Volumetric Terrain Rendering with WebGL

Raoul van Rüschen[1], Simon McCallum[2], Stefan Kim[1], and Reiner Creutzburg[1]

[1]Technische Hochschule Brandenburg, Department of Informatics and Media, P.O.Box 2132, D-14737 Brandenburg, Germany
[2]Norwegian University of Science and Technology, Department of Computer Science, P.O.Box 191, NO-2802 Gjøvik, Norway

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 more sophisticated solutions becomes 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 the application of contouring methods in a real-time environment where the volume frequently changes through user interaction. The final system uses the Dual Contouring surface extraction technique and maintains discrete volume data in adjacent cells 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.

Introduction

WebGL is a young and exciting technology used for the creation of web browser games and animations. A central part of many games is terrain and traditional approaches use heightmaps to elevate the vertices of a regularly subdivided plane mesh. 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. Modern terrain implementations use advanced algorithms to construct a surface based on volume data. Such a volumetric solution can replace the heightmap approach entirely and provides much more freedom, but it inherently requires more memory as it operates on 3D data instead of 2D textures. Level of Detail (LOD) algorithms for this type of terrain are also more involved than the heightmap variants depending on the chosen mesh construction technique.

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 feasible 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 work 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 inside the browser such as the management of large amounts of volume data and the efficient and dynamic construction of the terrain mesh during runtime.

Contributions

This work provides the following contributions that can serve as a basis for future research:

1. A fully documented, open-source terrain engine implementation in JavaScript, including the Dual Contouring (DC) algorithm and a Quadratic Error Function (QEF) solver.
2. An octree-based space partitioning solution that can be used to manage large amounts of volume data.
3. Multithreaded Constructive Solid Geometry (CSG) for discrete volume data modifications based on Signed Distance Functions (SDFs).
4. Performance measurements for memory consumption and processing times.

Related Work

Most of the related literature focuses more on the characteristics of existing or newly developed volume contouring techniques and less on the practical application in real-time environments. [1] describes a voxel-based terrain visualisation system that relies on ray tracing for rendering. [2] describes a voxel-based occlusion technique for improved heightmap-based terrain rendering. [3] describes an alternative solution for the calculation of feature points for the DC technique in the form of “particle-based feature approximation”. This approach replaces the QEF calculations and trades accuracy for performance while also making it easier to perform the contouring process on the GPU. [4] presents a practical implementation of the Marching Cubes (MC) contouring technique that is designed specifically for voxel-based terrain in real-time desktop games. An important contribution of the latter is the Transvoxel algorithm that extends the MC technique with LOD capabilities. [5] provides an alternative extension to the MC technique that uses longest edge bisection (LEB) to support LOD. [6] is the most recent related work that broaches the issue of applying the DC technique to a set of volume chunks in real-time while also presenting a possible seam patching solution.

There are a few mentionable web blogs that provide inspiring information on the topic of volume-based terrain systems: [7] published a series of articles about the commercial Voxel Farm Engine. Many of the insights from these articles have influenced the design decisions in the present work. [8] talks about the implementation of DC and provides concrete information on how seams between multiple adjacent cells of volume data can be handled while [9] explains volume generation in more detail. Furthermore, [10] provides basic JavaScript implementations of MC, Marching Tetrahedra (MT) and Surface Nets (SN) with a comparison of the three techniques in terms of performance and polygon counts.

Advanced Web Browser Capabilities

Modern web-browsers incorporate advanced technologies that can be exploited for high-performance 3D games today. While WebGL grants access to the Graphics Processing Unit (GPU) and provides the means to implement hardware-accelerated 3D animations, the Web Worker API enables true multithreading inside the browser. These features are available through JavaScript, the dynamically typed high-level programming language of the web which runs in a virtual machine, uses Just-in-time (JIT) compilation and relies on automatic garbage collection.

Since JavaScript engines have been heavily optimised over the past decades, the execution speed of JavaScript has come very close to native performance. Although JavaScript is still slower than native applications when it comes to demanding tasks like physics simulations, it's already fast enough to build rich, interactive animations and 3D games. According to [11], the creator of the language, future versions of JavaScript will further address the performance issues of JavaScript 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 will be released faster to allow browser vendors to implement the new features quicker.

WebAssembly (WASM) is a new experimental feature that was inspired by the Emscripten project and the JavaScript subset asm.js. It provides a way to compile C++ code to an assembly-like language which can be run at near native speed in the browser. Game engines such as Unity and Unreal have relied on this technology early on to support HTML5 as a target platform. WASM is designed to run alongside JavaScript to run performance sensitive code. The terrain engine presented in this work currently doesn't use WASM as there are still some open issues with this feature. One could try to replace crucial parts of the current system with compiled WASM modules, but this goes beyond the scope of this project.

Preliminaries

According to [12], a Signed Distance Function 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.

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" [13]. In the context of implicit surfaces, the methodology is used to combine SDFs into complex descriptions of volumes. Figure 1 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" [13]. 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 highly complex solids using CSG. Another project that uses this methodology is OpenCSG which 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.

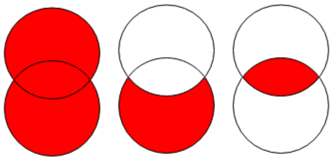

Figure 1. *The effect of CSG operations from left to right: Union, Difference and Intersection.*

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 on 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 processed and visualised efficiently. Various isosurface contouring techniques exist that are closely related, but perform the conversion in different ways.

Contouring Techniques

One of the oldest and most prominent isosurface contouring techniques is the MC algorithm published by [14]. It translates the continuous values of an SDF into a discrete grid of uniformly distributed material indices. This 3D grid is subdivided into voxel cells. As the name of the technique suggests, MC marches over these cubic cells and evaluates the SDF at 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 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. In a final step, all polygons are tied together. Figure 2 shows a voxel cell with an exemplary material configuration and the generated triangles next to it. The MC extraction method is not without flaws: it often produces degenerate triangles and can't preserve sharp features. Furthermore, the algorithm doesn't support LOD in its basic form.

