

# **3D model retrieval using Constructive Solid Geometry (working-title)**


Bachelor thesis

Natural Science Faculty of the University of Basel  
Department of Mathematics and Computer Science  
Database and Information Systems Research Group  
<https://dbis.dmi.unibas.ch/>

Examiner: Prof. Dr. Heiko Schuldt  
Supervisor: Ralph Gasser, MSc.

Samuel Börlin  
[samuel.boerlin@stud.unibas.ch](mailto:samuel.boerlin@stud.unibas.ch)  
16-051-716

Hand-In-Date



Missing: Date

## **Abstract**

# Table of Contents

<b>Abstract</b>	<b>ii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Approach and aim . . . . .	1
<b>2 Related Work</b>	<b>2</b>
2.1 3D model/multimedia retrieval . . . . .	2
2.2 Visualization of voxels . . . . .	2
2.3 (VR) sculpting . . . . .	2
<b>3 Concepts and Architecture</b>	<b>3</b>
3.1 General overview . . . . .	3
3.2 Cineast . . . . .	3
3.2.1 Extraction Modules . . . . .	3
3.2.2 Queries . . . . .	3
3.3 Voxels . . . . .	3
3.4 Isosurface Polygonization . . . . .	4
3.5 CSG . . . . .	4
3.6 Signed Distance Functions . . . . .	5
3.7 Voxelization . . . . .	5
3.8 Virtual Reality . . . . .	6
3.8.1 UI Design . . . . .	6
3.8.2 Sculpting interactions . . . . .	6
3.9 Unity . . . . .	6
<b>4 Implementation</b>	<b>7</b>
4.1 Cineast . . . . .	7
4.1.1 Cineast core changes . . . . .	7
4.1.2 Model Formats . . . . .	7
4.1.3 ClusterD2+Color feature extraction . . . . .	7
4.1.3.1 Sample generation . . . . .	7
4.1.3.2 Sample Selection . . . . .	8
4.1.4 Comparing features for similarity . . . . .	9
4.1.5 REST API . . . . .	10
4.2 Voxels . . . . .	11

Table of Contents	iv
4.2.1 Voxel storage . . . . .	12
4.2.2 CMS . . . . .	13
4.2.2.1 Multi-material extension for CMS . . . . .	13
4.2.2.2 Lookup table based implementation . . . . .	15
4.2.3 CSG operations on a voxel grid . . . . .	17
4.2.4 Rendering . . . . .	19
4.2.5 Voxelizer . . . . .	20
4.3 VR Sculpting . . . . .	21
4.3.1 Sculpting features . . . . .	21
4.3.2 Brush properties menu . . . . .	21
4.3.3 Custom brush editing menu . . . . .	22
<b>5 Evaluation</b>	<b>23</b>
5.1 Technical Evaluation . . . . .	23
5.2 User Evaluation . . . . .	23
5.2.1 Structure . . . . .	23
5.2.2 Results . . . . .	23
<b>6 Discussion</b>	<b>24</b>
6.1 Conclusion . . . . .	24
6.2 Lessons learned . . . . .	24
6.3 Future work . . . . .	24
<b>Bibliography</b>	<b>25</b>
<b>Appendix A Appendix</b>	<b>26</b>
A.1 User Evaluation Questionnaire . . . . .	26
<b>Declaration on Scientific Integrity</b>	<b>31</b>

# 1

## **Introduction**

1.1 Motivation

1.2 Approach and aim

# 2

## **Related Work**

### **2.1 3D model/multimedia retrieval**

Cineast, spherical harmonics, clusterD2+color, etc.

### **2.2 Visualization of voxels**

Marching cubes, dual contouring, CMS

### **2.3 (VR) sculpting**

# 3

## Concepts and Architecture

### 3.1 General overview

How all components are connected to each other (cineast, feature module, cottontail, rest api, polygonizer, voxel storage, vr interaction controller, etc.)

### 3.2 Cineast

What is cineast, how does it work (feature modules, cottontail etc.)...

#### 3.2.1 Extraction Modules

What, how

#### 3.2.2 Queries

What, how, KNN, etc.

### 3.3 Voxels

What are voxels, purpose, hermite data, etc.

### 3.4 Isosurface Polygonization

There are two main groups of such algorithms: primal ( ) and dual ( ) contouring algorithms. Generally these algorithms cannot polygonize a single voxel by itself, but instead require a cell consisting of eight voxels, usually arranged in the form of a cube. Primal and dual contouring algorithms differ in that the primal variants place polygon vertices somewhere on the cell's edges and the dual variants place them not on the edges but somewhere within the cell's volume. Therefore dual contouring algorithms have the advantage that they can more reliably polygonize surfaces with sharp features, such as e.g. the corners of a cube. Primal contouring algorithms can also polygonize sharp features in certain cases, however only if said sharp feature lies exactly on a cell's edge. Fig. 3.1 shows a polygonized voxel cell where the sharp feature lies within the voxel cell.

Marching Cubes [1] (MC), a primal contouring algorithm, checks whether the material at each corner of the voxel cell is filled or empty and then packs all 8 resulting booleans into a single 8 bit integer. That integer is then used to index into a lookup table that contains which cell edges are to be connected to each other to form polygons. Vertex positions can either lie on the middle of their respective cell edge or if the voxel contains a density value the vertex position can be smoothly linearly interpolated.

Dual Contouring [2] (DC) uses another approach. A valid voxel cell always contains either no edge with a material, or at least three edges with a material change. If a voxel cell does contain edges with a material change then Dual Contouring finds the sharp feature that minimizes the squared distance between the sharp feature and all planes spanned by the normals at the edges with the material changes. In a second pass the generated vertices are then connected between neighboring voxel cells to form polygons.

Cubical Marching Squares [3] (CMS) makes use of both primal and dual contouring features. Much like MC it places vertices on voxel cell edges. However it can also place vertices on voxel cell faces and within the voxel cell volume, like dual contouring algorithms do. This gives it the advantage that each cell can be polygonized individually while still being able to reliably polygonize surfaces with sharp features.

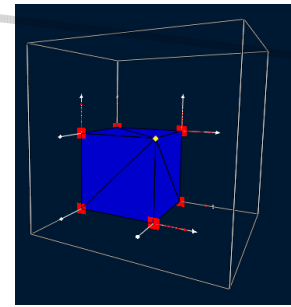


Figure 3.1: A polygonized voxel cell (white) containing the polygon surface of a cube's corner (blue) with a sharp feature (yellow).

### 3.5 CSG

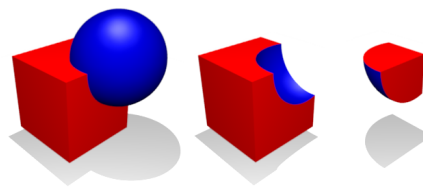


Figure 3.2: Boolean union (left), difference (middle) and intersection (right) of a cube and sphere primitive. Adapted from Wikipedia<sup>1</sup>.



Constructive Solid Geometry (CSG) is a method to create geometric shapes by constructing them from primitives and a few operations. The three most common operations are the boolean union ( $\cup$ ), difference ( $-$ ) and intersection ( $\cap$ ) operations are shown in Fig. 3.2. With just those three simple tools and a couple of primitives, e.g. spheres, cubes, cylinders, etc., the user can easily create arbitrary shapes or sculptures. Additionally CSG need not be restricted to only boolean operations, but can also make use of smooth operations that blend two shapes together in a continuous manner. In the context of signed distance functions (see further below) these smooth operations correspond to the smooth minimum and maximum operations.

