University of Basel

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
16-051-716

Hand-In-Date

Abstract

Table of Contents

1 Introduction

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.

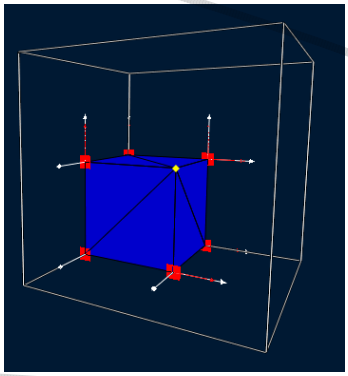

Figure 3.1: A polygonized voxel cell (white) containing the polygon surface of a cube's corner (blue) with a sharp feature (yellow).

Marching Cubes [] (MC), a primal contouring algorithm, checks whether the material at each corner of the voxel 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 [] (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 [] (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.

3.5 CSG

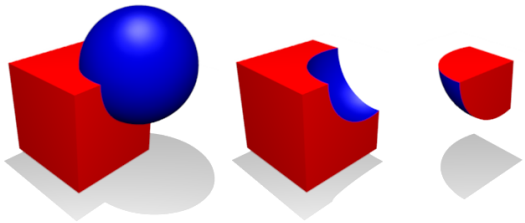

Figure 3.2: Boolean union (left), difference (middle) and intersection (right) of a cube and sphere primitive. Adapted from Wikipedia[1].

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

[1] https://en.wikipedia.org/wiki/Constructive_solid_geometry

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[2].
SDFs are well suited as a representation of CSG primitives because one can approximate their union and difference by taking the minumum, 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 $f \colon \mathbb{R}^3 \mapsto \mathbb{R}$, 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 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 Hermite data. Since Hermite data contains not only the voxel material, but also the surface normals, we can reconstruct the mesh from the Hermite data voxels while keeping most of the mesh's features intact, given a large enough voxel grid with sufficient resolution. In practice a resolution of 128x128x128 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 Hermite data voxel grid.

Figure 3.3: The Stanford Armadillo mesh[3](left) and its voxel representation at a resolution of 128x128x128 voxels (right).

[2] https://iquilezles.org/www/articles/distfunctions/distfunctions.htm
[3] http://graphics.stanford.edu/data/3Dscanrep/

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

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 [] 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 [] 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

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 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 [], which was used for this 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

4.1.3.1 Sample generation

Many isosurface extraction algorithms [], such as the non-adaptive CMS [] 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 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. Before any sample points are generated the model is first scaled such that its total surface area equals 100. After that the sample points are generated: 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. 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

Missing before

Missing abbr be

the absolute Gaussian curvature over the model's surface.

4.1.3.2 Sample Selection

Once the initial samples have been generated they are filtered and tested for certain criteria. These criteria differ between shape and color samples.

Shape Samples These samples should convey relevant information about the geometric shape. Previous work done by [] has shown that mean curvature is a good indication for visual saliency. The mean curvature of the mesh vertices is computed using Taubin's method []. 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 Laplacian smoothing [].
[]

Color Samples

4.1.4 Comparing features for similarity

L2 distance, Jensen-Shannon divergence, χ^2 distance

4.1.5 RESTful API

OpenAPI, 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 []. 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 []. Hermite data, first described by [], consists of two attributes: a material, exact intersection points and normals. Fig. 4.1 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.

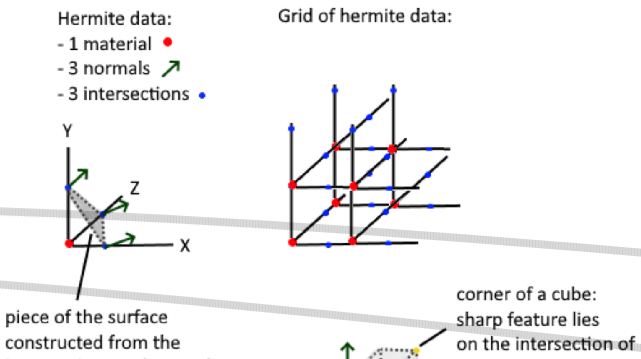

Figure 4.1: Hermite data.

