

Level-of-Detail Independent Voxel-Based Surface Approximations

Author
Robbin Marcus
ICA-3850366

Supervisor
Amir Vaxman



Figure 1: Voxel models voxelized up to 2^{12} resolution in all axes. The complete scene would be nearly 12GB using sparse voxel octrees, exceeding memory capacity of nearly all graphics cards at time of writing. Rendered above are the models simplified using our adaptive simplification. We can store visually comparable models of the same scene in 2GB.

Utrecht University
March 31, 2017

Abstract

We present a voxel-based surface representation that contains level of detail (LOD) independent surfaces. We approximate the surface in every voxel by storing scalar data in every corner. The representation is close to the original surface in terms of contour information. In order to preserve the contour as accurately as possible, we introduce a new error metric that operates on the voxels. By simplifying voxels that contain the least amount of contour information, we achieve a voxel structure with low memory requirements. Our error metric correlates the level of detail with the amount of error of the surface approximation. We can create a model to a specific level of detail by constraining either maximum memory requirements, or the maximum allowed error of the approximation, with respect to the original model.

Contents

1	Introduction	5
1.1	Scalar fields, vector fields and voxels	6
1.2	Problems with voxel-based rendering	7
1.3	Research Proposal	7
1.4	Contributions	8
1.4.1	Application	8
1.5	Structure	9
2	Related work	10
2.1	Polygon simplification	10
2.2	Volume representations	11
2.2.1	Isosurface rendering	11
2.2.2	Sparse Voxel Octree (SVO)	12
2.2.3	Vector to closest point (VCP) octree	13
2.2.4	Gigavoxels	14
2.3	Wavelets	15
2.3.1	Biorthogonal loop-subdivision wavelets	15
2.3.2	Wavelet rasterization	15
2.3.3	Adaptive multilinear tensor product wavelets	17
2.4	Summary	17
3	Methodology	18
3.1	Research question	18
3.2	Assumptions	18
3.3	Definitions	18
3.4	Representation	22
4	Algorithms	23
4.1	Voxelization	23
4.1.1	Pre-processing	23
4.1.2	Attribute generation and attribute merging during voxelization	24
4.1.3	Post-processing	24

4.2	Simplification	24
4.2.1	Corner simplification	25
4.2.2	Moving least squares simplification	25
4.2.3	Adaptive sampling	25
4.3	Offline LOD optimization	26
4.3.1	LOD stitching	26
5	Implementation	28
5.1	Voxelization	28
5.2	Simplification	28
5.2.1	Adaptive sampling	28
5.2.2	Solving duplicate voxel corners with least squares	29
5.2.3	Solving sparse voxels	29
5.3	Surface extraction	29
5.3.1	Ray-isosurface intersection	30
5.3.2	Normal approximation	30
5.4	Optimizations	31
5.4.1	Leaf array	31
5.4.2	Dynamic LOD	32
6	Results	33
6.1	Datasets	33
6.2	Root-mean-square error	34
6.3	Memory compression	35
6.4	Frames per second	36
6.4.1	LOD-optimized and unprocessed models	37
6.4.2	Leaf array	37
6.4.3	Dynamic LOD	38
6.5	Cross comparisons	38
6.6	Catmull-Clark subdivision rules	40
6.7	Error quadrics in triangle meshes	40
7	Discussion	42
7.1	Conclusion	42
7.2	Future work	42
7.2.1	Prolongations	42
7.2.2	In-place optimized LOD generation	42
7.2.3	Dynamic octree depth	43
7.2.4	Additional compression	43
A	The M matrix in Mathematica	47
B	Expanding scalar fields	48
B.1	Completed points and direct neighbors	48

B.2	Smallest distance to the surface	48
B.2.1	Four voxel faces	48
B.2.2	Smallest distance	49
B.2.3	Correctness	49

Acknowledgements

I would like to express my deepest appreciation to my supervisor Amir Vaxman throughout this thesis. Without his endless support and expertise in this field, this dissertation would not have been possible.

I would also like to thank Eric Marcus for the lengthy discussions. Even though his field of expertise is theoretical physics, he had numerous of insightful remarks regarding the topic of this thesis.

1 | Introduction

One of the goals of computer graphics research is real-time photo-realistic rendering. This is a very challenging problem due to the complexity of multi-scale scenes and detailed objects.

Objects are mainly represented as triangle meshes, since triangles are the simplest piecewise-linear representation of geometry. The biggest advantages of triangle meshes are easy rendering pipelines due to their trivial planar shape and a small memory footprint. The triangles can be stored efficiently using compression schemes like triangle strips or triangle fans. More fine-grained details can be added using textures for color and illumination properties.

To achieve photo-realism, many millions of primitives are required to represent all the detailed features of surfaces and complex shapes. Rendering methods tend to become inefficient for complex scenes, because the rendering cost is proportional to the number of primitives. Small-sized primitives can also lead to aliasing artifacts, which require even more calculation to avoid. This is also unfortunate because they add very little to the result. In addition, costly intersections between triangles and rays dominate the rendering time.

A common approach to handle the aliasing problem and reduce rendering time is to simplify the geometry to the rendering resolution. Reducing the amount of triangles in a mesh based on viewing distance is done using geometric level-of-detail (LOD) approaches. Downsides of this method are additional memory requirements, difficulty to control the LOD, and popping artifacts when transitioning between LODs. Figure 1.1 shows different levels of detail for a triangle mesh.

Beside triangle meshes, uniform grid data structures are a good alternative to surface rendering. Volumetric representations like axis-aligned cubes have a cheap ray intersection test and therefore more efficient to render. In the next sections we show advantages and disadvantages of alternative surface representations.

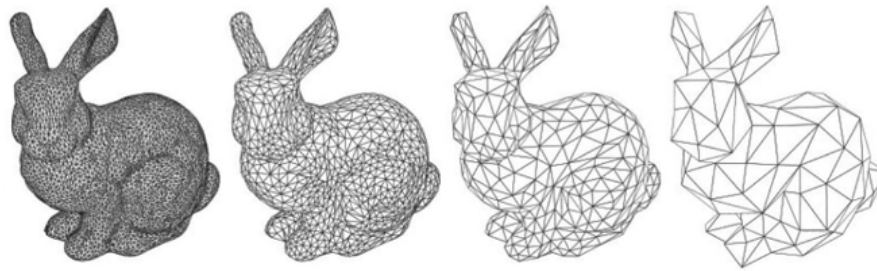


Figure 1.1: The stanford bunny represented as a triangle mesh in multiple level of details.

1.1 Scalar fields, vector fields and voxels

A discrete scalar field is an evenly spaced grid with a scalar value on every grid point. A *level set* is a subdomain where the function value is constant. In 2D, these level sets are isolines, and in 3D they are commonly denoted as isosurfaces. A *vector field* assigns a vector per point, and is used to store directional data (such as surface normals). Scalar and vector fields are commonly used in data visualization, for example isolines are used to show differences in height on a map. Figure 1.2 shows the difference between scalar and vector fields.

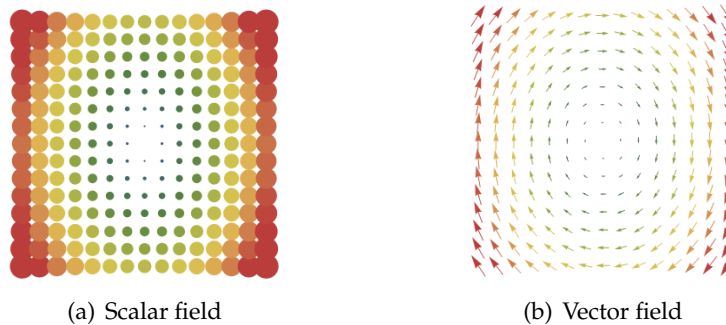


Figure 1.2: **Left:** A scalar field representing values as differently sized discs. **Right:** A vector field also stores the direction in addition to the values of the scalar field.

Voxels are the three dimensional equivalent of pixels: cubes. Similar to triangles, voxels can be used to store arbitrary data like geometrical properties.

Voxels have many applications. Some practical examples are storing volumetric data for 3D scans, and simulating fluids like smoke and water. The volumetric data in voxels can be rendered using ray marching [IKLH04]. Voxels are also used as intermediate representations in the production pipelines. This is mainly because constructive solid geometry (CSG) operations work efficient on a voxel grid [JBS06], as spatial querying can be done effectively.

Voxel models can also be combined with triangle models, using the best properties of both methods: rasterization methods for geometry close to the viewer, and ray casting methods to trace isosurfaces further away [GM05,RCBW12]. This reduces the amount of overshading introduced by rasterizing small triangles, which can grow up to eight times per pixel [FBH*10]. In this thesis, we will only consider pure voxel representations.

1.2 Problems with voxel-based rendering

Despite many advantages, the voxel representation has drawbacks and limitations. Triangles are natively supported on dedicated graphics hardware. Voxel rendering cannot be done using the same pipeline. Instead, voxels are handled on the GPU using programmable shaders [LK10]. Another problem is the memory requirement of voxel data structures, which is most often $O(n^3)$ where n is the amount of voxels per dimension. The voxel structure has to be readily available in the memory of the graphics card in order to render real-time. Because of this, voxels are not commonly used in games or other real-time rendering applications.

According to [CNLE09], there are three major issues to be solved in order to make use of voxels instead of triangles:

- Rendering the voxel structure efficiently
- Storing a pre-filtered voxel representation
- Store large amounts of voxels efficiently

The next chapter shows related work, specifically section 2.2 shows different approaches to above problems. We mainly focus on the last point. Adding more details to a voxel structure makes the storage problem increase exponentially. However, the error in surface approximation does not diminish with the same rate. There are a lot of very small voxels that contribute very little to the geometric structure, whilst taking up most of the memory. To tackle this problem, we require a storage method that would also balance the structure based on accuracy of the surface representation.

1.3 Research Proposal

We propose a method to approximate a triangular model, represented implicitly, to varying levels of detail. The approximation is done by an incremental simplification of function values, based on voxel grids. The function values are decomposed hierarchically as a multi-resolution function, where the error objective is chosen to minimize the simplification error in the surface itself.

The domain of the scalar field is a voxel grid, represented as an octree, which is a recursive uniform subdivision of cubes into eight smaller cubes. We start with an initial guess for the depth of the subdivision. The multiresolution transformation uses a novel error metric to simplify a group of voxels to their parent. The error metric measures how much a coarse voxel differs from the original fine voxels. The simplification is designed to minimize this metric as much as possible.

1.4 Contributions

- We present a memory efficient voxel structure, that can represent very detailed models.
- We achieve a good surface approximation with less voxels, by introducing a novel metric that measures the error of a simplification, and creating adaptive simplifications that minimize this metric.
- We detail a novel way to convert triangle meshes into implicit representation, which stores voxels on different depths based on the error in the implicit representation.

