (2006), *Level set methods and dynamic implicit surfaces*, Vol. 153, Springer Science & Business Media.

Rassovsky, G. (2014), Cubical Marching Squares Implementation, PhD thesis, Bournemouth University.

Requicha, A. A. & Voelcker, H. B. (1977), 'Constructive solid geometry'.

Revelles, J., Urena, C. & Lastra, M. (2000), 'An efficient parametric algorithm for octree traversal'.

Salomon, D. (2004), *Data compression: the complete reference*, Springer Science & Business Media.

Schaefer, S., Ju, T. & Warren, J. (2007), 'Manifold dual contouring', *IEEE Transactions on Visualization and Computer Graphics* **13**(3), 610–619.

Schaefer, S. & Warren, J. (2004), Dual marching cubes: Primal contouring of dual grids, *in* 'Computer Graphics and Applications, 2004. PG 2004. Proceedings. 12th Pacific Conference on', IEEE, pp. 70–76.

Stichting Blender Foundation (2005), 'Blender documentation volume i - user guide: Chapter 7. advanced mesh modelling', http://www.ru.is/kennarar/hannes/useful/BlenderManual/htmlI/ch07.html (Visited 20.12.2016).

Tavares, G. (2016), 'Webgl2fundamentals', http://webgl2fundamentals.org/webgl/lessons/webgl2-whats-new.html (Visited 20.12.2016).

Trettner, P. (2013), 'Terrain engine part 2 - volume generation and the csg tree', https://upvoid.com/devblog/2013/07/terrain-engine-part-2-volume-generation-and-the-csg-tree (Visited 20.12.2016).

Tzur, R. (2003), 'Contouring implicit surfaces', https://www.sandboxie.com/misc/isosurf/isosurfaces.html (Visited 20.12.2016).

Unknown Worlds Entertainment (2016), 'Subnautica', http://unknownworlds.com/subnautica (Visited 20.12.2016). Video Game.

Glossary

Babylon.js A JavaScript game engine.

Broccoli A JavaScript task runner.

CommonJS Defines a module format for JavaScript.

csg.js A JavaScript library that implements CSG operations on meshes.

GitHub A Git repository hosting service.

Goo Create A JavaScript game engine.

Google Chrome A web browser by Google.

Gradle A Java task runner.

Grunt A JavaScript task runner.

Gulp A JavaScript task runner.

heightmap A raster image used to store surface elevation data.

IndexedDB A browser interface for client-side storage of large amounts of data.

isosurface A surface that represents the contour of an implicit surface.

isovalue A constant that denotes the boundary of an implicit surface.

JavaScript A high-level, dynamic, untyped programming language.

Leap Motion A sensor device that uses hand and finger movements as input.

Maven A Java task runner.

Node.js An asynchronous, event-driven JavaScript runtime.

OpenCSG The CSG rendering library.

Persistence of Vision Raytracer A 3D graphics ray tracing tool.

PlayCanvas A JavaScript game engine.

Rollup A JavaScript module bundler.

Scene.js A JavaScript rendering framework.

Three.js A JavaScript rendering framework.

Transvoxel An algorithm for stitching together neighbouring triangle meshes.

Unity3D A 3D game engine.

Unreal Engine A suite of game development tools.

VoxelFarm A commercial 3D volumetric content engine.

Vulkan A new generation graphics and compute API.

Web Worker A browser interface for JavaScript multithreading.

WebGL The 3D graphics API of modern web browsers.

YUIDoc An API documentation generator.

Acronyms

AABB Axis-Aligned Bounding Box.

Ajax Asynchronous JavaScript and XML.

API Application Programming Interface.

Blob Binary large object.

BSP Binary Space Partitioning.

CMS Cubical Marching Squares.

CPU Central Processing Unit.

CSG Constructive Solid Geometry.

DC Dual Contouring.

DMC Dual Marching Cubes.

DOM Document Object Model.

ECMA European Computer Manufacturers Association.

EMC Extended Marching Cubes.

ERASMUS European Region Action Scheme for the Mobility of University Students.

ES2015 ECMAScript 6th Edition.

FIFO First in, first out.

GLSL OpenGL Shading Language.

GPGPU General-purpose computing on Graphics Processing Units.

GPU Graphics Processing Unit.

HTML HyperText Markup Language.

JSON JavaScript Object Notation.

LOD Level of Detail.

MC Marching Cubes.

MS Marching Squares.

MT Marching Tetrahedra.

NASA National Aeronautics and Space Administration.

NPM Node Package Manager.

NTNU Norwegian University of Science and Technology.

OpenGL Open Graphics Library.

OpenGL ES OpenGL for Embedded Systems.

QEF Quadratic Error Function.

RLE Run-Length Encoding.

SDF Signed Distance Function.

SIMD Single Instruction Multiple Data.

SN Surface Nets.

THB Brandenburg University of Applied Sciences.

URL Uniform Resource Locator.