The application developed for this thesis makes extensive use of CSG to enable the users to edit and shape their voxel based sculptures. The union ("add") and difference ("remove") operations are intuitive and easy to use. Additionally it also provides a "replace" operation that allows the user to replace solid materials inside a primitive with another material. Logically it is equivalent to  $(A - B) \cup (B \cap A)$  where A is the existing sculpture and B is the primitive or bounds within which solid materials should be replaced with B's material.

### 3.6 Signed Distance Functions

Signed distance functions (SDF) are mathematical formulas that describe the signed distance between a point and the surface of a certain shape. For points inside the shape the signed distance is negative. For example it is trivial to derive the signed distance function  $f(x, y, z) = \sqrt{x^2 + y^2 + z^2} - r$  of a sphere at  $(0, 0, 0)$  from the implicit formula of the sphere's surface,  $x^2 + y^2 + z^2 - r^2 = 0$ . There exist readily available formulas for SDFs of various shapes, e.g. Inigo Quilez' collection of such functions<sup>2</sup>.

SDFs are well suited as a representation of CSG primitives because one can approximate their union and difference by taking the minimum, respectively maximum, of two SDFs. This approach generally works well and can produce the exact union or difference, however in certain cases it can result in an incorrect approximation, especially on the inside of union'd SDFs. For the purposes of this application the SDFs need not be exact on the inside, hence we have decided to use SDFs as the representation of our CSG primitives. Additionally since the domain and range are the same for all SDFs, namely  $f: \mathbb{R}^3 \mapsto \mathbb{R}$ , they can all be treated exactly the same besides their formulas and can be encapsulated as a simple method in code.

### 3.7 Voxelization

An important part of multimedial retrieval is the ability to take query results and reuse them to formulate a new, refined, query. In the context of this thesis the multimedia retrieval is only concerned with polygonal meshes. To facilitate the modification of such polygonal meshes the application includes a voxelizer module. A voxelizer takes a polygonal mesh as input and converts it into a suitable voxel representation. In particular, to achieve high accuracy our voxelizer converts the polygonal mesh into voxels. Since our voxels contain not only the voxel material, but also the surface normals, we can reconstruct the mesh from our voxels while keeping most of the mesh's features intact, given a large enough voxel grid with sufficient resolution. In

<sup>1</sup> [https://en.wikipedia.org/wiki/Constructive\\_solid\\_geometry](https://en.wikipedia.org/wiki/Constructive_solid_geometry)

<sup>2</sup> <https://iquilezles.org/www/articles/distfunctions/distfunctions.htm>

<sup>3</sup> <http://graphics.stanford.edu/data/3Dscanrep/>

practice a resolution of  $128 \times 128 \times 128$  seems to be sufficient for most models that aren't highly detailed or noisy. Fig. 3.3 shows an example of a mesh that has been converted to a voxel grid.

### 3.8 Virtual Reality

What is VR, use cases

#### 3.8.1 UI Design

Windows vs. using 3D space

#### 3.8.2 Sculpting interactions

One hand to grab, other to sculpt, rotating brush with trackpad, etc.

### 3.9 Unity

What, why



Figure 3.3: The Stanford Armadillo mesh<sup>3</sup>(left) and its voxel representation at a resolution of  $128 \times 128 \times 128$  voxels (right).

# 4

## Implementation

### 4.1 Cineast

#### 4.1.1 Cineast core changes

UV + texture support in meshes and OBJ loader

#### 4.1.2 Model Formats

The current implementation supports the Wavefront OBJ [1] format as it is comparatively simple to parse. However, the feature extraction algorithm is agnostic to the original 3D model format and only requires a 3D mesh consisting of triangles and vertices that contain the following attributes: position and texture coordinates. Because of Cineast's [2] modular nature it is possible to add parsers for other 3D model formats, or to extend the currently existing parsers to also read the aforementioned vertex attributes, without having to alter the feature extraction algorithm.

Missing: Missing r

Missing: Missing r

#### 4.1.3 ClusterD2+Color feature extraction

In order to compare two or more 3D models for similarity the algorithm must first identify or extract information that is relevant for the calculation of the similarity metric. A common way to do so is to reduce the 3D model into an N-dimensional real-valued feature vector which contains only the most relevant information, or a reduced and compacted version thereof, for a similarity comparison. This feature vector can then be used to compute a similarity metric, e.g. by calculating the L2 distance between feature vectors.

Since our application allows the user to color their sculptures it makes sense that similarity queries also take color into account. Hence part of the color information must be contained in the feature vector. The ClusterD2+Color algorithm described by [3], which was used in our application, achieves this by creating random sample points on the 3D model's surface according to certain criteria and then use those sample points to compute a feature vector.

Missing: Missing r

##### 4.1.3.1 Sample generation

Many isosurface polygonization algorithms, such as our non-adaptive CMS [4] implementation used in this application, can produce highly tessellated meshes. Opposite to that meshes exported by 3D modeling software are often optimized to have large triangles where the surface has little detail, and small triangles where the surface is more detailed. Hence it is important that the

Missing: Introduc  
abbr before

sample point generation is mostly independent of the triangle sizes of the mesh, so that meshes of varying levels of tessellation can be compared for similarity effectively. The ClusterD2+Color algorithm achieves this by making the sampling directly proportional to the surface area of the triangles, hence a sample point has a higher chance to be positioned on a large triangle, and a smaller chance to be positioned on a small triangle. Before any sample points are generated the model is first scaled such that its total surface area equals 100. After generating a certain number samples the result is a collection of samples that are uniformly distributed on the mesh's surface. The number of samples is determined by a function of the integral of the absolute Gaussian curvature over the model's surface as described in [1].

Missing: Missing r

#### 4.1.3.2 Sample Selection

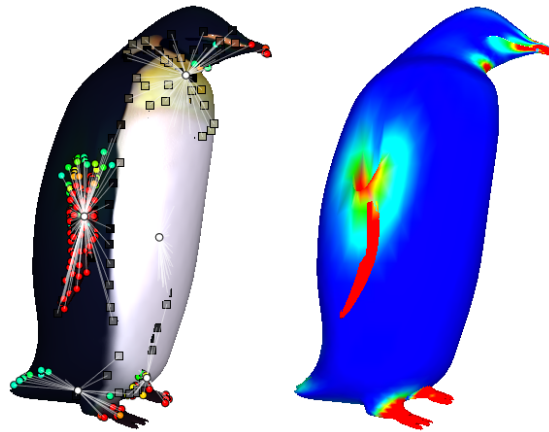


Figure 4.1: Visualization of the sample selection and clustering. On the left is shown the textured input model and the resulting shape (circles) and color (squares) samples and their cluster assignments (white circles and lines). The picture on the right shows the computed difference-of-Laplacian  $D$  of the model (red: high, blue: low).