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. Instead of multi-dimensonal arrays we use one contiguous linear array 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 32x32x32. 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. 4.2 shows the quantized Hermite data structure in more detail.

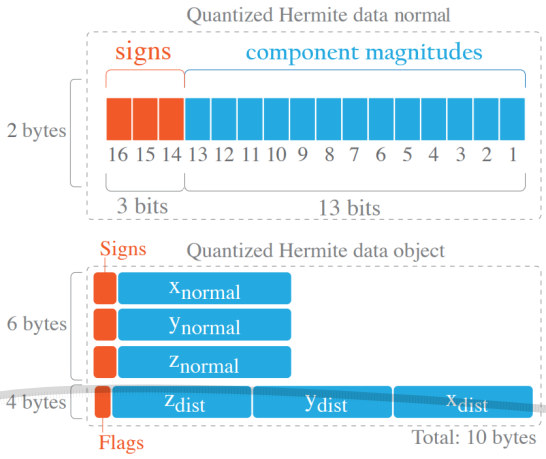

Figure 4.2: Quantized Hermite data structure. Top: a normal's component magnitudes are packed into 13 bits plus 3 bits for the signs. One normal thus fits into two bytes. Bottom: the entire quantized Hermite data object. x, y, z_{dist} refer to the intersection points. Adapted from [XXX].

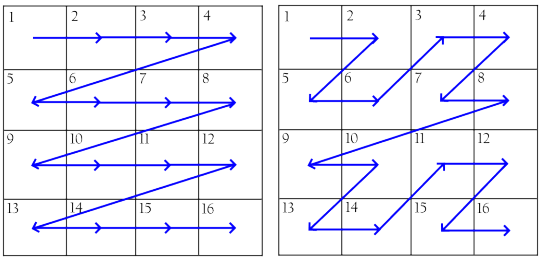

Figure 4.3: Simple linear indexing (left), Z-Order indexing (right). Each square represents a voxel in a 2D array. The blue line represents the order in which they are stored in main memory.

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 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. 4.3 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, spacially 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[4] were used and ported to C#. Unfortunately in our case using Z-Order indexing has made no significant difference, perhaps due to the rather small 32x32x32 chunk size.

Missing

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 contouring algorithms we have decided to use the Cubical Marching Squares algorithm described by [].

Missing CMS

4.2.2.1 Multi-material extension for CMS

Algorithm

4.2.2.2 Lookup table based implementation

Lookup table generator, lookup table based algorithm, limitations (time, multi-material support)

4.2.3 CSG operations on hermite data

How union and difference operations work on hermite data, algorithm

4.2.4 Rendering

Vertex colors encode material (RGB, A=texture id), texture array, triplanar texturing shader

4.2.5 Voxelizer

Purpose, explain method used for voxelization (assigning triangles to bins, patching holes, etc.), use of job system

[4] https://github.com/Forceflow/libmorton

4.2.6 SDFs

Implementation, arbitrary linear transformations by using the inverse to transform space instead of SDF

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

A
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 X button to reset the scene.

Missing: V
ton?

Background

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

1. How experienced are you with Virtual Reality?	1	2	3	4	5
2. How experienced are you with 3D sculpting applications?	1	2	3	4	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. Time limit: 5min.	1	2	3	4	5

Feedback:

_____

_____

4. Select the sphere brush and place it the world to create a shape or simple sculpture using the 'Add' mode.
Time limit: 3min. 1 2 3 4 5

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: 3min. 1 2 3 4 5

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: 3min. 1 2 3 4 5

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: 3min. 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: 5min. 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 place it in the world. Using a brush, remove a piece of it and then run a similarity search for the modified sculpture.
Time limit: 8min. 1 2 3 4 5

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. Further below you can give general feedback.
Time limit: 5min. 1 2 3 4 5

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

Signature — Unterschrift