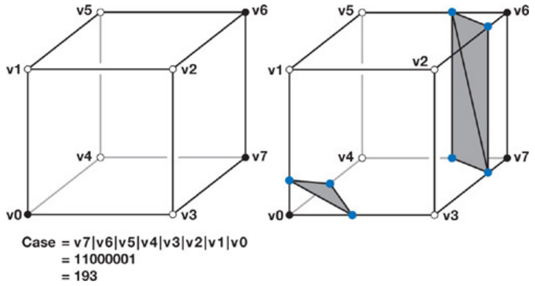

Figure 2. *A Marching Cubes voxel cell. The materials at the eight corners are known and determine the case at hand; the right picture shows the generated polygons. Source: [15]*

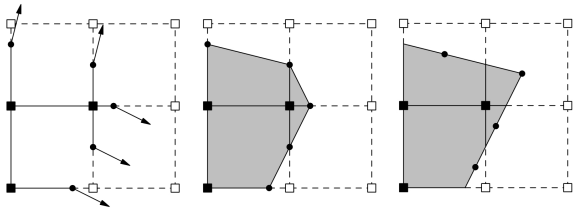

Figure 3. A comparison of triangle generation with Marching Cubes (centre) and Dual Contouring (right) in 2D. MC relies on the material indices and surface intersection points while DC also considers the intersection normals. Source: [17]

A solution to the LOD problem was presented by [4] 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 [16] introduces a mechanism for sharp feature preservation with 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 evaluated 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.

Following the idea of preserving distinct details of the volume's surface, [17] published the straight-forward DC technique that takes after the approach of SN presented by [18] 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 voxel 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 much additional effort, because it allows voxels to be of any size.

Figure 3 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, [19] 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. [20] addressed the issue by presenting the Manifold Dual Contouring (MDC) algorithm that allows multiple vertices per voxel cell and implements a basic criterion for vertex clustering which, however, results in a slight increase in computational complexity. In an earlier publication, [21] also presented the Dual Marching Cubes (DMC) algorithm which introduced the concept of a dual grid for the preservation of sharp features using MC.

Cubical Marching Squares (CMS) is another unique contouring method presented by [22]. 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. [23] provides a partial implementation of the CMS contouring technique that proves the feasibility of the technique but lacks crucial features such as sharp edge preservation.

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. 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 is determined through sampling of the SDF at the respective grid point world positions. A grid point lies inside the volume and represents solid material if the SDF returns a density value $d \leq c$ where c is the isovalue.

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. Figure 4 depicts a 2D example grid with a resolution of 8 on the left side and a single 3D 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. In 3D, a 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.

While a high grid 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.

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

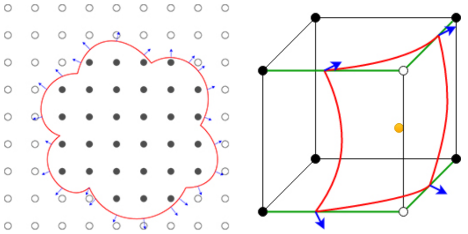

Figure 4. *A 2D example of Hermite data with a grid resolution of 8 (left) and a voxel cell that contains a segment of the original implicit surface.*

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" [24]. 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. 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 $1\mathrm{e}{-6}$. Lastly, a bias of $1\mathrm{e}{-2}$ 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 and smaller details will be lost if the grid resolution isn't high enough. When the intersection point is known, the surface normal can be approximated using a finite difference method or it can be calculated accurately using analytic derivation of the SDF.

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 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 single grid is not an option as this results in an oversimplified surface mesh. For that reason, the sampled volume data is kept in separate cells and the SDFs are discarded as soon as the Hermite data has been built.

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 and further. It's easy to see that this approach would quickly become impractical. The benefit of using multiple clustered containers to store the volume data is the ability to execute volume modifications as well as 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 in a progressive way. Furthermore, the partitioned volume can be culled to focus computation power on portions of the volume that are close to the viewer.

ECMAScript 6th Edition (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. Material indices are kept in a one-dimensional typed array of 8-bit unsigned integers and the global grid resolution can be any integer 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 grid 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 cell's Axis-Aligned Bounding Box (AABB). Consequently, the complete material grid needs to be available for modifications, resamplings 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. An appropriate data structure for this undertaking would be a hash map, but JavaScript hash maps can't efficiently be sent to 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. Edge data consists of edges, Zero Crossings and normals. Additionally, each of these three groups 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 respective axis 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 grid 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.

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 ray-casting, the octree data structure is a fundamental component of the DC algorithm where it maintains information about voxel adjacency. An octree's root octant may contain eight smaller octants. Each octant is an AABB that is exactly half the size of its parent node. Each child octant can also contain up to eight children. 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 octants has been found which contains 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 custom conditions.

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" [25]. 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 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 a linear world octree to organise volume data cells and pointer-based voxel octrees to extract isosurfaces from individual world cells on the fly.

World Octree

The concept of linear octrees stems from linear quadtrees that were first proposed by [26]. Linear octrees store all of their octants in a hash map. Individual octants can be identified by calculating a so called locational code. Since no pointers need to be stored, linear octrees require less memory than traditional octree. Furthermore, the tree hierarchy is encoded in the locational codes which allows intermediate octants to be omitted entirely as they can be reconstructed on demand. 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" [27]. Therefore, the extent of linear octrees should be static.

The terrain system uses a custom linear octree to efficiently organise a sparse, multilevel collection of volume data cells that maintain a description of a portion of the terrain at a fixed resolution. Note that the world distance between the grid points inside a cell describes its effective resolution. The world octree is axis-aligned, cannot be rotated and allows direct access to different LOD layers, octant neighbours and parents. It also differentiates between intermediate octants and leaf octants where intermediate octants store additional information about the existence of their potential children. The data of all octants may be modified at any time using CSG.