Once the initial samples have been generated they are filtered and tested for certain criteria. These criteria differ between two different types of samples:

**Shape Samples** These samples should convey relevant information about the geometric shape. Previous work done by [1] has shown that mean curvature is a good indication for visual saliency and thus for comparing geometric shapes for similarity. The mean curvature of the mesh vertices is computed using Taubin's method [2]. Our implementation of the ClusterD2+Color algorithm computes the mean curvature of each vertex and stores it as a vertex attribute. This mean curvature attribute is then smoothed over the mesh's surface by a Laplacian smoothing approximation described by [3], resulting in a curvature map. In particular, our implementation sets  $\lambda = 8$  and, to satisfy the stability criterion  $\lambda dt \leq 1$ ,  $dt = \frac{1}{8}$ , i.e. equivalently it applies small smoothings with  $\lambda = 1$ ,  $dt = 1$  a total of eight times for one full smoothing operation. This is repeated multiple times to create three smoothed curvature maps  $C^1$  through  $C^3$ . Each curvature map  $C^i$  has been smoothed  $i$  times. As opposed to [1] we use  $C^1$  as the baseline curvature map instead of  $C^0$ , because certain non-manifold (e.g. vertices or edges touching without being connected) models exhibit significant artefacts in  $C^0$  as shown in Fig. 4.2. Lastly each shape sample calculates its  $D = D^2 \oplus D^3$  value, where  $D^i = C^i - C^{i-1}$  is the difference-of-Laplacian

Missing: Missing r  
Lee,C.H.,Varshney  
pp. 659–666 (2005)

Missing: Missing r

Missing: Missing r  
Implicit Fairing of  
regular Meshes us  
Diffusion and Cur  
ture Flow

Missing: Missing r

space, as shown in Fig. 4.1, and only the samples with the top 10% highest  $D$  values are kept. Indeed the remaining top 10%  $D$  value shape samples in Fig. 4.1 are concentrated to the relevant features of the penguin model, namely its arms, feet and beak.

**Color Samples** The color samples' purpose is to allow the algorithm to distinguish between models that have the same geometric shape but differ in color or texture. This is achieved by only selecting those samples that lie on significant color transitions. As described by Y.-J. Liu et al. [10], all generated samples extract the mesh's color at their position and then transform the color to the CIE-LAB color space [11]. The resulting three CIE-LAB coordinates,  $L^*$ ,  $a^*$  and  $b^*$  are then quantized into 6 ( $L^*$ ), respectively 4 ( $a^*$ ) and 4 ( $b^*$ ) integer values. These three, now quantized, coordinates are combined into a single color code integer, such that each of those color codes corresponds to a unique combination of the quantized coordinates. Similar colors are thus mapped to the same color code.

Lastly the algorithm only selects those samples as color samples of which less than 80% of the neighboring samples' color codes are the same. In our implementation the neighborhood range is defined as a function of the model's mean edge length, or if less than 5 samples are found within said range then it chooses the 5 nearest samples as the neighborhood. In Fig. 4.1 the color samples are selected mostly along the transition between the white, black and beige hair of the penguin

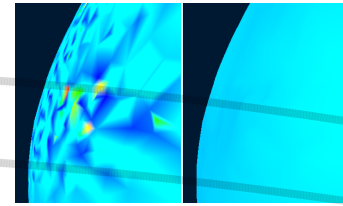


Figure 4.2:  $C^0$  (left) exhibits significant artefacts on a non-manifold mesh.  $C^1$  (right) is a more sensible curvature map as baseline for the shown model.

#### 4.1.4 Comparing features for similarity

The selected shape and color samples now need to be reduced into an N-dimensional feature vector. ClusterD2+Color algorithm achieves this by first clustering nearby samples together and then creating a histogram from the euclidean distances, i.e. L2 norm, between all samples that are not in the same cluster.

More specifically, the samples are clustered together using a modified ISODATA algorithm described in [12]. In our implementation we set  $P_n = 5\%$  of the total number of sample points,  $P_s = 2 * E$ ,  $P_c = 10 * E$ ,  $I = 100$  with  $E = \text{mean edge length of the model}$ . Additionally we introduce two new parameters:  $I_{min} = 10$ , the minimum number of iterations, and  $I_{nc} = 200$ , the total maximum number of iterations after which the algorithm is stopped if no convergence is happening. Without the  $I_{min}$  parameter we have observed that the algorithm would commonly stop too early with inadequate cluster assignments. Then, once the samples are clustered together, the euclidian distances between all samples in different clusters are computed and stored in an array. The values in this array are then normalized s.t. the lowest value in the array is mapped to zero and the highest value is mapped to one. Finally this array is then converted into a histogram with 20 bins.

Y.-J. Liu et al. [10] have also proposed a second method called ClusterAngle+Color which calculates the values for the histogram from the angle between triplets of samples that are each in different clusters. According to the authors this method results in a better similarity measure,

however due to the  $O(N^3)$  complexity we have unfortunately found it to be infeasible with the large number of samples generated from the user sculpted models in our application.

In either case the result is a histogram which serves as 20-dimensional feature vector and in fact, since they are a geometric signature of the model, they can be used to compare models for similarity, e.g. by using the  $\chi^2$  distance metric, a commonly used distance metric to compare two histograms for similarity. The lower the  $\chi^2$  distance the more similar the models. Conveniently Cineast and its multimedia retrieval database Cottontail DB<sup>4</sup> offer an efficient k-nearest neighbors query for the  $\chi^2$  distance metric.

Along with the  $\chi^2$  distance metric the Jensen-Shannon divergence ( $D_{JS}$ ), described by Y.-J. Liu et al. [1], was also implemented. In short, the  $D_{JS}$  metric treats the histogram feature vector as an independent and identically-distributed sample of a random variable and thus first converts the histogram into a probability density function by using kernel density approximation with a gaussian kernel. The Jensen-Shannon divergence of the two probability density functions to compare is then computed and serves as a probabilistic measure ranging from 0 to 1 for how similar they are, so in a sense how likely it is that the two histograms were generated from the same model. Fig. 4.3 shows the histogram of a penguin model and its probability density function, along with some debug information of curves required for the  $D_{JS}$  calculation.

#### 4.1.5 REST API

In order for a program to interface with Cineast's query capabilities an API is required. Cineast provides two main APIs, a WebSocket API and more importantly a REST API which was used for the purposes of this thesis.

REST APIs, an acronym for Representational State Transfer, aim to communicate between to programs via requests and responses over the HTTP web protocol. The following, incomplete, list should give an insight into how REST APIs operate - according to R. Fielding [2] REST APIs follow at least these following principles:

- Client-Server: User interface concerns should be separated from data storage concerns.
- Stateless: Communication must be stateless. All information required for a request to be understood must be included in said request.
- Cache: Responses' data must be labeled whether the data can be cached on the client side or not. If it is cacheable the client may reuse the same data for future equivalent requests.
- Uniform Interface: The REST API should form a uniform interface. Said interface should remain the same independent of the back-end implementation.

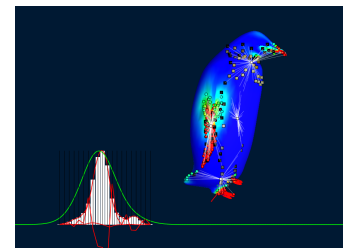


Figure 4.3: The feature visualization tool that was developed during the feature extraction implementation. White: histogram feature vector. Green: probability density function of the histogram.

<sup>4</sup> <https://github.com/vitrivr/cottontaildb>

- And several other principles.

Usually REST requests are exclusively executed by sending an HTTP GET, PUT, POST or DELETE request through certain server provided URLs. For example Cineast's similarity query endpoint has the following URL and is executed by sending a POST request: `http://cineast.domain/find/segments/similar`. Additional data, such as the model to run the similarity query for, is included in the HTTP request body in the form of JSON data.

Throughout the course of development of this thesis Cineast's REST API was partially rewritten to use the Javalin<sup>5</sup> web framework instead of Spark<sup>6</sup> and SparkSwagger<sup>7</sup>. Javalin makes great use of Cineast's already built-in JSON serialization structure using Jackson<sup>8</sup>. This enables Javalin, in combination with its OpenAPI plugin, to generate an OpenAPI specification that describes Cineast's REST API in a standardized way. Moreover this OpenAPI specification allows programs like Swagger Codegen<sup>9</sup> to generate fully functional code libraries in various programming languages capable of interfacing with the REST API defined by the specification. Since Javalin's OpenAPI plugin makes use of Jackson, it is capable of more reliably creating the specifications for JSON data structures than SparkSwagger, especially for subclasses of superclasses with generic types. The C# library that was used to access Cineast's API endpoint from Unity was generated by the aforementioned OpenAPI specification and Swagger Codegen.