1.4.1 Application

Our method provides the following applications: we can convert arbitrary triangle meshes to a sparse voxel octree structure storing a C^0 -continuous signed distance field. We provide algorithms to simplify this distance field to a sparser representation using offline processing steps. We can render voxels from different levels of detail using ray tracing algorithms. We achieve a watertight model consisting of multiple levels of detail by providing a stitching algorithm that combines multiple levels of detail together.

The offline simplification couples a multiresolution transformation to the surface simplification. We capture details as much as possible by storing implicit function values. The higher levels of the hierarchy contain a sparser representation which is as close as possible to the details with regard to an error metric.

The advantage of our voxel hierarchy is that we do not require different versions of a model for different levels of detail. The LOD can be determined by the distance between the viewer and the model, and the surface is represented with the necessary details without computational overhead. Other advantages include the possibility for interpolation and filtering.

1.5 Structure

The thesis is structured as follows: Chapter 2 discusses previous work tackling problems with voxel-based rendering. Chapter 3 presents the theoretical framework, and the derived algorithms are presented in Chapter 4. Chapter 5 shows implementation details. The results are listed in Chapter 6 and we conclude with a discussion in Chapter 7.

2 | Related work

In this chapter we look at previous work related to voxel data structures and memory compression of those structures. We look at simplification algorithms for triangle models in section 2.1. We show voxel-based volume representations in section 2.2 followed by related work using wavelets in section 2.3.

2.1 Polygon simplification

In Luebke’s survey [Lue01] we see several examples of triangle model simplifications. In this thesis, we use the algorithms below to compare our method against other simplification algorithms. More specifically, the quadric error meshes will determine the quality of the simplification.

There are several techniques to simplify a triangle model. For example, the vertices of a cluster of triangles can be merged together, and it is also possible to merge two vertices together by collapsing an edge. Both techniques are illustrated in 2.1.

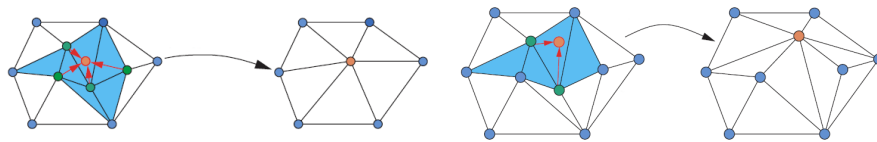


Figure 2.1: **Left:** Merging four vertices together simplifies the whole cluster. **Right:** Merging a pair of vertices together.

We describe a simplification algorithm for voxel models that originate from triangle models. The survey shows many different methods to simplify, or remesh an existing triangle model. Some of the relevant algorithms to simplify triangles are briefly summarized below:

- Multiresolution analysis (MRA): by using wavelets to guide a subdivision process, a polygonal model can be interpolated smoothly between different LODs.

MRA creates a mesh by growing Voronoi-like regions across the triangles. A Delaunay triangulation of those regions forms the base mesh.

- Quadric error metrics: by iteratively merging pairs of vertices together, the model can be simplified. Candidates are found by using a quadric error metric: a 4x4 matrix that represents the sum of squared distances from the vertex to the planes of neighboring triangles.

2.2 Volume representations

Volumetric data storage is a classical and common recurring problem. There have been many attempts to compress volumetric information over the years. We first discuss triangle conversions, followed by implicit and directly rendered voxel structures.

2.2.1 Isosurface rendering

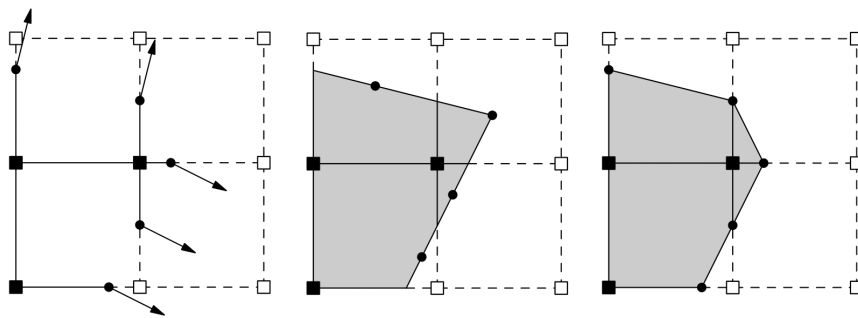


Figure 2.2: The approximation introduced by using marching cubes compared to dual contouring. **Left:** The Hermite data presented to the algorithm. **Middle:** The output from the dual contouring algorithm, a sharp corner. **Right:** The marching cubes approximation can only represent a linear zero-order interpolation between two surface points.

Early solutions to make volume rendering feasible were to convert the volumetric model to a triangle mesh. The most well-known solution is marching cubes [LC87]. Ju et al [JLSW02] present dual contouring of Hermite data. Dual contouring presented a qualitative improvement over marching cubes when creating a mesh from volumetric data. The algorithm reduces the surface approximation error per voxel to a least-squares problem. A vertex is placed in such a position that when connected to neighboring vertices, the resulting polygon mesh is a good approximation of the original volumetric data. The difference between dual contouring and marching cubes becomes clear in figure 2.2. Marching cubes and dual contouring both present algorithms to extract an isosurface from a discrete scalar field to a polygon. We will extract the surface from a scalar field, but only implicitly. This allows for direct visualization using ray casting.

2.2.2 Sparse Voxel Octree (SVO)

Laine et al. [LK10], extended in [LK11], present a data structure to store voxel data, called efficient sparse voxel octree (ESVO). We use a pointerless, memory saving variant of their SVO to store voxels more sparsely. We tackle some of the flaws of the ESVO: the lack of interpolation and filtering. They have to apply a blur in order to combine surface data from different voxels. We show that it is possible to maintain detail in various levels of detail, reducing memory requirements drastically.

The ESVO is not built to maximal depth. By providing each voxel with a contour, they compensate for missing details. The contour defines a set of parallel planes which bound the geometry in the voxel. If this contour suffices in representing the surface, the subdivision is stopped. A comparison between ordinary voxels and voxels with contours is shown in figure 2.3.

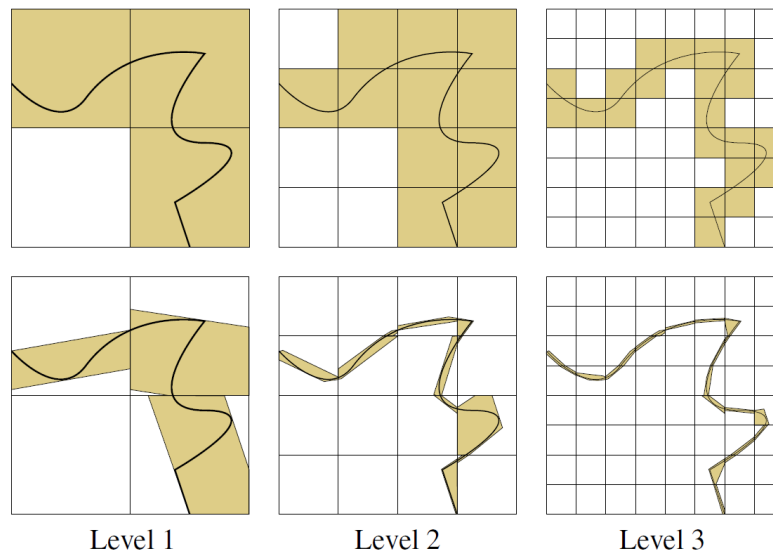


Figure 2.3: Top: Surface data encoded using regular voxelization. **Bottom:** The same surface encoded in voxels using contour data.

The SVO achieves sparseness by implementing eight pointers per octree node, pointing to all children. They use a "valid mask" (a bit mask) to tell which children contain geometry and a pointer to the first child in a list of children containing geometry. Using this method, the unique path describes the voxel spatially.

A benefit of the SVO, and voxels in general, is that the required information for triangle meshes can be stored on a per voxel basis. We can store a color and normal in every voxel and thus the texture- and normal-maps used in triangle meshes are not necessary for sparse voxel octrees. The major improvement over previous methods is the reduction in memory requirements to store large voxel meshes.

The downsides to this representation are the lack of interpolation and filtering. Interpolating between two levels in the octree is not possible, and produces popping artifacts when transitioning. Aliasing occurs at distance as well as close to the viewpoint even when using several samples per pixel. Screen-space blurring has to be applied to reduce these flaws.

We use the SVO data structure to store our signed distance fields sparsely. We store only the voxels that contain sign changes, and thus contain surface data. We overcome the shortcomings of this structure by merging different levels of details together using a stitching algorithm. We get a C^0 -continuous surface by merging the surface data together, and we can avoid the blurring.

2.2.3 Vector to closest point (VCP) octree

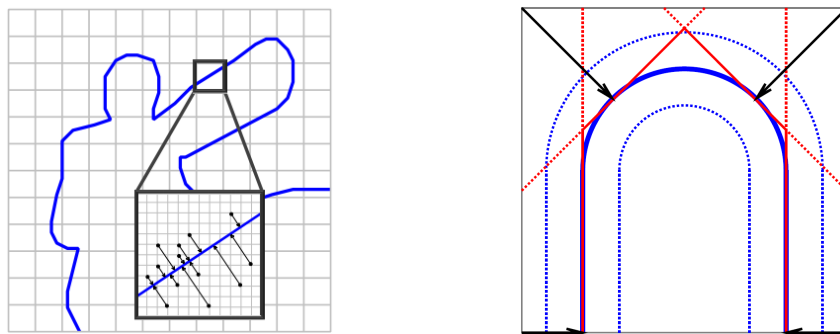


Figure 2.4: **Left:** Illustration of the VCP representation in 2D. At each grid point, a vector to the point closest to the original geometry (blue) is stored; this is visualized by a few such vectors (black). **Right:** Illustration of a feature (blue) stored in a VCP cell. The black arrows are the VCP vectors stored in every corner which save the feature as shown by the red lines.

The vector to closest point (VCP) [DW15] octree stores surfaces using a vector field. The advantage of a vector field over a scalar field is the ability to store a direction in addition to the distance. The main disadvantage is the required memory to store vector fields: every voxel requires 8 vectors, whereas in a scalar field only 8 scalar values are required.

An example of a feature stored by vectors is shown in figure 2.4. Saving the complete grid at full resolution would be very memory inefficient. The presented solution is to store the complete model in voxel format, and only save the high resolution VCP grid in a narrow band around the model. The internal voxels will save the minimum distance to the model in a single scalar value. Our structure does not require any distance values in internal voxels, and the surface data is stored sparsely in leaf nodes.