Each world octant can be uniquely identified by a 3D coordinate and a LOD value. A perfect hash function is used that packs the individual values for X, Y and Z into a unique 53-bit key. As a result, there will be no collisions in the underlying hash maps. This approach was inspired by the Voxel Farm engine [28]. In contrast to Voxel Farm which has native 64-bit integers at its disposal, JavaScript uses IEEE 754 binary64 Doubles for Numbers and, consequently, only supports 53-bit integers safely as of ES2017. Since 11 bits are unusable, the bit allotments for the key coordinates must be carefully chosen to maximise the effective number ranges. For that reason, the LOD value is not part of the key and is used to select a distinct octant grid instead in which the packed keys are unique. Figure 5 shows the default octant key design that uses 40% of the available bits to encode the X-coordinate, 20% for the Y-coordinate and the remaining 40% for the Z-coordinate.

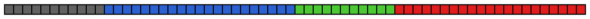

Figure 5. The default octant key design with 53 available bits. The red slots represent the amount of bits reserved for the X-coordinate, green represents Y and blue represents Z. The grey slots are unusable.

With the default bit allotments, it's possible to represent $2^{21} = 2097152$ distinct integers along the X- as well as the Z-axis and $2^{11} = 2048$ integers along the Y-axis. Assuming a base cell size of 20 metres, the full extent of the managed world space is $41943.04km \times 40.96km \times 41943.04km$. Furthermore, bit operations can only safely be applied to DWords (32-bit). This means that all 53-bit keys must be split into a high and a low part in order to retrieve the 3D coordinates from octant keys. Another pitfall is that JavaScript interprets the result of 32-bit operations as signed integers by default. In order to prevent the sign bit from being preserved, most bit operations must be followed by a zero-fill right shift. In the following example, the second variant is preferred:

$$((2^{32}-1) \mid 0) = -1$$

$$((2^{32}-1) \mid 0) \; >>> \; 0 = 4294967295$$

It's worth mentioning that a BigInt feature is going to be included in a future version of JavaScript that will allow the use of arbitrary precision integers. This could improve bit operation capabilities and may substantially increase the maximum possible world size.

The world octree provides a way to find an octant based on a given 3D position. Since negative 3D coordinates are allowed, the given position needs to be translated to the origin $(0,0,0)$. This yields zero-based unsigned coordinates that are then divided by the cell size of the target LOD grid which can be calculated by $s_0 \times 2^{lod}$ where s_0 is the base cell size of LOD zero. Octant key coordinates are all integers, so the fractional part of the coordinates must be truncated. The final key coordinates can then be packed into a unique octant key that is valid for the target LOD. Adjacent keys can easily be calculated since the coordinates are contiguous and normalised in that the step size between two keys is exactly 1 in any direction. The existence of the octant identified by the computed key is not guaranteed.

Table 1. A binary pattern lookup table that describes the eight relative octant position offsets.

	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

The amount of LOD grids is artificially limited to save resources which means that the world octree structure is not actually complete. The highest grid is always the starting point when any kind of search is performed and since there is no coarser grid above the highest one, there is also no meta information about the existence of the octants in that grid. Therefore, the hash map must be queried for every octant in question and empty space can't be skipped on that level. However, the octants of the highest LOD usually cover enough space to alleviate this concern. As stated earlier, each intermediate octant stores information about the existence of its immediate children that reside in the next lower grid. This information is available in the form of an 8-bit mask which corresponds to the octant layout specified in Table 1. The positional offset of any octant relative to its parent is the remainder after the division of the key coordinates by 2. The remainders r_x, r_y, r_z can be calculated using a fast bitwise modulo operation of the form $r = i \ \& \ 1$ because the key coordinates are positive integers and the divisor is a power of two. Note that this bit operation is safe to use as only the lowest bit of the operands is relevant. The index into the offset table can be obtained using a reversed packing order: $(r_x << 2) + (r_y << 1) + r_z$.

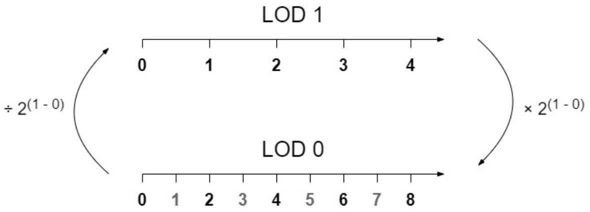

Figure 6. *Key translation between LODs. The calculations use integer arithmetic.*

Figure 6 illustrates the relationship between keys from different LOD grids. Translating key coordinates from any grid to a higher one requires a zero-fill right shift by $lod_{target} - lod_{current}$ bits while a translation into a lower grid requires a left shift by $lod_{current} - lod_{target}$ bits. Note that the key design disallows bit allotments that are greater than 32 bits because bit operations can currently only be applied to DWords. The coordinate translation to a higher LOD is safe for 32-bit values as long as the zero-fill right shift operator is used. A translation into a lower grid with a left shift will never occur if the current LOD is zero because there is no lower grid. So even if 32-bit coordinates are used, there will be no overflow.

Raycasting

Finding objects that intersect with a ray is called raycasting. This approach is also referred to as "picking" when the ray direction is controlled by the mouse cursor to actively select objects. Raycasting support is an important feature of the terrain system as it provides a way to quickly select a part of the terrain surface. The linear world octree implementation that is used in this project incorporates an efficient raycasting technique that was originally proposed by [29]. The algorithm capitalises on positional assumptions and therefore requires the octants to adhere to a common layout.

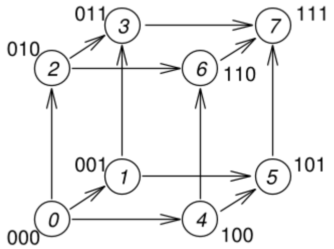

Figure 7. *The depicted octant layout is crucial for positional assumptions during raycasting.*

Figure 7 illustrates the expected octant positions from Table 1 which maps each position to a unique identifier. This layout dictates the linear order of the eight children of an octant based on their relative position. The chosen raycasting 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. For instance, raycasting a point cloud consisting of 1048576 spheres with a naive approach may take upwards of 65 milliseconds while the octree approach culls a large amount of candidates 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.