## 4.2 Voxels

The foundation of the sculpting feature is based on voxels. In the most basic sense a voxel represents a value on a regular grid in 3D space  $\mathbb{R}^3$ . These voxels are very useful to represent volumetric structures, such as sculptures or terrain, since they fill the 3D space and each voxel represents one volume element.

Since voxels are an abstract representation of volumes they can be visualized in various ways depending on the data the voxel contains. Usually the data of a voxel consists of a single attribute, a color, in which case the voxel can be visualized e.g. by cubes, spheres, or splats. More sophisticated voxels contain additional attributes such as normals or texture coordinates. In particular, our application uses voxels that consist of a 32 bit material ID plus Hermite data  $\mathbb{H}$ . Hermite data, first described by  $\mathbb{H}$ , consists of two attributes: a material, exact intersection points and normals. Fig. 4.4 illustrates the Hermite data and how it can represent diverse shapes. The Hermite data and voxel material provide enough information to construct the piece of surface represented by a voxel in the form of polygons.

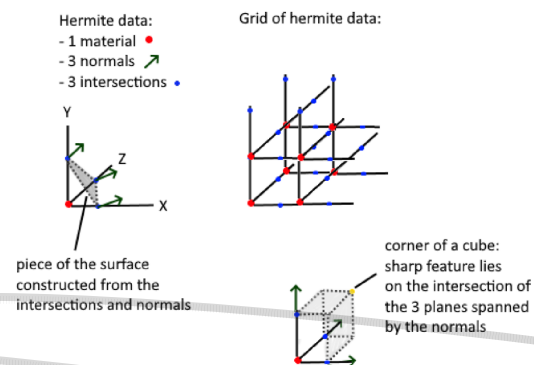


Figure 4.4: Hermite data.

<sup>5</sup> <https://javalin.io/>

<sup>6</sup> <http://sparkjava.com/>

<sup>7</sup> <https://github.com/manusant/spark-swagger>

<sup>8</sup> <https://github.com/FasterXML/jackson>

<sup>9</sup> <https://swagger.io/tools/swagger-codegen/>

Missing: Missing n  
Wikipedia

Missing: Missing n  
Dual Contouring

Missing: Missing n  
Dual Contouring

### 4.2.1 Voxel storage

There are many different data structures to store voxels, the simplest of which is simply storing the voxels' values in large 3D arrays. Other data structures however need not necessarily store all voxel values on a regular grid. Often there are regions with many voxels of the same value, and in that case it is beneficial to use a more advanced data structure that can store voxel values more memory efficiently.

The most common method to store voxels in memory is to split the voxels into fixed size blocks or chunks. Within each chunk the voxels are stored in an array. The chunks are stored separately in a sparse data structure, e.g. a hash map, where they can be accessed by their position. This gives a good trade-off between memory usage and access speed, because often voxel accesses happen within the

same chunks and hence the hash map lookups can usually be cached. Instead of multi-dimensional arrays for the chunk voxel storage we use one contiguous linear array per chunk to avoid having to dereference multiple pointers for a single access. The index into this linear array can be directly computed from a given X, Y and Z position within the chunk by the following formula:  $index(x, y, z) = x * N * N + y * N + z$ , where  $N$  is the chunk size. Our implementation uses a chunk size of  $N = 32$ . Voxels on the faces towards the positive X, Y and Z axes are padding voxels that always mirror the state of the neighbour voxel in the respective neighbour chunk. These padding voxels allow the voxel polygonizer to run faster since it entirely removes the need to query neighbour chunks during meshing. This method is described by [ ] in more detail.

As previously mentioned the voxels in this application consist of a 32 bit material ID plus Hermite data. Therefore the size of a single voxel in memory is: 4 (material) + 3 \* 3 \* 4 (three normals, each with three 32 bit float components) + 3 \* 4 (three intersection points) = 52 bytes in total. Considering that a single chunk contains  $32 * 32 * 32 = 32768$  voxels, hence consumes roughly 1.7MB, and one scene can contain many of these chunks, the main memory will be exhausted rather quickly. To remedy this issue the Hermite data compression scheme described by [ ] was used. This quantized Hermite data compression is lossy since it relies on quantizing the floating point values into small integers that are then packed together using bitwise operations. The errors introduced by the compression are negligible: the exact intersection points remain accurate to a  $\frac{1}{1024}$ th, respectively less than 0.1%, of the voxel's edge length. Similarly the maximum error in a normalized normal's component is  $\frac{1}{1024}$ . The effective size of the quantized Hermite data is 12 bytes (two of which are struct alignment padding), i.e. only 23% of the uncompressed size. Fig. 4.5 shows the quantized Hermite data structure in more detail.

A further potential optimization is to change the linear array indexing scheme. Modern CPUs have multiple cache layers - each cache layer is a fast memory unit and generally the faster the

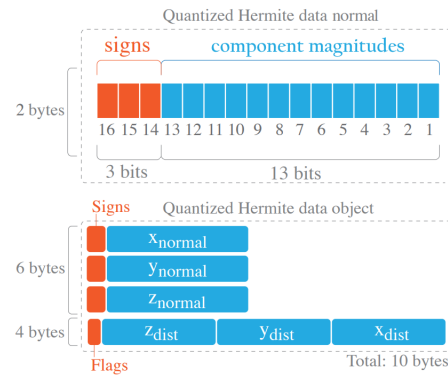


Figure 4.5: Quantized Hermite data structure. Top: a normal's component magnitudes are packed into 13 bits plus 3 bits for the signs. One normal thus fits into two bytes. Bottom: the entire quantized Hermite data object.  $x, y, z_{dist}$  refer to the intersection points. Adapted from [XXX].

Missing: Missing r  
Fig in Paper

Missing: Missing r  
refined data addre  
and processing sch  
to accelerate volum  
raycasting

Missing: Missing r  
Fast and Adaptive  
Polygon Conversion  
By Means Of Spac  
Volumes



cache the less memory it can store. When a program accesses a piece of memory from main memory the CPU loads the surrounding address space into a fixed size cache line, a contiguous piece of the heap memory around the access, into the cache. If the program then accesses a nearby address again the CPU can read the value from the cache which is a lot faster than reading it from main memory. This is called a cache hit. On the other hand if the program accesses an address further away from the previous access than the size of a cache line the CPU will have to read the value from main memory instead, ergo a cache miss. Hence by optimizing for cache hits a program's speed can be improved if it is memory bound. The most common access pattern for our voxel storage is to read eight neighbor voxels as a voxel cell for polygonization. For simplicity's sake we will consider a 2D instead of 3D array: if we use simple linear indexing as shown in Fig. 4.6 to read the voxel cell (1, 2, 5, 6) there will be a large address difference between the voxels on the first and second row. With large arrays this will often lead to cache misses. By instead using Z-Order indexing, also known as Morton indexing, spatially nearby voxels are much more likely to be nearby to each other in the address space, increasing the chance for cache hits. However, such indexing schemes usually have a larger overhead due to the calculation of the index. To compute the Z-Order indices parts of the Libmorton library<sup>10</sup> were used and ported to C#. Unfortunately in our case using Z-Order indexing has made no significant difference.