A large advantage of this method is that normals are not stored explicitly. They can be derived from interpolating the VCP vectors. The normal to the surface is the local

derivative of the surface plane, and is arguably the most significant factor in adding more details to the surface. The VCP vectors have a significant advantage over signed distance fields when approximating surface normals. In our structure we rely on finite differences to approximate surface normals.

They mention some optimizations to overcome the high storage of eight VCP vectors in the paper: compressing the VCP vector as one 32 bit integer value is possible by converting the vector into two spherical angles and a radius. The VCP vectors can be shared between two direct neighbors by storing them next to each other in memory as a sparse 3D texture. Every VCP vector can be shared with up to eight neighbors.

We mainly focus on the simplification of sparse voxel octrees, and in this paper they present a simplification algorithm that checks for merging opportunities if there are multiple leaf nodes with surface data. To test whether it is possible to merge all child nodes to a single cell, they compare interpolated vectors from the single parent cell with the exact vectors in the non-empty leaf cells. This comparison is computed with Euclidean distances between the original VCP vectors and their interpolated replacement vectors. If the maximum of these distances is below a threshold, the nodes are merged to a non-empty leaf node at the next level.

2.2.4 Gigavoxels

The objective of the gigavoxels [CNS*11] structure is to present a new approach to efficiently render large scenes and detailed objects in realtime. Their approach is based on a volumetric pre-filtered geometry representation, and an associated voxel-based approximate cone tracing that allows an accurate and high performance rendering of very detailed geometry.

They allow a quality/performance trade-off and exploit temporal coherence by loading required data only when necessary. The solution is based on an adaptive hierarchical data representation, depending on the current view and occlusion information.

Their structure is based on sparse voxel octrees. Instead of using standard nodes, their hierarchy is comprised of bricks: low resolution regular grids, usually the size of 8^3 voxels. This combines the profit of local cache efficiency with the small memory footprint of sparse voxel octrees.

Their structure can be rendered efficiently using approximate cone tracing (see Figure 2.5). We use an adaptation of approximate cone tracing to dynamically change the level of detail, based on the current camera view and projection.

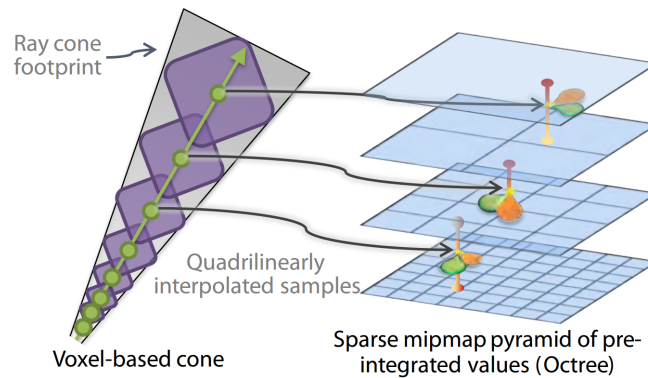


Figure 2.5: Approximate cone tracing footprint of a ray. Using the radius of the cone the intersected maximum voxel size can be determined.

2.3 Wavelets

2.3.1 Biorthogonal loop-subdivision wavelets

[Ber04] introduce biorthogonal wavelet construction for Loop subdivision, based on the lifting scheme. They apply a wavelet transform on meshes generated by Loop subdivision schemes, to store them more sparsely without computational overhead. The wavelets compute a local least squares fit when reducing the resolution, and convert geometric detail into sparse wavelet coefficients.

Their simplification is restricted to triangle meshes that are created by a Loop subdivision scheme. Our simplification will apply on voxels instead of triangle meshes. We can handle arbitrary cases by defining a prolongation that defines an approximate subdivision scheme to obtain a sparser function.

2.3.2 Wavelet rasterization

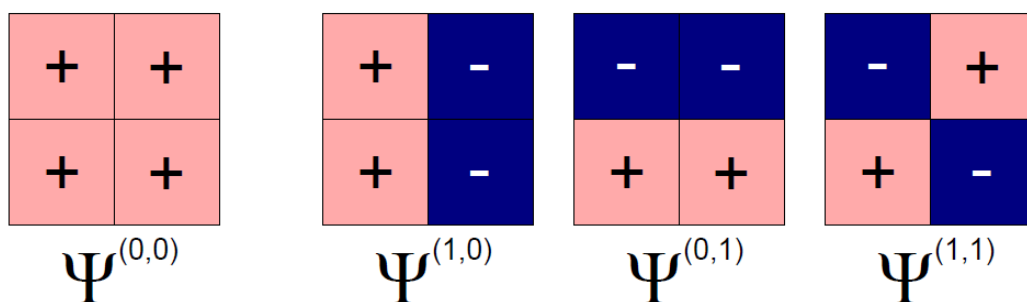


Figure 2.6: The 2d Haar basis functions. Each function is shown over the domain $[0, 1]^2$.

Manson and Schaefer [MS11] present a method for analytically calculating an anti-aliased rasterization of oriented triangle meshes in 3D by using wavelets. We use wavelets to compare different levels of detail, by comparing their respective function values.

The algorithm rasterizes multiple resolutions simultaneously using a hierarchical wavelet representation. They show that by using the Haar basis as wavelet, the result is equivalent to performing a box-filter to the rasterized image. In 3D, they compute the wavelet coefficients through analytic surface integrals over triangle meshes, and show how to do so in a computationally efficient manner.

Rasterizing objects is done by calculating the percent occupancy of voxels in a regular grid. If the area M is the set of points inside an object with boundary, represented as a set of edges, the function χ_M is defined as:

$$\chi_M(p) = \begin{cases} 1, & p \in M \\ 0, & otherwise \end{cases} \quad (2.1)$$

The Haar wavelet has small support functions. the scaling function ϕ is represented as:

$$\phi(t) = \begin{cases} 1, & 0 \leq t < 1 \\ 0, & otherwise \end{cases} \quad (2.2)$$

And the basis function ψ is given as:

$$\psi(t) = \phi(2t) - \phi(2t - 1) \quad (2.3)$$

Integrating over the domain of a triangle mesh is difficult, so in order to find the contribution for a single pixel value they use the divergence theorem to relate the integral over the triangle mesh to the integral over the boundary of the mesh. Because χ_M is constant over the interior and exterior, and wavelets have constant precision, the wavelet coefficients are only non-zero where the boundary intersects the basis function. This yields an adaptive quadtree that is refined only along the boundary of the polygon. The 2D Haar wavelets represents piecewise constant function on square domains (see Figure 2.6).

Our voxel structure is already an implicit function in square domain, and therefore we can employ the simplification directly to our representation. In [MS11], the Haar wavelets are used to find the result of the integral over a triangle mesh to a 2D pixel. We compute a result that is similar in the sense of a multiresolution representation, to find an approximation of the surface in a voxel.

2.3.3 Adaptive multilinear tensor product wavelets

Weiss and Lindstrom [WL16] use wavelets to represent a globally-continuous function, defined by the union of multilinear cells. The wavelets are formed by tensor products of linear B-splines. Similar to wavelet rasterization, the resulting function is represented as a regular refinement scheme such as a quadtree or octree. This allows for a sparse representation of the function, that can be evaluated at any point. Their approach allows them to build a mesh by selecting any subset of wavelets in the domain. By filtering wavelets on hierarchy depth, they can retrieve progressive level of detail extraction from the mesh.

Volume rendering using a regular grid is usually done on an edge-balanced mesh. Edge-balanced means that any edge of a cube in an octree is on the boundary from at most two consecutive octree levels. This approach results in a globally C^0 -continuous function. However, there are disturbing visual artifacts between neighbors (for example, two neighboring surfaces in marching cubes). They state that using a radial interpolant on the same mesh, the artifacts disappear. The reconstruction is equivalent to the interpolated linear B-spline wavelet approximation.

Downsides to the wavelet representation are the reconstruction phase when rendering the structure directly. Another limitation is that any modifications to the structure requires a global update of the entire support wavelet.

The difference between their work and ours is that they use B-spline surfaces and transform them to wavelets using prediction and update steps (w-lift and s-lift respectively). We stick to regular piecewise multilinear functions. They also store and render the wavelet function directly, which requires additional decoding overhead in rendering the function. We store only implicit values, and our wavelet transform is only visible by calculating the differences between hierarchy levels.

2.4 Summary

The related work shows LOD simplification for polygon meshes and several methods to store voxel models more sparsely. Previous work on voxel parametrization has shown to solve most of the problems with voxel based rendering as mentioned in section 1.2. There are different volume representations that can approximate a surface, but such methods are mostly constrained by the massive amount of memory required to store the model with all details.

We overcome this memory requirement by introducing a simplification algorithm that operates on voxels. By approximating the surface implicitly, we can define a hierarchy of function values. This hierarchy is built by coarsening voxels in which the function is smooth enough according to a novel error metric, as we describe in the following.

3 | Methodology

3.1 Research question

The main research question for this thesis is as follows: can we adapt the LOD structure to correlate well with surface approximation?

We answer this question by providing a method that can simplify an implicit voxel model, and offer a balance for the best fit: the best ratio of the amount of voxels versus the error in the surface approximation. In the next sections, we show definitions for our error metric and simplification, and in the following chapter we detail the structure completely.

3.2 Assumptions

In order to store the surface sparsely, we store only voxels that contain surface data. We make this distinction by looking at sign changes in the stored distance field (assuming our surface is the zero level set). If there is no voxel stored, there is no surface data. The second assumption is that the octree will contain only a single mesh, meaning that all surface data stored has to be continuous.

3.3 Definitions

A voxel is defined as an axis-aligned 3D cube. The surface is the zero level set of a signed distance function $\phi(x)$ is defined as follows:

$$\begin{aligned}\phi(x) &> 0 \text{ for } x \notin \Omega \\ \phi(x) &< 0 \text{ for } x \in \Omega \\ \phi(x) &= 0 \text{ for } x \in \partial\Omega\end{aligned}\tag{3.1}$$

Where Ω is the inner volume of the triangle mesh, and $\partial\Omega$ is the mesh itself. We define the discrete function f (and F in vector form) for a voxel as:

$$\begin{aligned} f &: V \rightarrow \mathbb{R} \\ F &: (f_0, f_1, \dots, f_n)^T \end{aligned} \quad (3.2)$$

Where V is the set of voxel vertices. We define a continuous function $f' : \mathbb{R}^3 \rightarrow \mathbb{R}$ as an *interpolation* of f , if $f'(V) = f(V)$. We look at functions that can be interpolated with values from the same voxel. That is, for given barycentric weights where $p = \sum_{v \in V} B_v(p)$, then $f'(p) = \sum_{v \in V} B_v(p)f(v)$. Note that the requirement of interpolation means the the Barycentric coordinates have the Lagrange property: only the relevant vertices are used for interpolation. For example, a point on one of the faces requires only four vertices instead of all eight. $B_v(p)$ is defined as a local function for every voxel. $B_v(p)$ can be written as a combination of first order Bernstein polynomials:

$$B(p) = \sum_{z=0}^1 \sum_{y=0}^1 \sum_{x=0}^1 b_{x,1}(p_x)b_{y,1}(p_y)b_{z,1}(p_z) \quad (3.3)$$

Where $b_{v,n}$ is a Bernstein basis polynomial of degree n .