Since the world octree has a limited amount of LOD grids, a specialised hybrid approach is deployed that relies on a voxel traversal algorithm proposed by [30] to iterate over the octants of the highest grid along a 3D line. The octree traversal algorithm is then used to raycast the identified octants which serve as adequate subtrees. The voxel traversal implementation is a 3D supercover variant of the Digital Differential Analyzer (DDA) line algorithm and is similar to the well-known Bresenham algorithm. Furthermore, the octree traversal algorithm relies on octant child existence information to accelerate searches.

Volume Modification

Before any volume data can be generated, the world octree structure must be prepared so that it can accommodate the new data. Since it's not feasible to create all possible octants in advance, the octree needs to be able to constantly adapt to changes. At first, the world octree is empty, but its dimensions are statically defined through the octant key design. This information paves the way for a straight-forward volume expansion and contraction strategy. 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, world octants that don't exist will be created across all LOD grids. For the second case, it's only necessary to find existing world octants that may be affected by the volume reduction. All volume modifications are guided by the AABB of the given SDF which gets translated into an octant key range to iterate over the affected octants. None of the volume data will be modified immediately. Instead, the SDF will be added to the CSG queue of the identified octants. The invaluable advantage of letting every octant maintain its own independent CSG queue instead of using a single central queue is that it allows lazy modifications which are only executed when the octant is close to the viewer. Furthermore, the data of the affected intermediate octants can be generated when it's actually needed. World octants will only be removed from the octree if the result of a modification task turns out empty. After an octant has been removed, the octree will be pruned recursively to remove all parent octants that became empty.

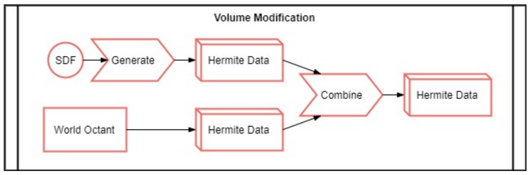

Figure 8. *An overview of the volume modification.*

An overview of the actual volume modification process can be seen in Figure 8. The input of the process consists of the SDF and an affected set of Hermite data. Each data set is modified in parallel and the result of every modification is another data set that 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 primitive solids: box, cylinder, cone, heightfield, noise, pill, sphere, pellet, torus, pipe and corridor. 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.

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	$A \cap B$	$\max(f,g)$

Recall that 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 [9] 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 the creation of complex implicit surfaces. The terrain system relies on sets of discrete Hermite data to describe its current volume and the existing data can be transformed using SDFs.

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. 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 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 world octant's boundaries.
2. Combine the generated data with the existing data according to the chosen CSG operation type.

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 CSG operation which contains the SDF. It's possible to rely on the simple mathematical approach to combine SDFs during the generation of volume data, but merging two sets of discrete Hermite data requires more complex combination logic. The generated material indices and edges describe the result of the given CSG operation in a form that is compatible with the existing 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.

With a time complexity of $\mathscr{O}(n^3)$, 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

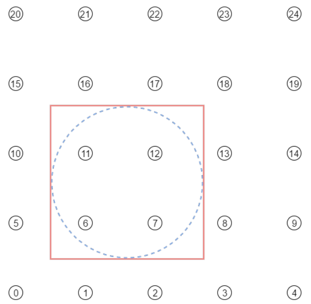

Figure 9. *Identification of affected grid points using the operation's reach. The implicit surface is depicted as a blue circle. The red rectangle represents the operation's AABB. In most cases, only a small subset of the grid points needs to be processed.*

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 9 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. Furthermore, the shape that is described by the SDF 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. The calculated bounds are used to iterate over a portion of the material grid. In this case, the AABB contains the grid points $\{6,7,11,12\}$ which translates to the lower bounds $(1,1)$ and the upper bounds $(2,2)$.

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

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

Adding the offset to the associated world octant'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 time complexity of $\mathcal{O}(3 \times n^3)$. In order to handle this task efficiently, a divide and conquer approach is used. Edge data is stored separately for each axis so that edges can first be 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. Due to the structure of the material grid, 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 9 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 using the Zero Crossing approximation to obtain the surface intersection data. 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 of the SDF and to avoid processing of non-existing edges at the grid borders.

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. In both cases, the remaining empty space can safely be cut off afterwards.

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 $\mathcal{O}(n)$. To provide a clear description of the actual combination process, a visual example is given for each CSG operation type.

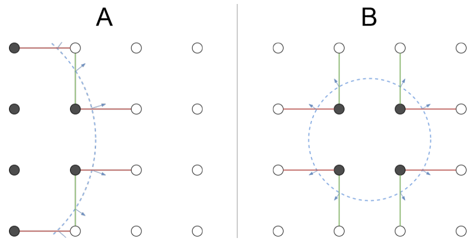

Figure 10. *Two exemplary sets of 2D Hermite data.*

Figure 10 shows two exemplary sets of Hermite data. Set A represents existing volume data while set B represents the predominant 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 depicted as blue arrows.

"For $A \cup B$, all non-air materials of B override the corresponding material in A" [9]. 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,

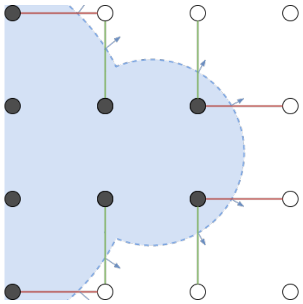

Figure 11. An example of a CSG Union operation.

all edges in A that no longer exhibit a material change are discarded. The effect of the Union operation can be seen in Figure 11. Notice how the green edges from set A have been selected instead of the conflicting edges from set B.

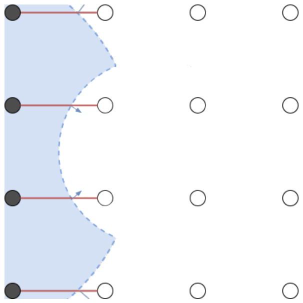

Figure 12. An example of a CSG Difference operation.

"For $A \setminus B$, all non-air materials of B result in air" [9]. 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" [9]. Additionally, the intersection normals of edges that were adopted from B must be inverted to keep the description of the surface consistent. Figure 12 shows the result of the Difference operation and demonstrates that the normals from B have been inverted.