Missing: Validate

#### 4.2.2 CMS

Since the voxels are just an internal volumetric representation some algorithm is required to convert them into something that a computer can display on the monitor, e.g. a polygonal 3D mesh. From the group of iso-surface and surface polygonization algorithms we have decided to use the Cubical Marching Squares algorithm described by [1].

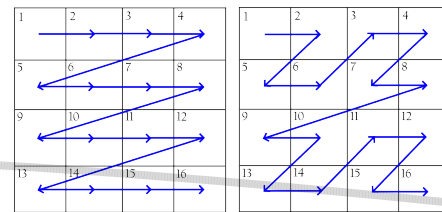


Figure 4.6: Simple linear indexing (left), Z-Order indexing (right). Each square represents a voxel in a 2D array. The blue line represents the order in which they are stored in main memory.

Missing: Missing CMS

##### 4.2.2.1 Multi-material extension for CMS

The regular CMS algorithm does not support multiple materials. Hence due to the need for multi material volumes in this thesis's application a multi-material CMS extension was developed. In order for this to work the Hermite data normals must additionally also be stored for each material change edge, and not only for non-solid to solid changes or vice versa. Furthermore each segment vertex must store a material. Only one change was required in the algorithms shown in the CMS paper: GENERATESEGMENT's modification is shown in Alg. 1. Since our application only uses regular voxel grids the modification was made to work on regular voxel grid cells. It can however be extended to work with the adaptive CMS version in a similar manner.

<sup>10</sup> <https://github.com/Forceflow/libmorton>

---

**Algorithm 2 DetectMaterialTransition.** *This procedure finds and inserts material transitions into the segments of a face  $f$ . INSERTTRANSITION inserts the transition points into  $f.list$  at their appropriate indices s.t. the segment's pieces remain straight and don't form folds, i.e. s.t. they subdivide the segment, and transition2 is to be inserted after transition1. The MATERIAL\_TRANSITION table returns the edge connecting the two face corners with solid materials given the marching squares case.*

---

```

1: procedure DETECTMATERIALTRANSITION(Face  $f$ )
2:   if  $f.marchingSquaresCase$  is 2 then
3:      $edge \leftarrow \text{MATERIAL\_TRANSITION}[f.marchingSquaresCase]$ 
4:      $normal \leftarrow \text{GETNORMAL}(f, edge)$ 
5:     if  $normal \neq \emptyset$  then  $\triangleright$  There can only be a transition if a normal exists.
6:        $intersection \leftarrow \text{GETINTERSECTION}(f, edge)$ 
7:        $tangent \leftarrow \text{CONSTRUCTTANGENT}(intersection, normal)$ 
8:       for all  $l \in f.list$  do
9:         for all  $p \in l$  do  $\triangleright$  The variable  $p$  is one piece of the segment  $l$ .
10:          if  $tangent$  intersects with  $p$  then
11:             $transitionPoint \leftarrow \text{INTERSECTIONPOINT}(tangent, p)$ 
12:             $transition1 \leftarrow \text{CREATETRANSITION}()$ 
13:             $transition1.position \leftarrow transitionPoint$ 
14:             $transition1.material \leftarrow edge.points[0].material$ 
15:             $transition1.normal \leftarrow normal$ 
16:             $transition2 \leftarrow \text{CREATETRANSITION}()$ 
17:             $transition2.position \leftarrow transitionPoint$ 
18:             $transition2.material \leftarrow edge.points[1].material$ 
19:             $transition2.normal \leftarrow normal$ 
20:            INSERTTRANSITION( $f.list, transition1, transition2$ )
21:          end if
22:        end for
23:      end for
24:    end if
25:  end if
26: end procedure

```

---



---

**Algorithm 1 GenerateSegment.** *This procedure remains mostly the same as described in the CMS paper. The only difference is the last line 10.*

---

```

1: procedure GENERATESEGMENT(Face  $f$ )
2:   if there are two segments then
3:      $\{l_1, l_2\} \leftarrow \text{RESOLVEFACEAMBIGUITY}(f)$ 
4:      $f.list \leftarrow \{l_1, l_2\}$ 
5:     DETECTFACESHARPFEA-
6:     TURE( $f.list$ )
7:   else if there is one segment  $l$  then
8:      $f.list \leftarrow \{l\}$ 
9:     DETECTFACESHARPFEA-
10:    TURE( $f.list$ )
11:   end if
12:   DETECTMATERIALTRANSITION( $f$ )
13: end procedure

```

---

Additionally to the GENERATESEGMENT modification the DETECTSHARPFEATURE implementation also requires a change. The transitions generated in DETECTMATERIALTRANSITION, which

are added to  $f.list$ , and their normals must be included in the linear system of equations to be solved ([]) for the sharp feature detection, as suggested in []. This causes the sharp feature to be placed where one would expect the material transition to occur on the surface, as shown in Fig. 4.7. This of course means that the multi-material CMS algorithm must always generate a sharp feature when material transitions are present, even if the sharp feature conditions/heuristics described by [] are not met.

Missing: CMS and Kobbelt et al.

Missing: Missing DC paper

Missing: Kobbelt

#### 4.2.2.2 Lookup table based implementation

Much of the CMS algorithm can be done and perhaps be sped up using appropriate lookup tables, especially in the case of regular grid voxel cells. Indeed we have generated such lookup tables in order to show the viability of a lookup table based approach. Due to time constraints and it breaking the previously mentioned multi-material extension however, it has ended up not being used in the final implementation. Nevertheless with some more work it can also be made to work with multi-material volumes. Furthermore this lookup table based approach may prove useful for a future GPU based implementation as it splits the problem into several small steps that can all be run in parallel with very little divergent behaviour.

A tool was implemented to generate the lookup tables for all  $2^8 = 256$  possible cell states (two states per cell corner, solid vs. non-solid). Using rotational symmetries several lookup tables have been reduced in size - the initial lookup table maps one of the 256 cell states to one of 23 base configurations by rotating it appropriately (one or multiple  $90^\circ$  rotations around the X, Y and Z axes). Each of the in total 24 unique rotations gets its own index. All other lookup tables are based on these base configurations, their components (in the context of CMS) and the rotation index. From all the possible components that CMS can generate in a regular grid voxel cell, the lookup table tool has identified a total of 86 unique components for all base configurations. These components have then been indexed and their size and edges were stored in a separate lookup table.

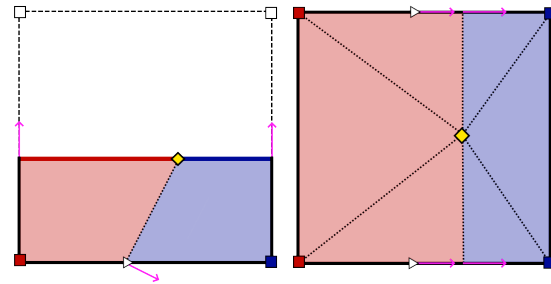


Figure 4.7: Multi-material CMS. The side view (left) shows how the material transitions (white triangles) are projected onto the segment, causing a split where one would expect the material transition to occur on the surface. The top view (right) shows how those splits are included in the sharp feature (yellow diamond) solution. The right side also shows the resulting triangulation (dotted lines).