We define two grids: a coarse and a refined one, so that the fine grid V_0 and the coarse grid V_1 hold $V_0 \subset V_1$. Every coarse voxel contains two fine voxels exactly along each dimension (and thus 8 in total). We index every corner similar to z-order curve: The first corner is indexed 0 and is the minimum of the grid. The index then increases in direction of ascending axes x , y and z . The z-order curves are stored as Morton codes [Mor66]: the mapping from a multidimensional space (3D) to one dimension while preserving locality of the data.

We define two operators that compute functions from the coarse to the fine grid and vice versa. The *prolongation* and the *coarsening* operators:

$$\begin{aligned} P_{1 \rightarrow 0} &: V_1 \rightarrow V_0 \\ C_{0 \rightarrow 1} &: V_0 \rightarrow V_1 \end{aligned} \quad (3.4)$$

These operators are consistent, meaning:

$$C(P) = Identity. \quad (3.5)$$

We define the error metric function E_X for a single voxel X as:

$$\begin{aligned}
E_X(f, g) &= |f' - g'|^2 \\
E_X(f, g) &= \int_V \left| \sum_{p \in V} B_v(p)(f(p) - g(p)) \right|^2 dV \\
E_X(f, g) &= (F - G)^T \left(\int_V B^T B dV \right) (F - G)
\end{aligned} \tag{3.6}$$

Where f and g are two functions in the same domain. F and G are the row vectors of all vertex values f and g respectively. F and G are of dimension $(n, 1)$ and the resulting matrix of $B^T B$ is of dimension (n, n) , where n is the amount of vertices in a voxel.

The residual energy between two functions on the mesh can then be calculated as:

$$E = \sum_X E_X \tag{3.7}$$

We derived the function E_X to a bilinear form on the functions f, g and integrals over the barycentric weights inside the voxel: $\int_V B^T B dV$. We can calculate the result of the integral beforehand as the integral over a unit cube for every function in the matrix: $\int_0^1 \int_0^1 \int_0^1 (B^T B) dz dy dx$. The Mathematica code we used can be found in [appendix A](#).

$$\begin{pmatrix}
\frac{1}{27} & \frac{1}{54} & \frac{1}{54} & \frac{1}{108} & \frac{1}{54} & \frac{1}{108} & \frac{1}{108} & \frac{1}{216} \\
\frac{1}{54} & \frac{1}{27} & \frac{1}{108} & \frac{1}{54} & \frac{1}{108} & \frac{1}{54} & \frac{1}{216} & \frac{1}{108} \\
\frac{1}{54} & \frac{1}{108} & \frac{1}{27} & \frac{1}{54} & \frac{1}{108} & \frac{1}{216} & \frac{1}{54} & \frac{1}{108} \\
\frac{1}{108} & \frac{1}{54} & \frac{1}{54} & \frac{1}{27} & \frac{1}{216} & \frac{1}{108} & \frac{1}{108} & \frac{1}{54} \\
\frac{1}{54} & \frac{1}{108} & \frac{1}{108} & \frac{1}{216} & \frac{1}{27} & \frac{1}{54} & \frac{1}{54} & \frac{1}{108} \\
\frac{1}{108} & \frac{1}{54} & \frac{1}{216} & \frac{1}{108} & \frac{1}{54} & \frac{1}{27} & \frac{1}{108} & \frac{1}{54} \\
\frac{1}{108} & \frac{1}{216} & \frac{1}{54} & \frac{1}{108} & \frac{1}{54} & \frac{1}{108} & \frac{1}{27} & \frac{1}{54} \\
\frac{1}{216} & \frac{1}{108} & \frac{1}{108} & \frac{1}{54} & \frac{1}{108} & \frac{1}{54} & \frac{1}{54} & \frac{1}{27}
\end{pmatrix} \tag{3.8}$$

Our goal is to find a coarsening operator that preserves the zero set (the surface) as much as possible. One straightforward way to approximate this, is to compare the coarsened-and-prolonged version of a function against the original. The objective can be formulated as follows. Our purpose is to find a coarse function F_1 in grid V_1 such that it is a good coarsened approximation of fine points F_0 . A good coarsening is defined such that the prolongation of F_1 leads to values in a fine grid, that are as close as possible to the values in F_0 .

To do this we use the error metric E as an energy function. We can now define the comparison between F_1 and F_0 using a prolongation:

$$F_1 = \operatorname{argmin}_X \sum E_X(P(F_1), F_0) \quad (3.9)$$

To find the minimum, we calculate the derivative of the energy function:

$$\begin{aligned} E &= (P(F_1) - F_0)^T \left(\int_p B^T B dp \right) (P(F_1) - F_0) \\ E &= (P(F_1) - F_0)^T M (P(F_1) - F_0) \end{aligned} \quad (3.10)$$

We can use the prolongation P as a matrix and expand the equation. When expanding, we can simplify parts of the equation with M . They can be rearranged since M is symmetric.

$$\begin{aligned} E &= (PF_1)^T M (PF_1) - F_0^T M (PF_1) - (PF_1)^T M F_0 - F_0^T M F_0 \\ E &= (PF_1)^T M (PF_1) - 2(PF_1)^T M F_0 + F_0^T M F_0 \end{aligned} \quad (3.11)$$

In matrix notation, the gradient of the above is:

$$\nabla_{F_1} E = 2P^T M P F_1 - 2P^T M F_0 \quad (3.12)$$

We can then formulate it as a system to solve for the coarse F_1 by rearranging:

$$\begin{aligned} 2P^T M P F_1 - 2P^T M F_0 &= 0 \\ P^T M P F_1 &= P^T M F_0 \end{aligned} \quad (3.13)$$

This is a least squares problem: $(Ax - b)^2 = 0$, which is already rearranged as: $A^T A x = A^T b$. The brute-force solution for x is then found by solving: $x = (A^T A)^{-1} A^T b$. This is a sparse system, and the system can be solved more efficiently using pre-factorization. Using found values for x we can find the best fit for F_1 . Note that if the energy is zero, an exact solution can be found with: $PF_1 = F_0$.

Note that in essence we are trying to find the inverse of the prolongation operator, but P is not an invertible matrix. We try to find the best solution by solving the least squares problem.

Even though the effect of the simplification is global, we can solve the system sparsely for a point f_p . The prolongation P is a local function, that allows us to form a sparse system for f_p . For example when using Bernstein polynomials, it is clear to see that coarse voxels which do not have f_p as one of their corners have no direct effect on the energy of the linear system of f_p .

3.4 Representation

Our voxel hierarchy is most similar to the SVO presented in section [2.2.2](#). We store a sparse representation of an octree, which contains only voxels that have surface data. In addition, every voxel that contains surface data stores scalar values at all corners. These values represent the minimum distance to the surface.

We use a pointerless SVO format, removing the built in external pointers as described for the ESVO structure [[LK10](#)]. While this reduces the lookup efficiency on the GPU, the structure is much simpler to use and adapt. We store all the attribute data in separate arrays. When traversing the SVO, the child index to find the voxel on the next level is also used to find its corresponding attributes.

4 | Algorithms

A brief overview of the voxelization and simplification algorithm is as follows:

- The first step is voxelizing the model to a sparse voxel octree. We use conservative voxelization techniques from [PK15]. We generate attributes for every voxel during voxelization.
- Based on user input regarding memory constraints and a prescribed error tolerance τ , we simplify the voxel model using the error metric.
- After the simplification, we optimize our datastructure by storing the leaf nodes separately.

We first explain the transformation from a triangle model to our voxel model, followed by the simplification algorithm. We end this chapter with optimizations for the simplified structure.

4.1 Voxelization

We made some changes to the attribute creation of [PK15] in order to create scalar fields per voxel. An overview of the voxelization algorithm is as follows:

- Pre-processing stage: we calculate angle weighted normals for every vertex, and store them locally on a per triangle basis.
- During voxelization: we combine duplicate voxels, while preserving correct scalar fields.
- Post-processing stage: to guarantee a C^0 -continuous surface, we correct scalar fields by comparing them with their neighbors.

4.1.1 Pre-processing

A method for generating discrete scalar fields from triangle meshes is detailed in [BA02]. They show angle-weighted normals for every edge and vertex of the triangle, which are

the weighted sum of normals of all incident faces. Without this additional normal, we are unable to generate the correct sign for the discrete distance field.

The voxelization from [PK15] generates voxel attributes on a per-triangle basis. In this stage we are unable to query neighboring triangles efficiently. We calculate the angle-weighted normal for every vertex in a pre-processing step, and store these on a per-triangle basis before the voxelization.

4.1.2 Attribute generation and attribute merging during voxelization

The voxelization algorithm generates multiple versions of the same voxel when multiple triangles overlap this voxel. In their reduction step they remove the duplicates by calculating the unique Morton indices. We have to merge these different versions in one unique voxel. For scalar fields this operation is rather simple: we compare all versions of the voxel corners and store the smallest distance.

After merging the voxels, we store the data for the SVO structure and the data for the attributes in separate arrays. This allows us to use different algorithms on both types of data. In this stage, we only create the SVO structure as described in the paper. After this stage, the internal SVO structure is complete. We now have the valid masks and indices [LK11] to the next level in the SVO available for efficient nearest-neighbor querying.

4.1.3 Post-processing

After the voxelization process, we can guarantee that the discrete signed distance fields are correctly stored per voxel, but we do not have a global C^0 -continuous distance field. There can still be differences between voxels, because the voxelization algorithm only processes all triangles contained inside a single voxel locally.

The post-processing step is applied to all attributes on the finest level. Specifically for scalar field attributes, we have to query all voxel neighbors. For every corner, we check overlapping corners from all neighbors. We store the smallest distance similar to the voxel merging.

After this post-processing step, we are guaranteed to have a global C^0 -continuous signed distance field representation.

4.2 Simplification

The simplification algorithms target the attributes of the voxels. To query the correct attributes, we use the existing SVO structure from previous sections to find the right voxel index.

The simplification algorithm works by repeatedly applying the same process on a coarser result. We recursively coarsen our geometry based on the outcome of the previous attribute level. We coarsen the level by minimizing the residual energy from Equation 3.7. The resulting hierarchy can be seen as a function that describes the difference between geometric details per level.

We build our attribute hierarchy bottom-up starting on the lowest level in the SVO structure, which are the parents of the finest voxel level. For every parent in this level, we query all child voxel attributes and merge them to form a voxel of the parent level. To find the new scalar field, we create the linear system described in 3.13.