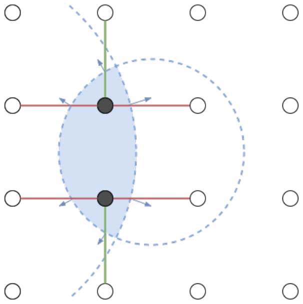

Figure 13. An example of a CSG Intersection operation.

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 13.

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 can be exploited to collect all relevant edges in one sweep. While iterating over the set of generated edges, the starting and ending grid point indices of each edge are inspected 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.

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 for an exemplary setup in which a complex SDF is captured by a single volume data cell with a material grid resolution of 64. The SDF describes a massive pipe with rounded edges that fully occupies the data cell.

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 stores meaningful information 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." [31]. 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 values $\{0,0,0,0,1,1\}$ would result in the compressed data $\{0,1\}$ plus the run-lengths $\{4,2\}$. Run-lengths are stored as 32-bit unsigned integers because 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 set 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 sets that are full outputs only one material index and one run-length that holds the total material index count. While empty sets can safely be discarded, full sets must be preserved as they still contain meaningful information.

With a resolution of 64, the data cell contains 274625 ma-

Table 3. Memory usage statistics for a single volume data cell that captures a complex SDF with a grid resolution of 64.

	Material Indices	Edges
Max. Count	274625	811200
Solid Materials	94760	-
Actual Count	9021	21528
Max. Mem. Usage	268.19 KB	15.47 MB
Actual Mem. Usage	44.05 KB	420.47 KB
Compression Ratio	6.09	37.68
Space Savings	83.58%	97.35%

terial indices. Assuming that material indices use 8 bits, the memory usage for the uncompressed material indices is roughly 268.19 KB. Table 3 shows that by applying the RLE algorithm, the amount of material indices in this example can be reduced to 9021 indices plus 9021 run-lengths. Since material indices use 8 bits and run-lengths 32 bits, each run-length value counts as four material indices. Together, this amounts to 45105 8-bit long values which is only 16.42% of the maximum material index count. The actual space requirement of the compressed data is roughly 44.05 KB which proves that RLE is well suited for the data at hand.

The maximum possible number of potential edges for a set with the same resolution is $3 \times (64+1)^2 \times 64 = 811200$. 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 is roughly 15.47 MB. Storing this much data for a single cell would quickly become a problem 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 this example is 21528 which is only 2.65% of the maximum amount. A data cell could only ever be fully populated with edges if it captured 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 420.47 KB for this example, the space requirement for edge data at this resolution can be considered manageable with no need for further compression.

Surface Extraction

Contrary to previous presentations of isosurface extraction techniques, the input of the terrain engine's extraction process is a single set of discrete Hermite data instead of an SDF. This means that the SDF doesn't need to be evaluated on the fly as the data is already available. Furthermore, the system uses the DC algorithm to create polygonal meshes from volume data. For this technique, the raw volume data needs to be converted into a Sparse Voxel Octree (SVO). All of the voxels are constructed on top of the material grid; the corner vertices of the cells match with the position of the grid points. Consequently, the material information and edge data is shared by adjacent voxel cells.

A voxel contains QEF data which is an accumulation of edge data. 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 voxel cell. Moreover, the generated feature point becomes a vertex of the final polygonal mesh and the respective vertex normal is the average of the involved surface intersection normals.

"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)" [32]. 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 [17], 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 in which case the solution falls back to the mass point of the voxel - the average of the intersection positions.

Apart from that, each voxel cell also contains one byte that stores the 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. 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, albeit costly, can reduce the vertex count significantly by preventing unnecessary tessellation. At last, the DC algorithm traverses the voxel octree to construct the polygonal mesh by connecting the vertices of adjacent voxel cells.

The process of building the voxel octree relies on the edge data to determine which voxel cells need to be created. Due to the way the edges are stored, the voxels are created 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 grid resolution and i the index of the edge's starting grid point. The coordinates x, y, z can then be calculated as follows:

$$x = i \mod (n+1)$$
$$y = \lfloor (i \mod (n+1)^2) \div (n+1) \rfloor$$
$$z = \lfloor i \div (n+1)^2 \rfloor$$

After translating the obtained coordinates into world positions, the Zero Crossing interpolation value of the edge is used to compute the actual intersection position along the edge. This position and the respective intersection normal are then added to the QEF data of the voxels that touch the edge. As soon as the data of a voxel is complete, its QEF can be solved to create a vertex. Since adjacent voxel cells share their 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 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 describes which of the offsets from Table 1 should be used to identify the four potential voxel cells for each axis. For instance, 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 can be identified. However, if the adjusted coordinates no longer lie inside the bounds of the material grid, then there is no voxel. Granted that the coordinates still lie inside the grid, the voxel must be retrieved or created if it doesn't exist yet. An exemplary mesh that has been created with the developed engine is shown in Figure 14. Additionally, the Hermite data that describes the volume is visualised in the form of coloured edges.

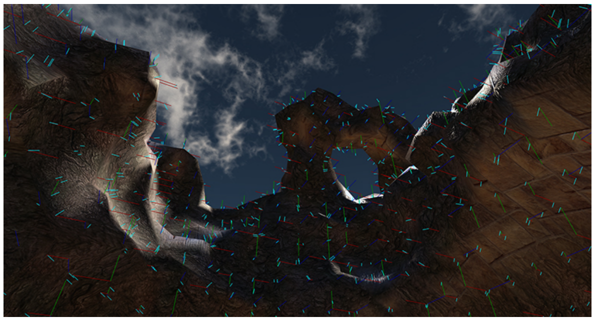

Figure 14. An exemplary mesh that has been created with the developed engine. The Hermite data is visualised in the form of coloured edges. The short blue lines represent the normals of the implicit surface and originate from the Zero Crossing positions along the grid edges.

Geometry Clipmap

Rendering a large terrain can be a very expensive task for the GPU. One way to improve 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 uses a geometry clipmap approach to render distant meshes with less vertices.