---

**Algorithm 3 DisambiguationBit.** *Determines the disambiguation bit for the specified base case and ambiguity number.*

---

```

1: procedure DISAMBIGUATIONBIT(int baseCase, int ambiguityNr)
2:   tableIndex  $\leftarrow$  baseCase * 6 * 4 + ambiguityNr * 4
3:   edges  $\leftarrow$  {CASE_AND_AMBIGUITY_NR_TO_EDGES[tableIndex + 0],
   ..., CASE_AND_AMBIGUITY_NR_TO_EDGES[tableIndex + 3]}
4:   segment1  $\leftarrow$  CONSTRUCTSEGMENT(edges[0], edges[1])  $\triangleright$  Constructs a segment from the
   intersections on edge[0] and edge[1]
5:   DETECTFACEHARPFEATURE(segment1)
6:   segment2  $\leftarrow$  CONSTRUCTSEGMENT(edges[2], edges[3])
7:   DETECTFACEHARPFEATURE(segment2)
8:   if segment1 intersects with segment2 then
9:     return 1
10:  else
11:    return 0
12:  end if
13: end procedure

```

---

One complication of the lookup table based approach are the potential 2D cell face ambiguities. All tables thus require as index not only the base configuration and rotation index, but also a 6 bit disambiguation index (one bit per cell face, also called "ambiguityRes"). For a given base configuration case and an ambiguity number (i.e. the index of the ambiguity to check, ranging from 0 to 5) Alg. 3 calculates a disambiguation bit. These disambiguation bits are then bit packed into a single integer where the bit with *ambiguityNr* = 0 is the most right bit and then from right to left according to *ambiguityNr*, resulting in the 6 bit disambiguation index. Although less common, the potential 3D ambiguities as mentioned in the CMS paper are not being handled by lookup tables, those still need to be resolved dynamically.

With the base configuration, rotation index and now also the disambiguation index a list of components can be obtained from yet another lookup table. The proposed lookup table based CMS algorithm then creates the segments for each component, finds their 2D and 3D sharp features similarly to the regular CMS algorithm, and finally triangulates them.

In total the proposed lookup table based CMS algorithm primarily requires the following lookup tables which have been generated by our lookup table tool:

- RAW\_CASE\_TO\_CASE\_ROTATION\_AND\_AMBIGUITY\_COUNT[*caseIndex* \* 3 + *value*]
 

Returns either the base configuration case, rotation index or ambiguity count

  - *caseIndex*: One of the 256 possible cell cases
  - *value*: 0 for base configuration case, 1 for rotation index, 2 for ambiguity count
- CASE\_AND\_AMBIGUITY\_NR\_TO\_FACE[*baseCase* \* 6 + *ambiguityNr*]
 

Returns the base configuration face index (i.e. after reducing the rotational symmetries) for the given *ambiguityNr*

  - *baseCase*: One of the 23 base configuration cases
  - *ambiguityNr*: Index of the ambiguity, as discussed above
- ROTATION\_TO\_RAW\_FACES[*rotationIndex* \* 6 + *baseFaceIndex*]
 

Returns the original face index (i.e. before reducing the rotation symmetries)

  - *rotationIndex*: One of the 24 unique rotations

- *baseFaceIndex*: Base configuration face index
- CASE\_AND\_AMBIGUITY\_NR\_TO\_EDGES[ $baseCase * 6 * 4 + ambiguityNr * 4 + entry$ ]  
Returns the base configuration edge index (see Alg. 3)
  - *baseCase*: One of the 23 base configuration cases
  - *ambiguityNr*: Index of the ambiguity, as discussed above
  - *entry*: Which edge index to return, i.e. 0 - 3 (four edges per face)
- ROTATION\_TO\_RAW\_SIGNED\_EDGES[ $rotationIndex * 12 + baseEdgeIndex$ ] & 0b01111  
Returns original edge index (i.e. before reducing the rotation symmetries)
  - *rotationIndex*: One of the 24 unique rotations
  - *baseEdgeIndex*: Base configuration edge index
- CASE\_AND\_AMBIGUITY\_RES\_TO\_SIZE\_AND\_COMPONENTS[ $baseCase * 64 * 5 + ambiguityRes * 5 + componentNr$ ]  
Returns the base configuration component index
  - *baseCase*: One of the 23 base configuration cases
  - *ambiguityRes*: Disambiguation index, as discussed above
  - *componentNr*: Which component to return, i.e. 1 - 4 (at most 4 configurations per voxel cell). If 0, the number of components in the voxel cell is returned.
- COMPONENT\_TO\_SIZE\_AND\_EDGES[ $componentIndex * 13 + edgeNr$ ]  
Returns the base configuration edge index
  - *componentIndex*: One of the 86 unique base configuration components
  - *edgeNr*: Which component to return, i.e. 1 - 12 (at most 12 edges per component). If 0, the number of edges in the component is returned.

For further details about the proposed lookup table CMS algorithm please refer to our lookup table tool and sample implementation written in Java.

Missing: Clean up link somewhere

### 4.2.3 CSG operations on a voxel grid

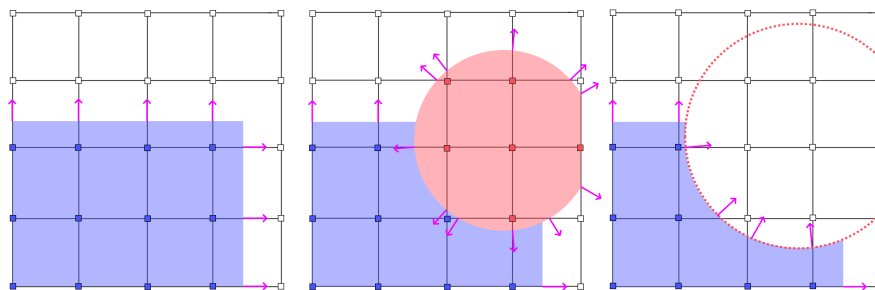


Figure 4.8: CSG operations on a 2D Hermite data voxel grid. From left to right: initial grid, union and difference. Each square and its line segments to the right and above, including any normals (pink) on them, represent one Hermite data voxel.

The CSG operations that our application uses for the sculpting brushes can be applied using variants of Alg. 4.

---

**Algorithm 4 Union.** *Applies a CSG union to a voxel grid. FINDINTERSECTION finds the intersection of a primitive with the specified edge and returns an intersection object. intersection.distance is the distance between the intersection and a voxel or edge.voxels[0], and intersection.normal is the surface normal vector at the intersection.*

---

```

1: procedure UNION(VoxelGrid grid, CSGPrimitive primitive)
2:   for all  $v \in \text{grid.voxels}$  do
3:     if  $v$  is inside primitive then
4:        $v.\text{insidePrimitive} \leftarrow \text{true}$ 
5:        $v.\text{material} \leftarrow \text{primitive.material}$ 
6:     else
7:        $v.\text{insidePrimitive} \leftarrow \text{false}$ 
8:     end if
9:   end for
10:  for all  $e \in \text{grid.edges}$  do
11:    if  $e.\text{voxels}[0].\text{insidePrimitive} \neq e.\text{voxels}[1].\text{insidePrimitive}$  then
12:       $\text{intersection} \leftarrow \text{FINDINTERSECTION}(e, \text{primitive})$ 
13:      if  $e.\text{voxels}[0].\text{insidePrimitive} \cap \text{intersection.distance} \geq$ 
          $e.\text{voxels}[0].\text{intersections}[e.\text{axis}]$  then
14:         $e.\text{voxels}[0].\text{intersections}[e.\text{axis}] \leftarrow \text{intersection.distance}$ 
15:         $e.\text{voxels}[0].\text{normals}[e.\text{axis}] \leftarrow \text{intersection.normal}$ 
16:      else if  $e.\text{voxels}[1].\text{insidePrimitive} \cap \text{intersection.distance} \leq$ 
          $e.\text{voxels}[1].\text{intersections}[e.\text{axis}]$  then
17:         $e.\text{voxels}[1].\text{intersections}[e.\text{axis}] \leftarrow \text{intersection.distance}$ 
18:         $e.\text{voxels}[1].\text{normals}[e.\text{axis}] \leftarrow \text{intersection.normal}$ 
19:      end if
20:    end if
21:  end for
22: end procedure

```

---

First the algorithm for a CSG union operation tags all voxels of the voxel grid whether they are inside the CSG primitive. Then it iterates over all edges of the voxel grid and checks if the two voxels that form the edge have a different *insidePrimitive* state. If that is the case then the edge must intersect with the primitive's surface and the exact intersection is found using FINDINTERSECTION. Then the algorithm checks if any already existing intersections can and must be replaced by comparing their values. For multi-material voxel volumes some additional checks are required, mostly so that intersections are placed correctly at solid to solid material transitions. In addition to that, our implementation also gives each CSG primitive an axis aligned bounding box, such that only those voxels and edges inside the bounding box need to be checked.