We describe three simplification algorithms that operate purely on the signed distance fields in the following.

4.2.1 Corner simplification

The first algorithm is the most straightforward simplification that is still C^0 -continuous on all levels. The coarsening function simply keeps the same values on vertices that are shared between levels without any manipulation.

To find all new corner values, we need to fill in the missing voxels from the child level because our octree is sparse. For this we use a scalar field expansion algorithm detailed in appendix B.

4.2.2 Moving least squares simplification

This method uses Moving Least Squares (MLS) [Nea04] to create new values for every corner point in a higher level of the hierarchy. The prolongation P for this simplification is a quadratic polynomial.

The input for this simplification is a given point, and all child voxels that are within the windowing function (Section 3.3). We set the windowing function to 2 child voxels, to match the adaptive sampling method described in the next section.

The output is a quadratic polynomial that describes the function for our given point. We compute the distance value by using the coordinates of the given point, and store the result.

The weights for a point are computed using the Wendland function [Nea04].

4.2.3 Adaptive sampling

This is our method described in section 3.3. For the prolongation P , we use Bernstein polynomials of degree 1. This results in a constant windowing function of one parent

voxel, or two child voxels.

Similar to the MLS method, the inputs are a given point and the child voxels within the windowing function. The outputs are distances for all values of $F1$. We store the result for our given input point.

4.3 Offline LOD optimization

The goal of the LOD optimization is to achieve a model that is as sparse as possible without storing redundant information. We achieve this by storing the surface approximation as coarse as possible. To measure if we can remove a voxel, we use equation 3.6 to find the energy between two levels of detail. If this energy is smaller than the user-defined tolerance τ , we can safely remove all the child voxels.

This is a bottom-up algorithm starting at the parents of the finest level. To compare the parent voxel and child voxels, we prolong the parent voxel using trilinear interpolation. We compare the prolonged voxels to the original voxels and sum all errors together. Finally we divide by the size of the voxel to obtain a normalized comparison.

An additional check has to be applied when the algorithm is applied in any level higher than the finest level: we first check all the children of a voxel before comparing. If any of the child voxels still has children in the next level, we cannot simplify this voxel and continue with the next voxel.

4.3.1 LOD stitching

After we have applied the LOD optimization, we notice small gaps on edges between voxels of different levels in the hierarchy (see Figure 4.1). These gaps only appear at edges of simplified voxels and more detailed voxels. The detailed voxels store additional scalar data points that were removed in the simplified voxel. We apply a stitching algorithm to remove the gaps.

The LOD stitching is a top-down algorithm starting at the root level of the octree. For every leaf voxel marked by an empty valid mask (Section 2.2.2), we query neighbors that reside in a higher level in the hierarchy. This can be achieved by terminating the neighbor querying at leaf nodes, or when the depth is equal to the depth of the current voxel.

If we find a neighbor, we test which points of the current voxel touch the neighboring voxel. For every point that touches the neighboring voxel, we calculate a new distance for this point by trilinearly interpolating the corners of the neighbor voxel.

After the LOD stitching we are guaranteed to have no holes between different levels of detail for all leaf voxels. Both the LOD optimization and LOD stitching can be done

completely offline, and we do not have to change the structure while rendering.

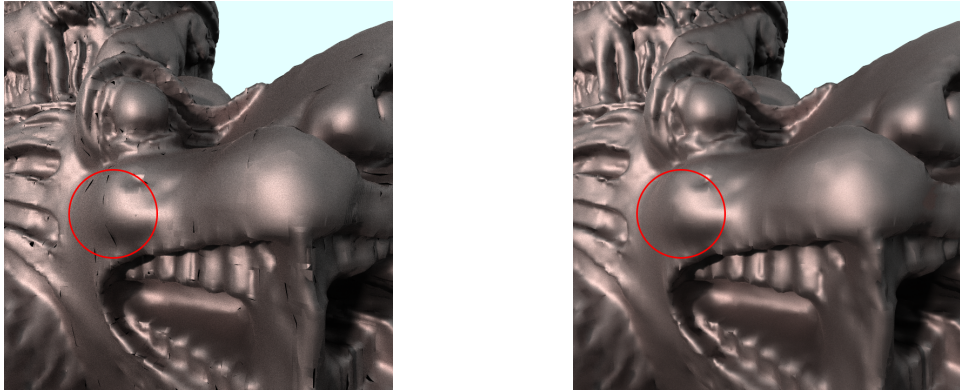


Figure 4.1: The difference between **left:** a model rendered without LOD stitching, and **right:** the same model with LOD stitching.

5 | Implementation

This chapter details the aspects of the implementation for reproducing the results. The framework was built using C++ with CUDA for GPU functionality. We used Intel MKL to solve a linear system of equations.

We follow the same structure as last chapter: first we look at the voxelization, followed by the simplification. We continue with surface extraction methods, and conclude with optimizations that we applied.

5.1 Voxelization

We changed the voxelization algorithm from [PK15] to a CPU version, in order to quickly separate different parts of the algorithm. Other than the described scalar field generation from the previous chapter, we made no significant changes to the algorithm.

5.2 Simplification

We describe the process to create the necessary matrices for the adaptive sampling in the next subsection. The next sections describe optimizations and problems we found while implementing the simplification methods.

5.2.1 Adaptive sampling

We calculate the P and M matrices locally from the neighboring voxels as dense matrices, as these will be small. We then combine them to obtain the P and M in a sparse format. When the matrices are complete, we transpose P to find P^T . Having the necessary components for Equation 3.13, we use a least-squares solver implemented in Intel MKL.

5.2.2 Solving duplicate voxel corners with least squares

We avoid avoid solving voxel corners multiple times by doing an encoding and decoding pass for the simplification methods using a least squares solver.

The encoding pass finds all unique points we have to solve. We create this unique list by converting all voxel corners to Morton codes. We sort the list, and remove all duplicates.

We can now solve all the unique corners. We find all the neighbors for every encoded point, and use the simplification algorithm of choice. Once we solve the linear system, we store the result in an array at the same index of the Morton code.

The decoding pass iterates over all voxels again. We find the Morton codes for all corners, and using a binary search in the Morton codes array we find the index for the corner. We can now lookup the result array to find the calculated distance value for this corner.

5.2.3 Solving sparse voxels

We often ran into problems when solving outer vertices of very sparse parent voxels. These vertices do not have enough neighboring child voxels to create a linear system of equations from. As the resulting values can both be positive (outside bounds) or negative (inside the model) we could not find a simple solution without querying more information.

We first tried solving the system using a singular value decomposition (also included as least squares solver in MKL), but this would often still result in incorrect values. We solved these cases using the scalar field expansion method described in [appendix B](#).

5.3 Surface extraction

To render the voxel models with distance fields, we use the optimized raytracing kernel from [\[LK10\]](#). We made two minor changes to the algorithm:

- We perform an additional intersection test for every leaf voxel we find in the hierarchy traversal. This intersection test is done using the raymarching algorithm described in the next section. If the ray does not intersect the distance field, we continue ray traversal.
- To render the voxel model to a specific level, we added a premature stop when the valid mask (Section [2.2.2](#)) is set to zero (no children). This allows us to ren-

der the model from various levels of detail dynamically. Again, if the premature intersection with the distance field fails, we continue traversal.

We explain our ray-isosurface intersection and normal calculation in the next two subsections.

5.3.1 Ray-isosurface intersection

We extract the surface information from a voxel by calculating the intersection point of a ray and an isosurface. A ray is defined as follows:

$$x(t) = O + tD$$

Where O is the ray origin and D the ray direction. An intersection with the surface is found as the first point where the distance between the ray and the zero level set is smaller than a small value ϵ . As a rule of thumb, we set ϵ to 1% of the smallest voxel size to accurately encompass the surface area. We implemented two methods to test ray-isosurface intersections.

The first is a straightforward iterative ray-marching algorithm to traverse the scalar field. The current distance to the surface is found using trilinear interpolation, and the ray progresses by adding this distance to the current position. The marching is terminated when either the distance is smaller than ϵ or $x(t)$ is no longer inside the voxel bounds.

The second algorithm is the analytical solution to the cubic polynomial that can be formed from rewriting the brute-force algebraic solution for t . The method is described in [PSL*98]. Solving a cubic polynomial is done using Cardano's formula. We find the closest root within the current voxel.

5.3.2 Normal approximation

We approximate the surface normal at the intersected point of a scalar field by using a finite difference method. The gradient can be approximated by:

$$\begin{aligned} N_x &= T(P_x + \epsilon) - T(P_x - \epsilon) \\ N_y &= T(P_y + \epsilon) - T(P_y - \epsilon) \\ N_z &= T(P_z + \epsilon) - T(P_z - \epsilon) \end{aligned}$$

Where N is the normal, and P the intersection point of the ray and scalar field. T is a trilinear interpolation to calculate the distance to the surface for a certain point. ϵ is the same as last section.

This approximation is continuous inside the voxel, but not on the edges. To create a smooth normal approximation we sample the finite differences from at most three neighboring voxels:

We change ϵ to a little less than half the voxel size. When any of the points for the finite difference calculation fall outside the voxel, we query the respective neighbor and sample the distance inside that voxel using trilinear interpolation. This way we avoid artifacts at edges of voxels. If the neighbor query does not return a result, we return the extrapolated distance from the current voxel.

5.4 Optimizations

Similarly to [AL09], we implemented a small change to the current traversal algorithm. They store the child node on intersection during their bounding volume hierarchy traversal. They start the intersection test only when enough threads in the current warp also have an intersection child node stored, or when one of the threads found a second intersection.

When we find an intersection, we store the current voxel intersection data. At the start of every traversal step, we check if enough threads have a stored intersection. If so, we proceed by doing an intersection test. Otherwise, we continue traversal with the complete warp. Occasionally there will be threads that find the next intersection test, while already having an intersection stored. In this case, the thread stops traversing until the next intersection test is performed.

5.4.1 Leaf array

In a full resolution model, we usually only render leaf voxels from the finest level of detail. This can be cached efficiently, since the array is stored in Morton order, meaning that neighboring voxels are stored close to each other in the array.

This is generally not the case in our LOD-optimized hierarchy, since most of the finest level voxels are reduced to other levels in the hierarchy. We tried to alleviate this problem by storing all the leaf voxels in a single array. We show the leaf array in Figure 5.1. We start with an initial copy of the finest level voxels, and then replace removed voxels with the simplified voxel in the place of the first finest voxel child. The removed finest level voxels are marked with a bit, and we compress the leaf array using a parallel reduction based on these bits. Finally, we store the new index to the leaf array in all the parents as the child index.