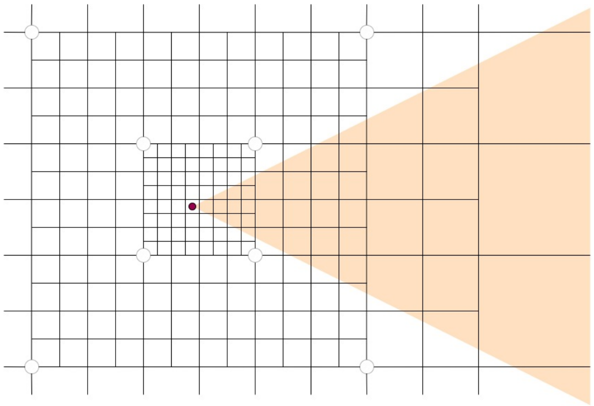

Figure 15. A 2D clipmap. In 3D, the clipmap rings around the viewer are shells.

Conventional geometry clipmaps use multiple concentric geometry rings that are positioned around the viewer as shown in Figure 15. Each ring contains roughly the same amount of data as its predecessor but is also twice as big to cover more space. The geometry patch in the middle represents the finest level of detail. In the volumetric terrain system, cubical shells are used instead of square rings. These shells consist of world octants of the same LOD grid. The farther away the shell is from the viewer, the higher the LOD. The thickness of each shell can be adjusted by the user.

The clipmap structure is supported by the world octree and can be updated in a straight-forward way. As the viewer moves around, the clipmap must continuously be updated. Internally, the clipmap maintains a list of world octant keys for every LOD shell. During an update, a new list of keys is created for each shell. The new keys are then compared with those from the current list to identify octants that have left or entered the shell. Volume modification tasks and contouring tasks are managed using two separate queues and modifications take precedence.

World octants of the LOD grids above LOD zero are used to create meshes with a lower effective resolution. The process of creating data for an octant is based on local CSG queues and is therefore the same across all LOD grids. Accumulating existing data from lower LOD grids into an octant of a higher grid is not feasible because all of the required data from the lower grids must be fully available. For example, creating a resampled data set in LOD 8 requires the data of up to eight child octants from LOD 7. Each of these eight octants also requires the data from their own eight children. As a result, $8^{8-1} = 2097152$ octants would need to be processed to obtain the required data for resampling.

Gaps and Seams

Since the volume data is maintained in separate octants, the generated isosurface also consists of multiple adjacent meshes. Splitting up large objects is usually beneficial because parts 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. Splitting the surface into patches also introduces a challenging problem: when a mesh is created from a single isolated cell, it won't automatically connect to its neighbours. This problem becomes even more complicated when a LOD system is involved.

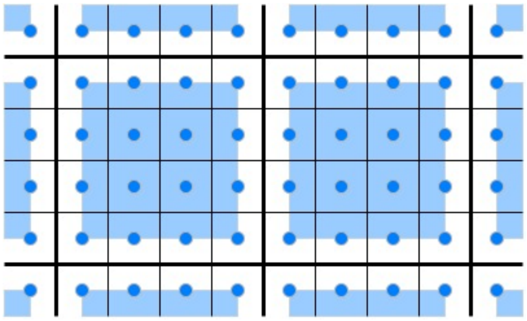

Figure 16. *The depicted blue plane occupies multiple adjacent world octants which causes gaps in the generated mesh.*

The DC algorithm can tie voxels of any size together, but keeping volume data in separate cells causes gaps between the generated meshes. Although the cells share their faces with their neighbours, the contouring process never gets the chance to connect the bordering voxels of adjacent cells as shown in Figure 16. Thus far, there are only two known approaches to solving this problem:

1. Let the cells overlap by one voxel in each positive direction.
2. Gather bordering voxels from neighbouring cells to create a seam mesh.

Letting the cells overlap requires little overhead and allows the generated meshes to connect naturally. “This works well for meshes of the same level of detail, but does not work well when one of the meshes is more detailed than the other. The extra vertices create cracks between the meshes, or holes in the surface.” [6]. Another drawback of this approach is that all cells need to contain a small amount of redundant data. Albeit incomplete, this solution appears to be the most elegant one and may be used in conjunction with the second option described by [8] and [6]: Closing the gap between different LODs can be achieved by creating a seam mesh. For that purpose, the contouring tasks that process cells at the border of a LOD shell would need to export the bordering voxels which are then used to create and update the seam mesh. Whenever a contouring task is performed on a cell that touches the seam, the seam itself must also be updated. Note that the meshes that are created from cells at LOD borders must not extend into the seam mesh. [8] also points out that the seam contouring process should only connect voxels of different data sets to prevent the creation of duplicate polygons.

The issue of duplicate polygons also applies to the overlapping cells approach. The additional voxels must not be connected among each other since the neighbouring cells will also create these polygons. Figure 17 illustrates the problem: the data cell A extends into cell B and connects the vertices b and d with a' and c' to close the gap, but a' and c' must only be connected once. Due to a lack of information on this issue, the developed system doesn't include a complete gap closing solution yet.

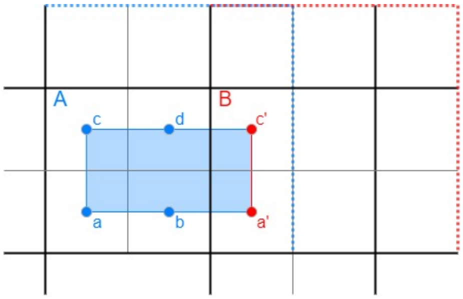

Figure 17. *Two overlapping cells. The extended boundaries of the cells are shown as dotted lines. The red edge must not be created twice.*

Performance

To measure the performance of the terrain engine's most crucial components, a series of tests has been conducted on an LG Google Nexus 5X smartphone using Android 8.1.0 and the Chrome App 7.17.28.21.arm64.

Hermite data compression and decompression as well as CSG operations, SVO creations and contouring processes have been measured for 200 samples each with a data resolution of 64 and 32. All tests, with the exception of the CSG test, operate on a data cell that is prepopulated with Hermite data using an SDF that describes a pipe with rounded edges. For the CSG test, the cell is prepopulated using a different SDF because a CSG operation skips the data combination step when the target data set is empty. In this test, the pipe SDF is added to the existing data using a Union operation.