With some minor adjustments, namely swapping the  $\geq$  and  $\leq$ , inverting the normals and changing the material assignment at the top to set to empty voxels, the algorithm for the CSG difference operation is obtained.

Previously it was mentioned that SDF's are well suited for CSG primitives. The reason for this is that they are versatile and checking whether a voxel lies inside the SDF or finding line to surface intersection points is simple. Checking whether a voxel is inside the SDF is as simple as checking whether the value of the SDF evaluated at said voxel is negative. Finding intersection points (FINDINTERSECTION) on an edge is done by running a binary search on the edge, starting with the edge start- and endpoints and repeatedly moving them closer together.

The binary search runs as long as the sign at the start- and endpoints are different, or until the value returned by the SDF for either start- or endpoint is within a certain threshold from zero. Both the start- and endpoints will thus converge towards the position where the SDF evaluates to zero, i.e. the SDF's surface.

This binary search can be optimized by using the values returned by the SDF for each iteration, since those values are a lower bound for the distance which the start- or endpoint can be moved along the edge without accidentally crossing the SDF's surface. Using this fact the adaptive binary search can make larger and more precise steps than a blind binary search. Consider a nearly linear surface perpendicular to the edge, like in Fig. 4.9:

the adaptive binary search will find the exact zero point after just one iteration because the two values returned by the SDF are the exact distances between the start- and endpoints and the surface. This may not always work for inexact SDF approximations since the returned values may not be actual lower bounds, hence the algorithm should still be able to use the regular binary search method as fallback for robustness.

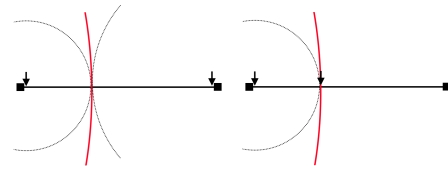


Figure 4.9: Adaptive binary search. Red circle: SDF's surface, Arrows: start- & endpoint, Black circles: circles with radius of SDF's value at start- or endpoint. The adaptive binary search finds the zero point from the initial state (left) after just one iteration (right).

#### 4.2.4 Rendering

The voxel polygonizer already does most of the heavy lifting for the voxel rendering. Once the voxel polygonizer has converted the voxels into a triangle mesh the rendering is done using the typical mesh rasterization techniques, provided by Unity, and shaders. The voxel materials and their colors are packed into the 32 bit RGBA color attachment of the mesh vertices. The first three components, RGB, are used to store the voxel color, the fourth component A for the voxel material ID. Since our voxel meshes are not UV mapped, i.e. they do not contain information about where the textures should be sampled, we have resorted to a tri-planar texture mapping shader to texture them. These kinds of shaders use the position of a mesh's pixels to determine where to sample a texture, without the need of any explicit UV mapping. In order to achieve a realistic looking texture mapping the tri-planar texture mapping shader projects an infinitely repeating texture from the X, Y and Z axes onto the mesh like shown in Fig. 4.10. The weight, i.e. how visible it is, of each texture projection is determined by the mesh's normals. For example the more a normal points along the X axis, the stronger the X axis texture projection is weighted, and the less the other two. The result of this is a mesh that is usually textured nicely on all sides. In certain situations, especially when the weights of

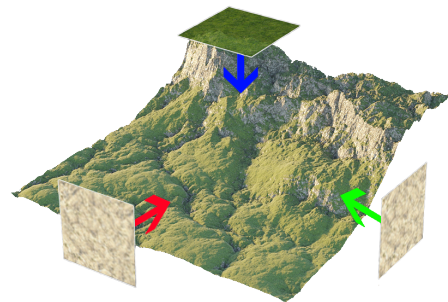


Figure 4.10: Tri-planar texture mapping of a mesh. Note how the hillsides of the terrain are rocky and the top is grassy. Adapted from Corona Renderer<sup>11</sup>.

<sup>11</sup> [https://corona-renderer.com/blog/wp-content/uploads/2017/04/Triplanar\\_01.jpg](https://corona-renderer.com/blog/wp-content/uploads/2017/04/Triplanar_01.jpg)

two texture projections are similar, e.g. for  $45^\circ$  diagonal parts of the mesh with respect to axes, there can be unnatural overlapping of two texture projections. This however is only a problem for textures with obvious structures, e.g. bricks, as opposed to natural or noisy textures like e.g. dirt or grass.

#### 4.2.5 Voxelizer

The precursor of our application already had an implementation of a voxelizer, however throughout the development of this thesis the voxelizer was rewritten to scale better with high triangle counts.

The implementation of a basic voxelizer is described in Alg. 5. This algorithm however will only work properly for meshes without holes.

---

**Algorithm 5 Voxelize.** *Voxelizes the given mesh into the voxel grid. FILLSECTION fills the voxel grid along and inside the given section with the given material and also sets the voxel Hermite data at the section's start and end points.*

---

```

1: procedure VOXELIZE(VoxelGrid grid, Mesh mesh, Material material)
2:   for all axis  $\in \{X, Y, Z\}$  do
3:     for all voxel  $\in$  grid.sideVoxels[axis] do
4:       for all section  $\in$  FINDSECTIONS(axis, voxel.position, mesh) do
5:         FILLSECTION(grid, section, material)
6:       end for
7:     end for
8:   end for
9: end procedure

```

---



---

**Algorithm 6 FindSections.** *Finds all intersections of a line with the mesh and then returns all sections that are between an ingoing and outgoing intersection, i.e. inside the mesh.*

---

```

1: procedure FINDSECTIONS(Axis axis, Position position, Mesh mesh)
2:   line  $\leftarrow$  CREATELINE(position, position + axis * gridSize)
3:   intersections  $\leftarrow \emptyset$ 
4:   for all triangle  $\in$  mesh do
5:     intersections  $\leftarrow$  intersections  $\cup$  FINDINTERSECTION(line, triangle)
6:   end for
7:   SORT(intersections)
8:   sections  $\leftarrow \emptyset$ 
9:   previous  $\leftarrow \emptyset$ 
10:  inside  $\leftarrow$  false
11:  for all intersection  $\in$  intersections do
12:    if  $\neg$ inside  $\cap$  previous  $\neq \emptyset$  then
13:      sections  $\leftarrow$  sections  $\cup$  CREATSECTION(previous, intersection)
14:    end if
15:    previous  $\leftarrow$  intersection
16:    inside  $\leftarrow \neg$ inside
17:  end for
18:  return sections
19: end procedure

```

---

Essentially the algorithm checks the grid lines perpendicular to the +X, +Y and +Z faces of the voxel grid for intersections with the mesh. Doing this for all the three grid faces will ensure that no triangles or voxels to be filled are missed. The found intersections are then sorted



from minimum distance from grid face to maximum distance. When iterating over these sorted intersections the algorithm alternates between inside the mesh and outside the mesh, since for each incoming intersection the next intersection must be an outgoing intersection. All voxels inside such a pair of ingoing and outgoing intersections are then filled with a chosen material. For the first and last voxel in such a section the algorithm also needs to set the Hermite data (i.e. the voxel intersection values and normals).