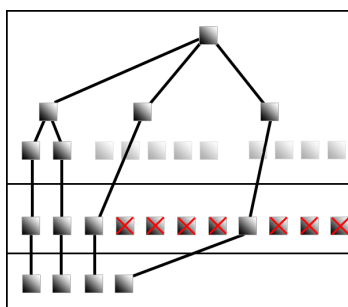


Figure 5.1: The process of creating the leaf array. First we make a copy of the finest level voxels, followed by removing voxels that were simplified. We copy the attribute information from voxels higher in the hierarchy to the first finest level child index in the leaf array, and unmark the voxel as deleted. Finally, we remove unused voxel data in the array to achieve the most sparse representation.

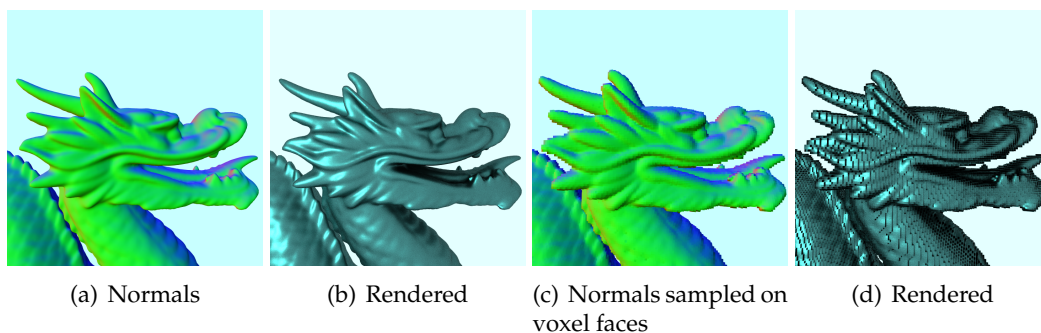


Figure 5.2: Comparison between different normal sampling functions. The normal functions are visualized by rescaling the unit normal to $[0, 1]$ range. We see that the normals evaluated on the faces of the voxels can still accurately describe the surface. When rendered, we see an overall darker result, because light rays are blocked by the voxels. This effect is removed when we only render voxels smaller than the projected pixel area.

5.4.2 Dynamic LOD

As mentioned in [LK10], a straightforward optimization is to stop traversing the octree when the projected area of the next intersected child voxel is smaller than the projected area of the pixel on the screen. In our structure we skip traversing a part of the octree. More importantly, we skip the ray-isosurface intersection test.

We implemented a version similar to the cone tracing detailed in section 2.2.4, approximating the radius of a cone around the ray. We also exploit the fact that the finite difference normal calculation can be evaluated at any point in the voxel.

A comparison between the normal sampling functions is shown in Figure 5.2. Note that the voxelized normals will only be rendered when the entire projected voxel area is smaller than the pixel, removing any aliasing effects seen in the figure.

6 | Results

To provide meaningful data for a discussion of our simplification algorithm we evaluate the resulting models in different settings. All results were computed with an Intel i7-4770K CPU and a Nvidia GTX 1070 GPU. All visual results are generated with our path tracer, using a GGX BRDF [WMLT07].

The models are resized to fit in a cube of 100^3 . The longest axis of the model's bounding box is used to determine the scaling for the model to fit in the cube. The voxelization is then done by recursively splitting every cube into eight smaller voxels, which are exactly half the size of the original cube. For example, a model voxelized to resolution 2^5 will have a grid size of 3.125.

6.1 Datasets

We used highly detailed triangle meshes, to create meaningful comparisons between the triangle mesh and our generated results. All the datasets we used can be seen in Table 6.1.

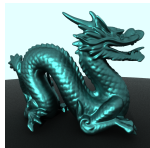
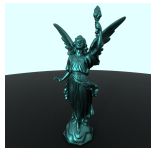
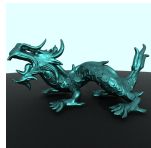
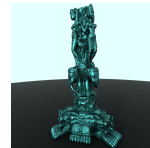
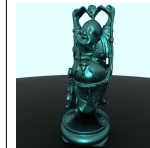

						
Name	Dragon	Lucy	XYZRGB Dragon	XYZRGB Statuette	Happy Buddha	Armadillo
Triangles	871,306	28,055,742	7,218,906	10,000,000	1,087,474	345,944
Size (MB)	17	520	133	220	55	6

Table 6.1: Depictions of all the triangle mesh datasets, rendered in their voxelized form using our path tracer.

6.2 Root-mean-square error

To measure the error of approximation, we sample values from the SVO generated by the simplification and optimization algorithms in Chapter 4. To compare the distance values, we take the input triangle mesh as ground truth. We generate uniform random samples on the surface of the triangle mesh, and query the distance of that location in the SVO. We measure the average over 1,000,000 samples.

We first measured the root-mean-square error (RMSE) for unprocessed models. The results are shown in Figure 6.1. We then converted all the datasets and applied both post-processing steps from section 4.3: LOD optimization and LOD stitching. The results are shown in Figure 6.2.

The unprocessed results show a logarithmic trend between the RMSE and the converted resolution. Figure 6.2 shows that the increase in RMSE is dependent on the size of the user-defined tolerance τ (see section 4.3). In Figure 6.2 we also see that the simplification starts at higher grid sizes, and is more aggressive with a higher tolerance.

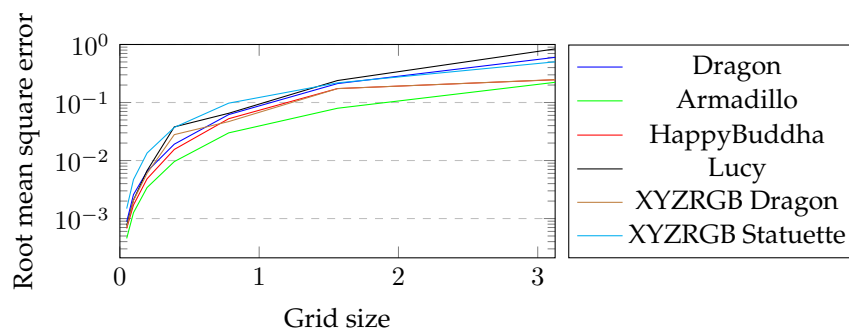


Figure 6.1: The root mean square error measured for unprocessed voxel models. There is a clear logarithmic trend between the measured RMSE and the grid size.

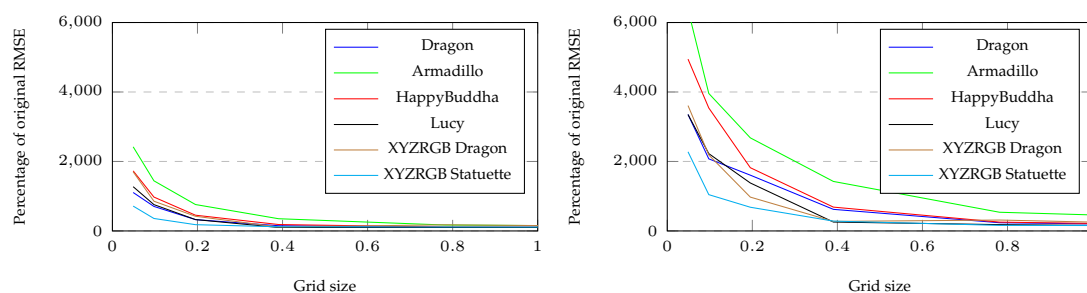


Figure 6.2: Left: $\tau = 10^{-2}$, Right: $\tau = 10^{-1}$. The percentage of RMSE compared to the original RMSE from Figure 6.1. The models are voxelized and optimized using LOD optimization and LOD stitching. We see that the RMSE of the models increases significantly when the original RMSE in Figure 6.1 is below the given tolerance.

6.3 Memory compression

We compare the memory usage of the converted voxel models to the size of the complete SVO structure. In Table 6.2 we show the total memory usage for models with and without LOD optimizations. Figure 6.3 shows the compression percentage mapped to the grid size.

We can clearly see the difference in complexity of simplification at lower resolutions. The higher resolutions are hard to separate in terms of compression rate, since this results in only small relative differences.

Name	Type	2^7		2^8		2^9		2^{10}		2^{11}		
Dragon	Full size	2.22		9.51		38.98		156.72		629.80		
	$\tau = 10^{-2}$	%	1.09	50.94 %	2.00	79.01 %	3.31	91.52 %	7.42	95.26 %	26.48	95.8 %
	$\tau = 10^{-1}$	%	0.37	84.23 %	0.65	93.27 %	1.60	95.88 %	5.57	96.44 %	23.07	96.34 %
Armadillo	Full size	2.25		9.33		37.73		152.04		610.26		
	$\tau = 10^{-2}$	%	1.35	40.21 %	2.42	74.03 %	3.60	90.46 %	8.35	94.51 %	27.74	95.45 %
	$\tau = 10^{-1}$	%	0.25	88.83 %	0.59	93.63 %	1.70	95.5 %	6.47	95.74 %	24.69	95.95 %
HappyBuddha	Full size	1.77		7.42		30.14		121.75		490.32		
	$\tau = 10^{-2}$	%	1.24	29.79 %	2.74	63.02 %	4.13	86.3 %	7.79	93.6 %	24.71	94.96 %
	$\tau = 10^{-1}$	%	0.40	77.63 %	0.73	90.12 %	1.61	94.69 %	5.16	95.76 %	20.88	95.74 %
Lucy	Full size	1.04		4.45		18.10		89.26		360.14		
	$\tau = 10^{-2}$	%	0.85	18 %	2.24	49.76 %	3.90	78.47 %	7.50	91.59 %	17.68	95.09 %
	$\tau = 10^{-1}$	%	0.32	65.94 %	0.57	85.74 %	1.16	93.32 %	3.55	96.02 %	13.41	96.28 %
XYZRGB Dragon	Full size	1.11		5.26		22.04		89.73		361.94		
	$\tau = 10^{-2}$	%	0.84	24.88 %	2.49	52.67 %	4.87	77.88 %	9.30	89.63 %	22.64	93.75 %
	$\tau = 10^{-1}$	%	0.30	75.21 %	0.55	89.41 %	1.36	93.81 %	4.65	94.82 %	17.82	95.08 %
Statuette	Full size	1.71		7.63		31.63		129.32		523.54		
	$\tau = 10^{-2}$	%	1.47	14.32 %	4.50	40.98 %	9.76	69.13 %	18.49	85.7 %	40.35	92.29 %
	$\tau = 10^{-1}$	%	0.49	72.14 %	1.02	86.6 %	2.32	92.67 %	7.01	94.58 %	26.61	94.92 %

Table 6.2: The memory usage in megabytes (MB). Full size is the size of the complete SVO structure including attributes. The next rows show the size and compression percentage of the LOD-optimized models with tolerance τ set to 10^{-2} and 10^{-1} respectively. All simplifications were done using our adaptive sampling method. We see a significant increase in compression rate when the tolerance is set to 10^{-1} compared to 10^{-2} .