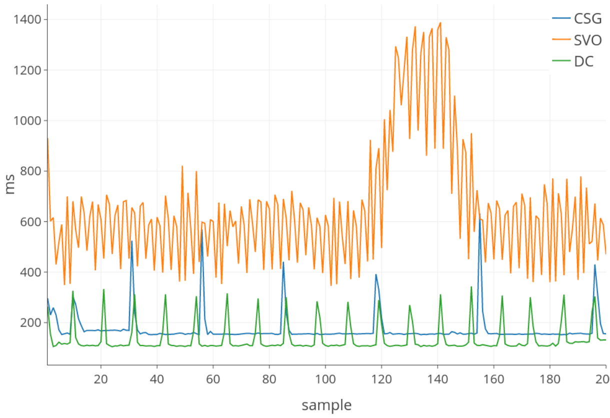

Figure 18. *Execution times in milliseconds for 200 CSG operations (blue), SVO creations (orange) and Dual Contouring processes (green) on a data set with a resolution of 64.*

Data cells need to be compressed and decompressed frequently. Surprisingly, the impact of this process on the overall performance is negligible. With a resolution of 64, the compression of 274625 material grid points takes 2.27ms on average with a minimum of 2.05ms and a maximum of 9.65ms. The decompression of 9021 material grid points and their respective run-lengths takes 2ms on average with a minimum of 1.47ms and a maximum of 6.64ms. Using a resolution of 32, the compression only takes 1.04ms on average with a minimum of 0.87ms and a maximum of 13.27*ms* while the decompression takes 0.41*ms* on average with a minimum of 0.32*ms* and a maximum of 7.34*ms*.

Figure 18 shows the test results for a data resolution of 64. CSG operations take 174.73*ms* on average with a minimum of 152.03*ms* and regular spikes of up to 632.14*ms* due to garbage collection. The initial data set contains 27493 solid materials and 7342 edges while the final material count is 102261 with 21278 edges. The SVO creation is currently the slowest system component with a mean time of 651.94*ms*. In this test, 21456 voxel cells are being created per sample. While the shortest SVO run took 345.35*ms*, the graph shows a clear performance drop between sample 115 and 155 with execution times as high as 1388.81*ms*. Each SVO test generates a large and fairly complex data structure that, when performed in quick succession, eventually exceeds the memory capabilities of the test device, causing a significant slowdown. Although the tests have been executed in a dedicated worker thread, the device struggled to manage the generated data at this resolution. In fact, the device would panic due to memory limitations when 500 consecutive CSG tests were performed using a resolution of 64, causing the browser tab to crash without a proper error message. Finally, DC operations take 135.42*ms* on average with a minimum of 104.45*ms* and regular spikes as high as 343.64*ms*. Each contouring test collected 21456 vertices from the voxel cells and created 43056 triangles.

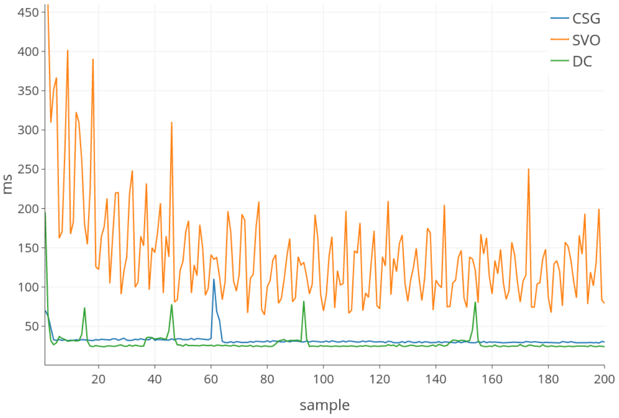

Figure 19. *Execution times in milliseconds for 200 CSG operations (blue), SVO creations (orange) and Dual Contouring processes (green) on a data set with a resolution of 32.*

Seeing that a resolution of 64 is not appropriate for mobile devices, it's worth looking at a lower resolution. Figure 19 provides the results of the same tests with a resolution of 32. Notice how the time needed for each test decreases at the beginning. This is due to the fact that the web browser optimises JavaScript programs at runtime. As expected, CSG operations perform much faster at this resolution and take 32.04*ms* on average with a minimum of 28.68*ms* and occasional spikes of up to 110.39*ms*. The initial data set contains 3623 solid materials and 1862 edges while the final material count is 12639 with 5126 edges. The lower resolution allows the test device to handle the generated data much easier. SVO creations now produce 5184 voxel cells and take 143.93*ms* on average with a minimum of 64.58*ms*. The very first run took 834.57*ms* and is not shown in the graph. Lastly, the contouring tests collect 5184 vertices and produce 10400 triangles at this resolution and take 28.87*ms* on average with a minimum of 23.91*ms* and a maximum of 195.02*ms*.

Conclusion

This work shows that the implementation of a terrain engine using JavaScript and WebGL is possible. The developed 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. 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. Although Three.js was used for testing, other frameworks such as Babylon.js could also be used with little effort since the terrain system doesn't depend on the rendering system and operates independently.

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 be noted that the developed software is still in an early development stage and that there is a lot of room for improvement. Nonetheless, the presented test results are very encouraging. The source code of the project can be found on GitHub:

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

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 MDC or CMS would be an improvement. Another aspect that can be improved is the calculation of surface normals at edge intersection points; instead of using a finite difference method, analytic derivation could be used to obtain perfect normals.

Although it was shown that volume data can be compressed very effectively, it may still use a lot of memory in total if the terrain is large. Most of the volume data remains in memory even if it's unused for long periods of time. Thus, it would be beneficial 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.

The idea of moving the contouring process to the GPU is also worth investigating while WASM could be used in an attempt to accelerate performance sensitive system components. Seam patching is another related issue that requires more attention and needs to be solved in a sane way.

With a robust terrain editor, the workflow of editing volumetric terrain could be compared to the traditional heightmap-based approach where additional 3D assets need to be used to create more complex terrain features. Furthermore, advantages and challenges of using a Leap Motion controller for terrain editing could be examined. Additionally, the potential of the terrain system for use in augmented as well as virtual reality environments could be investigated.