We have found that in practice some triangles seem to be missed during the intersection checks due to floating point precision errors. To address this problem our implementation performs a post-processing hole patching algorithm that checks for missing normals in the voxel Hermite data and then obtains them by sampling from the closest triangle to the hole's position.

Note that the complexity of the algorithm, disregarding `FILLSECTION`, is  $O(N^2 * K)$  where  $N$  is the voxel grid size and  $K$  the triangle count of the mesh, since for each side voxel (i.e. those touching one of the grid's +X, +Y or +Z faces,  $3 * N^2$  in total) the algorithm needs to check all triangles. For large  $N$  and  $K$  this poses a problem. Hence as improvement our new algorithm first runs Alg. 7.

---

**Algorithm 7 AssignTriangles.** *Projects the AABB's of all triangles onto all three voxel grid faces and then assigns the triangles to the bin of the according face and position.*

---

```

1: procedure ASSIGNTRIANGLES(Bins bins, Mesh mesh)
2:   for all axis  $\in \{X, Y, Z\}$  do
3:     for all triangle  $\in$  mesh do
4:       box  $\leftarrow$  CREATEAABB(triangle)
5:       area  $\leftarrow$  PROJECTONGRIDFACE(axis, box)
6:       for all position  $\in$  area do
7:         bins[axis][position] = bins[axis][position]  $\cup$  triangle
8:       end for
9:     end for
10:  end for
11: end procedure

```

---

Following that Alg. 6 is then modified to only iterate over the triangles in  $\text{bins}[\text{axis}][\text{position}]$ . In practice this provides a good speed improvement, although at the cost of more memory, because Alg. 6 will only have to check a tiny fraction of all the triangles for each position.

In our implementation `FINDSECTIONS` runs in parallel for all the voxels touching one grid face since all the voxels along one of the face's perpendicular grid lines can be written to independently in parallel. This is done three times, once per grid face, because otherwise the algorithm could potentially write to the same voxel multiple times at the same time, causing a write conflict.

## 4.3 VR Sculpting

### 4.3.1 Sculpting features

General overview of capabilities and functionality, UIs, SteamVR Plugin, etc.

### 4.3.2 Brush properties menu

Functionalities, color selection (why HSV: you can see most colors at once, as opposed to RGB sliders)

### 4.3.3 Custom brush editing menu

Explain custom brush tool, why it exists, etc.

# 5

## Evaluation

### 5.1 Technical Evaluation

Voxelizer, polygonization, queries, precision vs. recall?, etc.

### 5.2 User Evaluation

#### 5.2.1 Structure

#### 5.2.2 Results

TBD after evaluation

# 6

## Discussion

### 6.1 Conclusion

### 6.2 Lessons learned

### 6.3 Future work

Performance, LODs/Octrees (SVO)/memory use (e.g. run length encoding), meshes as brushes (using voxelizer), saving/loading sculptures & custom brushes, multiple sculptures at once, splitting sculptures

## **Bibliography**



## Appendix

### A.1 User Evaluation Questionnaire

The purpose of this user evaluation is to gather insight and feedback about the current implementation of the sculpting and querying functionality and the impression it has on users. The results will be helpful in identifying shortcomings and to improve the user experience.

If at any point you feel unwell or nauseous while using the VR headset please let me know. That can be a common reaction when not used to VR or when the program is not responsive enough. Each time before moving on a next task please press the "Reset Scene !" button in the ingame "Query & Utilities Menu" to reset the scene.

#### Background

1: not at all, 2: slightly, 3: moderately, 4: very, 5: extremely

1. How experienced are you with Virtual Reality? .....	<input type="text" value="1"/>	<input type="text" value="2"/>	<input type="text" value="3"/>	<input type="text" value="4"/>	<input type="text" value="5"/>
2. How experienced are you with 3D sculpting applications? .....	<input type="text" value="1"/>	<input type="text" value="2"/>	<input type="text" value="3"/>	<input type="text" value="4"/>	<input type="text" value="5"/>

## Sculpting

1: very easy, 2: easy, 3: neutral, 4: difficult, 5: very difficult

**3.** Read the controller hints to become familiar with VR and the control scheme. Disable the controller hints once you're ready. Please note that you can activate these hints again at any time if necessary.

Time limit: 5min. ....

Feedback:

---

---

**4.** Select the sphere brush and place it in the world to create a shape or simple sculpture using the 'Add' mode.

Time limit: 5min. ....

Feedback:

---

---

**5.** Select a brush and create a shape, then select another brush and remove a piece of your sculpture with it using the 'Remove' mode.

Time limit: 5min. ....

Feedback:

---

---

**6.** Select a brush and create a shape, then pick another colour and colour a piece of your sculpture using the 'Replace' mode.

Time limit: 5min. ....

Feedback:

---

---

**7.** Select a brush and create a shape, then pick another material (i.e. texture) and change the material of a piece of your sculpture using the 'Replace' mode.

Time limit: 5min. ....

1	2	3	4	5
---	---	---	---	---

Feedback:

---

---

**8.** Create your own brush (with at least two primitives) using the custom brush editor and then use your own brush to create a shape.

Time limit: 8min. ....

1	2	3	4	5
---	---	---	---	---

Feedback:

---

---

## Querying

1: very easy, 2: easy, 3: neutral, 4: difficult, 5: very difficult

**9.** Select a brush and create a shape, then use the query menu to run a similarity search.

Time limit: 6min. ....

1	2	3	4	5
---	---	---	---	---

Feedback:

---

---

**10.** Select a brush and create a shape, then use the query menu to run a similarity search. After that, pick one of the results and voxelize it into the world.

Time limit: 6min. ....

1	2	3	4	5
---	---	---	---	---

Feedback:

---

---



**11.** Select a brush and create a shape, then use the query menu to run a similarity search. After that, pick one of the results and voxelize it. Using a brush, remove a piece of it and then run a similarity search for the modified sculpture.

Time limit: 8min. ....

Feedback:

---

---

**12.** That was all, thank you! If you feel like playing around some more with the program feel free to do so for a couple more minutes. On the next page you can give general feedback.

Time limit: 5min.

**General feedback**

**13.** If you have any additional remarks or suggestions for improvements please write them down here.

---

---

---

---

---

---

---

---

# Declaration on Scientific Integrity

## Erklärung zur wissenschaftlichen Redlichkeit

includes Declaration on Plagiarism and Fraud  
beinhaltet Erklärung zu Plagiat und Betrug

**Author — Autor**

Samuel Börlin

**Matriculation number — Matrikelnummer**

16-051-716

**Title of work — Titel der Arbeit**

3D model retrieval using Constructive Solid Geometry (working-title)

**Type of work — Typ der Arbeit**

Bachelor thesis

**Declaration — Erklärung**

I hereby declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged. I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

Hiermit erkläre ich, dass mir bei der Abfassung dieser Arbeit nur die darin angegebene Hilfe zuteil wurde und dass ich sie nur mit den in der Arbeit angegebenen Hilfsmitteln verfasst habe. Ich habe sämtliche verwendeten Quellen erwähnt und gemäss anerkannten wissenschaftlichen Regeln zitiert.

Basel, Hand-In-Date

Missing: Date

---

Signature — Unterschrift