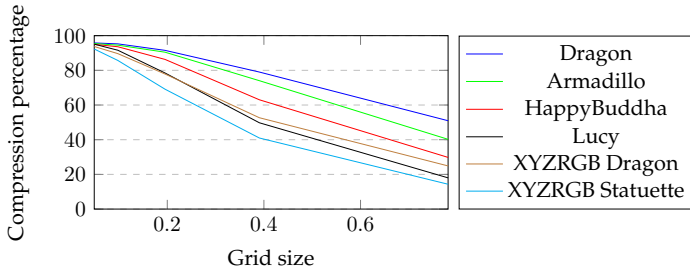


Figure 6.3: Compression percentage for all datasets compared to the grid size, using the results from Table 6.2 generated with our adaptive simplification. We see that the relative compression percentages are very similar at high resolutions, but diverge when the resolution is lowered.

6.4 Frames per second

To measure the rendering performance of our structure, we set up two static camera position benchmarks:

- Detail: a camera view that is rendering a part of the model up close, showing the details.
- Fit: a view that renders the complete octree such that it perfectly fits on the screen.

Other than the camera position, we also compared different resolutions and different ray intersection methods. We compare a relatively small resolution (2^7) and a larger resolution (2^{10}). The frames per second (FPS) are averaged over a time span of 10 seconds. We render the smooth normals of the model, using only primary rays.

Analytic ray intersection								
	Original SVO				LOD-optimized			
	Detail		Fit		Detail		Fit	
	2^7	2^{10}	2^7	2^{10}	2^7	2^{10}	2^7	2^{10}
Dragon	135.10	102.69	220.60	173.19	138.19	113.69	219.10	182.80
Armadillo	104.40	77.19	176.19	133.69	98.09	84.09	165.50	142.10
HappyBuddha	172.80	131.60	254.00	200.50	170.39	144.69	250.00	215.10
Lucy	201.50	161.39	302.70	256.79	204.00	169.89	298.20	250.39
XYZRGB Dragon	166.10	124.80	252.89	207.89	154.80	131.50	240.30	205.00
XYZRGB Statuette	194.10	148.69	238.60	190.69	184.10	150.89	229.00	185.69
Iterative ray intersection								
	Original SVO				LOD-optimized			
	Detail		Fit		Detail		Fit	
	2^7	2^{10}	2^7	2^{10}	2^7	2^{10}	2^7	2^{10}
Dragon	105.80	83.30	173.00	83.30	95.00	81.00	156.89	137.50
Armadillo	81.10	58.90	138.30	58.90	65.09	58.40	115.19	104.09
HappyBuddha	138.90	108.40	210.39	108.40	124.19	109.40	190.89	169.50
Lucy	181.30	131.10	250.80	218.69	148.50	124.40	246.19	209.19
XYZRGB Dragon	130.90	101.19	211.19	101.19	108.69	94.69	182.10	160.60
XYZRGB Statuette	158.90	120.0	196.89	120.00	134.89	109.90	173.89	145.80

Table 6.3: The measured FPS for the for primary rays at 720p. Higher is better. We compare the different ray intersection methods from section 5.3.1. For both methods, we compare measured FPS between an unprocessed model versus the LOD-optimized in different resolutions. Tolerance $\tau = 10^{-2}$. Detail and Fit camera positions are described in section 6.4. The LOD-optimized version does not show significant improvements nor decrease in performance compared to the original SVO using analytic ray intersections. The iterative traversal shows a decrease in performance when using the LOD-optimized version.

6.4.1 LOD-optimized and unprocessed models

We measured the FPS for unprocessed models, and for models with LOD optimization and LOD stitching. The results can be seen in Table 6.3.

From these results we can see multiple correlations. The most obvious trend is that the higher the resolution of the model, the lower the FPS. We see lower FPS on the details camera position compared to the fit camera position. This can be explained by the fact that more ray casts are terminated early if the model occupies less screen space.

Comparing between both intersection algorithms, we can see a clear difference: the iterative traversal algorithm shows lower FPS on both compressed and uncompressed datasets. The iterative traversal also shows decreased performance when rendering LOD-optimized versions compared to unprocessed models. These results show that the iterative ray traversal method is not voxel-size neutral, meaning that the method uses more iterations for larger voxels.

Comparing the LOD-optimized results with the unprocessed models, we see a slight improvement on the high resolution detail view when using LOD-optimized models. Other than that we can see no clear trend, the results differ per dataset.

6.4.2 Leaf array

We ran the same test for our LOD-optimized structure using the leaf array (see section 5.4.1). The results can be found in Table 6.4. There are no significant changes in performance compared to standard SVO structure.

Analytic ray intersection using leaf array				
	Detail		Fit	
	2^7	2^{10}	2^7	2^{10}
Dragon	98.78 %	100.26 %	100.97 %	100.00 %
Armadillo	97.22 %	101.56 %	99.29 %	99.59 %
HappyBuddha	99.53 %	100.41 %	100.16 %	100.51 %
Lucy	101.05 %	104.81 %	101.33 %	99.21 %
XYZRGB Dragon	99.62 %	104.29 %	99.09 %	99.66 %
XYZRG Statuette	99.35 %	104.50 %	100.62 %	100.81 %

Table 6.4: FPS measured for models with leaf array optimization (see Section 5.4.1). Shown are the percentages relative to the values shown in Table 6.3. Tolerance and camera positions are identical to Table 6.3. We see a slight improvement in tracing performance on high resolution detailed views of the model.

6.4.3 Dynamic LOD

We ran the FPS test similar to previous section with dynamic LOD enabled (see Section 5.4.2). We used simplified models generated by our adaptive method. Results can be found in Table 6.5. We see an increase in FPS on distant camera views compared to the results without dynamic LOD. This is due to the fact that we skip a lot of the traversal and intersection tests.

Analytic ray intersection						
Resolution	Dragon	Armadillo	HappyBuddha	Lucy	XYZRGB Dragon	XYZRGB Statuette
2^{10}	112.58 %	106.75 %	113.81 %	118.30 %	115.12 %	132.37 %

Table 6.5: FPS measured for models with dynamic LOD enabled (see section 5.4.2). Shown are the percentages relative to the values shown in Table 6.3. Tolerance $\tau = 10^{-2}$. The camera was set to Fit. We see improvements in FPS for all datasets.

6.5 Cross comparisons

We made several cross comparisons between different datasets. We already saw the resolution mapped versus the compression in Figure 6.3.

We compared the different simplification methods to each other. We picked the dragon dataset and ran the simplification algorithm for all three methods on different resolutions. We compared the compression percentage versus the resolution in Figure 6.4. We also compared the RMSE versus different resolutions in Figure 6.5. We combined both results to compare the RMSE versus the compression in Figure 6.6.

Figure 6.4 and Figure 6.5 both show that our simplification method is superior in finding simplifications on lower resolutions, and representing with a lower RMSE. Figure 6.6 shows that our adaptive method can both reach higher compression rates while maintaining a lower RMSE.

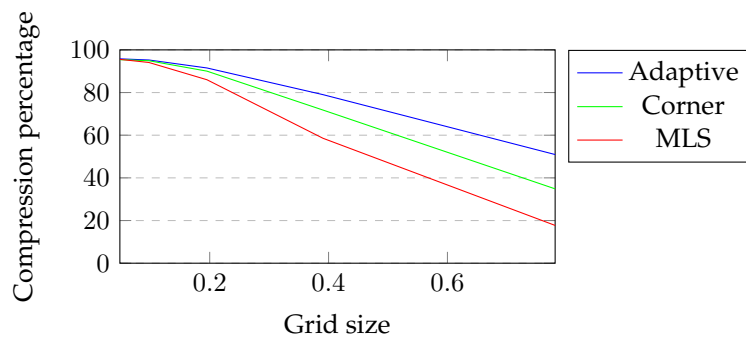


Figure 6.4: Compression compared to grid size for different simplification methods. Higher percentage is better. All results were generated for the dragon dataset at different resolutions. We see that our adaptive sampling has a higher compression rate at all resolutions.

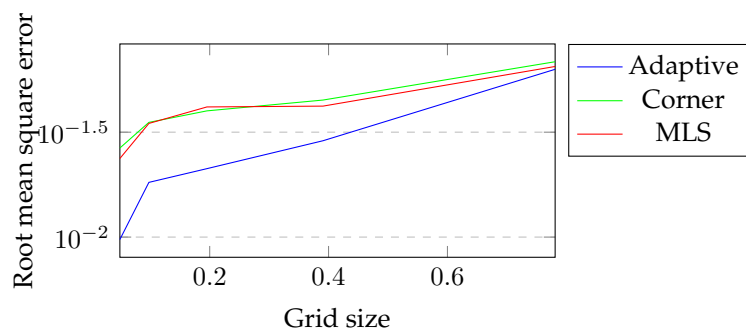


Figure 6.5: RMSE compared to the grid size. Smaller error is better. All results were generated for the dragon dataset at different initial grid sizes. The adaptive sampling method reduces the RMSE at smaller initial grid sizes.

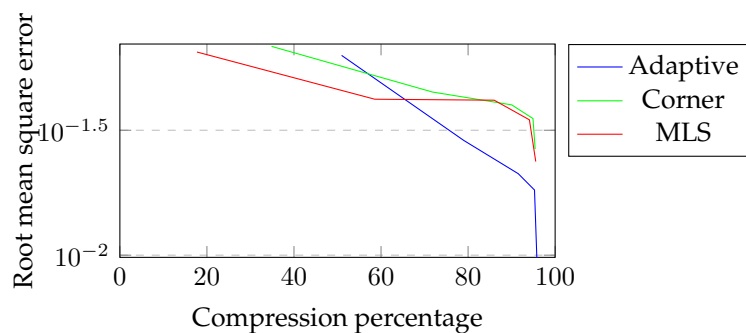


Figure 6.6: RMSE compared to the compression percentage. These are results for the dragon dataset combined from Figure 6.5 and 6.4. We see that our adaptive method reaches lower RMSE at similar compression rates.

6.6 Catmull-Clark subdivision rules

In our adaptive sampling, the prolongation weight for a corner which is not simplified is always identity. We tried to change these weights using the subdivision weights from the Catmull-Clark [CC78] subdivision:

$$q = \frac{F}{n} + \frac{2E}{n} + \frac{P(n-3)}{n} \quad (6.1)$$

Where q is a new calculated vertex based on: n the number of vertices, F : the average of all adjacent face points, E : the average of all adjacent edge points, and P : the original point.

Our new prolongation operator changed the identity corner weight to the weights above, based on the vertices of a voxel: $n = 8$. The corner of a voxel always has 3 edges, and thus 3 edge points E . We remove the point diagonal of the original point, as this is inside the volume, and is neither a direct edge or face point. The remaining three points are face points.