Acknowledgements

This research has been made possible by the ERASMUS exchange programme. We would like to thank the Norwegian University of Science and Technology for their kind support and cooperation.

References

[1] Ming Wan, Huamin Qu, and Arie Kaufman. Virtual flythrough over a voxel-based terrain. In *Virtual Reality, 1999. Proceedings., IEEE*, pages 53–60. IEEE, 1999.

[2] Brian Zaugg and Parris K Egbert. Voxel column culling: Occlusion culling for large terrain models. In *Data Visualization 2001*, pages 85–93. Springer, 2001.

[3] Leonardo Augusto Schmitz. Analysis and acceleration of high quality isosurface contouring. 2009.

[4] Eric Stephen Lengyel. *Voxel-based terrain for real-time virtual simulations*. PhD thesis, Citeseer, 2010.

[5] Manuel Scholz, Jan Bender, and Carsten Dachsbacher. Level of detail for real-time volumetric terrain rendering. In *VMV*, pages 211–218. Citeseer, 2013.

[6] Kieran Sockalingam. *Procedurally generating planetary objects*. PhD thesis, University of Oxford, 2017.

[7] Miguel Cepero. Procedural world. `http://procworld.blogspot.de/` (Visited 13.11.2017), 2017.

[8] Nicholas Gildea. Dual contouring: Seams & lod for chunked terrain. `http://ngildea.blogspot.no/2014/09/dual-contouring-chunked-terrain.html` (Visited 13.11.2017), 2014.

[9] Philip Trettner. 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 13.11.2017), 2013.

[10] Mikola Lysenko. 0 fps. `https://0fps.net/category/programming/voxels/` (Visited 13.11.2017), 2012.

[11] Brendan Eich. Ecmascript harmony: Rise of the compilers. `https://brendaneich.com` (Visited 13.11.2017), 2015.

[12] Stanley Osher and Ronald Fedkiw. *Level set methods and dynamic implicit surfaces*, volume 153. Springer Science & Business Media, 2006.

[13] Aristides AG Requicha and Herbert B Voelcker. Constructive solid geometry. 1977.

[14] William E Lorensen and Harvey E Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *ACM siggraph computer graphics*, volume 21, pages 163–169. ACM, 1987.

[15] Hubert Nguyen. *Gpu gems 3*. Addison-Wesley Professional, 2007.

[16] Leif P Kobbelt, Mario Botsch, Ulrich Schwanecke, and Hans-Peter Seidel. Feature sensitive surface extraction from volume data. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 57–66. ACM, 2001.

[17] Tao Ju, Frank Losasso, Scott Schaefer, and Joe Warren. Dual contouring of hermite data. In *ACM Transactions on Graphics (TOG)*, volume 21, pages 339–346. ACM, 2002.

[18] Sarah F Frisken Gibson. Constrained elastic surfacenets: Generating smooth models from binary segmented data. *TR99*, 24, 1999.

[19] Tao Ju and Tushar Udeshi. Intersection-free contouring on an octree grid. In *Proceedings of the 14th Pacific Conference on Computer Graphics and Applications*, volume 3, 2006.

[20] Scott Schaefer, Tao Ju, and Joe Warren. Manifold dual contouring. *IEEE Transactions on Visualization and Computer Graphics*, 13(3):610–619, 2007.

[21] Scott Schaefer and Joe Warren. Dual marching cubes: Primal contouring of dual grids. In *Computer Graphics and Applications, 2004. PG 2004. Proceedings. 12th Pacific Conference on*, pages 70–76. IEEE, 2004.

[22] Chien Ho, Fu-Che Wu, Bing-Yu Chen, Yung-Yu Chuang, Ming Ouhyoung, et al. Cubical marching squares: Adaptive feature preserving surface extraction from volume data. In *Computer graphics forum*, volume 24, pages 537–545. Wiley Online Library, 2005.

[23] George Rassovsky. *Cubical Marching Squares Implementation*. PhD thesis, Bournemouth University, 2014.

[24] Richard Hamming. *Numerical methods for scientists and engineers*. Courier Corporation, 2012.

[25] David Geier. Advanced octrees 2: node representations. `https://geidav.wordpress.com/2014/08/18/advanced-octrees-2-node-representations` (Visited 13.11.2017), 2014.

[26] Irene Gargantini. An effective way to represent quadtrees. *Communications of the ACM*, 25:905–910, 1982.

[27] David Geier. Advanced octrees 3: non-static octrees. `https://geidav.wordpress.com/2014/11/18/advanced-octrees-3-non-static-octrees/` (Visited 13.11.2017), 2014.

[28] Miguel Cepero. Voxel farm engine - reference. `http://docs.voxelfarm.com/reference` (Visited 13.11.2017), 2017.

[29] Jorge Revelles, Carlos Urena, and Miguel Lastra. An efficient parametric algorithm for octree traversal. 2000.

[30] Paul Heckbert. *Graphics Gems IV*. Elsevier, 1994.

[31] David Salomon. *Data compression: the complete reference*. Springer Science & Business Media, 2004.

[32] Christer Ericson. *Real-time collision detection*. CRC Press, 2004.

Author Biography

Raoul van Rüschen received his MS in Computer Science from the University of Applied Sciences Brandenburg (2017).

Simon McCallum received his PhD in Computer Science from the University of Otago (2007). He works as a lecturer at NTNU since 2009 and his research focuses on serious gaming and augmented reality.

Stefan Kim is a professor for media conception and production at the University of Applied Sciences Brandenburg since 1999. The main fields in his research are multimedia production, computer animation, cross-media publishing, photography and advertising film.

Reiner Creutzburg received his Diploma in Math from the University of Rostock, Germany (1976). Since 1992 he is professor for Applied Informatics at the Brandenburg University of Applied Sciences in Brandenburg. Germany. He is member in the IEEE and SPIE and chairman of the Multimedia on Mobile Device Conference at the Electronic Imaging conferences since 2005.