We ran the same tests to measure memory compression and the RMSE of this prolongation operator. While the RMSE showed similar results to our adaptive simplification, the memory compression rate seemed to be worse.

We think this result can be explained by examining the function we are trying to simplify: a combination of multilinear cells. The initial function is merely trilinear, and the Catmull-Clark subdivision approximates B-spline functions of higher order by design.

6.7 Error quadrics in triangle meshes

To compare our voxel simplification to an existing triangle mesh simplification, we chose to measure our results in a similar scenario for the error quadrics [GH97]. Using the error quadric simplification available in Meshlab, we simplified the dragon model.

We simplified the triangle mesh to the compression percentage found using our adaptive sampling method. The actual percentages can be found in Figure 6.7. We compared the simplified triangle mesh to the original by using our RMSE sampling method described in section 6.2.

Comparing the visuals shown in Figure 6.7, we see that our adaptive sampling method smooths details in a more natural way by design compared to triangular mesh quadrics. The quadrics suffer from triangle sampling problems at lower rates. Our adaptive simplification with tolerance $\tau = 10^{-1}$ shows small bumps or holes at the edges of

larger voxels, while the triangle mesh shows more irregular artifacts overall because the edges and vertices are clearly visible at lower resolutions. We also see the RMSE for both methods in Table 6.6, which shows that the RMSE for both methods are fairly similar.

Tolerance	10^{-1}	10^{-2}	10^{-3}
Quadrics	0.020231	0.017789	0.009564
Adaptive optimization	0.038207	0.015811	0.007605

Table 6.6: RMSE measured for quadric triangle decimation versus our adaptive LOD simplification. Lower is better. The results are generated by comparing the simplified dragon mesh with the original. We see that our simplification is similar to the triangle decimation for lower tolerance settings, but performs worse at 10^{-1} .

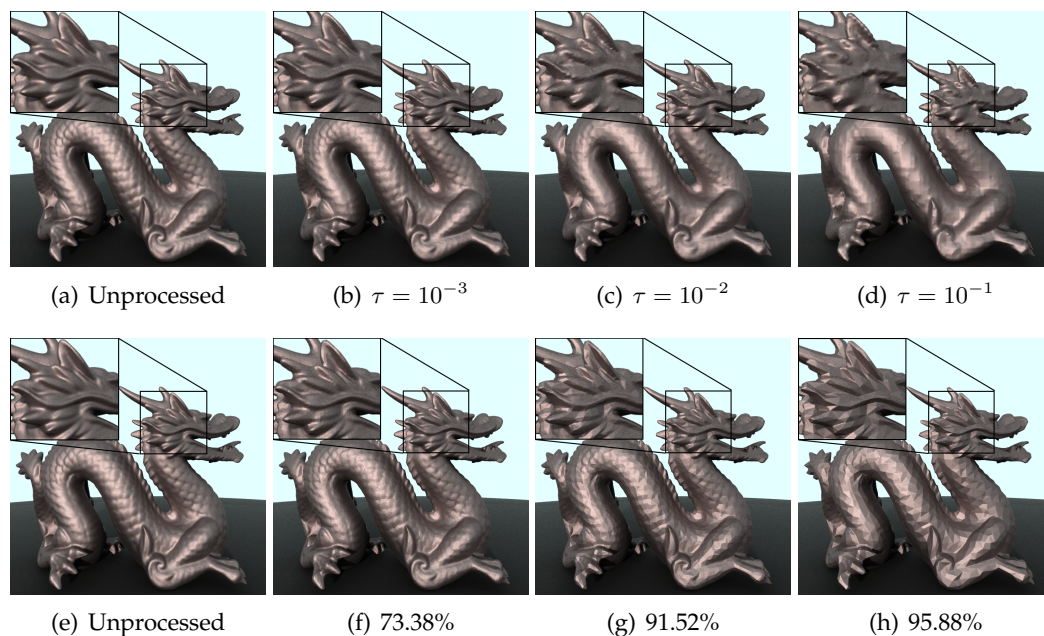


Figure 6.7: Visual comparison for different LOD optimization settings. **Top:** Voxel model simplified using our adaptive simplification. **Bottom:** Triangle model simplified using error quadric simplification. The triangle models were generated using quadric simplification to the same compression percentage found using our simplification. While the contours are preserved very accurate in both methods, the specular highlights show the rapid decrease of the approximated normal function. The triangle mesh shows more irregular artifacts.

7 | Discussion

7.1 Conclusion

Our new implicit simplification method can combine surface data from multiple levels of detail with more accuracy and compression than the other presented solutions. Results show that the simplification can couple the energy per voxel to a user-defined tolerance, and reach a simplification that can still accurately visualize the contour of the model. We can render these compressed models without any loss in ray traversal speed. We gain some improvements on traversal speed by using the dynamic LOD and leaf array optimizations.

We have shown comparisons with error quadrics that reduce triangle meshes, and achieved similar results for simplifying voxel meshes using our error metric.

7.2 Future work

7.2.1 Prolongations

Using other functions for the prolongation as a trilinear interpolation could prove to be useful. If we use a function that extends the window size to include more than direct neighbors, we can remove the LOD stitching. Smoothing will be done implicitly by the function.

7.2.2 In-place optimized LOD generation

We found that it is possible to generate the complete simplification in place during the voxelization steps. We specifically separated the simplifications for the SVO structure and the attributes, but this is not necessary.

[PK15] divides the triangle mesh in batches for processing. They do this by marking the end of the batch with a Morton code. Any voxels that have a higher Morton code

will not be processed to generate the SVO structure in this batch.

If we want to create a signed distance field, we need to be able to sample all neighbors for every processed voxel. If we can calculate a new end of the batch that allows every voxel with a smaller Morton code to also query all neighbors, we can create the complete structure with attributes in-place.

We calculate the Morton code that is exactly $(x - 1, y - 1, z - 1)$ from the current maximum. This allows for the neighbor lookups. The neighbor searching can be implemented by storing the Morton code in every voxel, and using a binary search to find the respective neighbors.

7.2.3 Dynamic octree depth

It might be interesting to see if we can avoid the initial guess for the depth. Using our error metric, we can determine the approximation error after the initial voxelization. If the error is large enough, we can subdivide the voxel and resample the input model to generate child voxels. Using this recursively until we meet the error requirement could remove the initial guess.

7.2.4 Additional compression

Directed Acyclic Graphs (DAG) for SVOs are presented in [KSA13]. Where the SVO stores indices to the next level to find child nodes, the DAG only stores all unique paths. We can apply the same optimization, if we use the decoupling scheme presented in [Dad15] to store the attributes.

Bibliography

- [AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on gpus. In *Proceedings of the conference on high performance graphics 2009* (2009), ACM, pp. 145–149. [31](#)
- [BA02] BÆRENTZEN J. A., AANÆS H.: Generating signed distance fields from triangle meshes. *Informatics and Mathematical Modeling, Technical University of Denmark, DTU 20* (2002). [23](#)
- [Ber04] BERTRAM M.: Biorthogonal loop-subdivision wavelets. *Computing* 72, 1-2 (2004), 29–39. [15](#)
- [CC78] CATMULL E., CLARK J.: Recursively generated b-spline surfaces on arbitrary topological meshes. *Computer-aided design* 10, 6 (1978), 350–355. [40](#)
- [CNLE09] CRASSIN C., NEYRET F., LEFEBVRE S., EISEMANN E.: Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games* (2009), ACM, pp. 15–22. [7](#)
- [CNS*11] CRASSIN C., NEYRET F., SAINZ M., GREEN S., EISEMANN E.: Interactive indirect illumination using voxel cone tracing. In *Computer Graphics Forum* (2011), vol. 30, Wiley Online Library, pp. 1921–1930. [14](#)
- [Dad15] DADO B.: *Geometry and Attribute Compression for Voxel Scenes*. PhD thesis, TU Delft, Delft University of Technology, 2015. [43](#)
- [DW15] DEMIR I., WESTERMANN R.: Vector-to-Closest-Point Octree for Surface Ray-Casting. In *Vision, Modeling & Visualization* (2015), Bommers D., Ritschel T., Schultz T., (Eds.), The Eurographics Association. [13](#)
- [FBH*10] FATAHALIAN K., BOULOS S., HEGARTY J., AKELEY K., MARK W. R., MORETON H., HANRAHAN P.: Reducing shading on gpus using quad-fragment merging. In *ACM Transactions on Graphics (TOG)* (2010), vol. 29, 4, ACM, p. 67. [7](#)
- [GH97] GARLAND M., HECKBERT P. S.: Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and*

- interactive techniques* (1997), ACM Press/Addison-Wesley Publishing Co., pp. 209–216. [40](#)
- [GM05] GOBBETTI E., MARTON F.: Far voxels: a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. In *ACM Transactions on Graphics (TOG)* (2005), vol. 24, 3, ACM, pp. 878–885. [7](#)
- [IKLH04] IKITS M., KNISS J., LEFOHN A., HANSEN C.: Volume rendering techniques. In *GPU Gems* (2004). [6](#)
- [JBS06] JONES M. W., BÆRENTZEN J. A., SRAMEK M.: 3d distance fields: A survey of techniques and applications. *Visualization and Computer Graphics, IEEE Transactions on* 12, 4 (2006), 581–599. [6](#)
- [JLSW02] JU T., LOSASSO F., SCHAEFER S., WARREN J.: Dual contouring of hermite data. *ACM Transactions on Graphics (TOG)* 21, 3 (2002), 339–346. [11](#)
- [KSA13] KÄMPE V., SINTORN E., ASSARSSON U.: High resolution sparse voxel dags. *ACM Transactions on Graphics (TOG)* 32, 4 (2013), 101. [43](#)
- [LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3d surface construction algorithm. In *ACM siggraph computer graphics* (1987), vol. 21, 4, ACM, pp. 163–169. [11](#)
- [LK10] LAINE S., KARRAS T.: Efficient sparse voxel octrees—analysis, extensions, and implementation. *Tech. Rep. NVR-2010 001* (2010). [7](#), [12](#), [22](#), [29](#), [32](#)
- [LK11] LAINE S., KARRAS T.: Efficient sparse voxel octrees. *Visualization and Computer Graphics, IEEE Transactions on* 17, 8 (2011), 1048–1059. [12](#), [24](#)
- [Lue01] LUEBKE D. P.: A developer’s survey of polygonal simplification algorithms. *Computer Graphics and Applications, IEEE* 21, 3 (2001), 24–35. [10](#)
- [Mor66] MORTON G. M.: *A computer oriented geodetic data base and a new technique in file sequencing*. IBM Ltd, 1966. [19](#)
- [MS11] MANSON J., SCHAEFER S.: Wavelet rasterization. In *Computer Graphics Forum* (2011), vol. 30, 2, Wiley Online Library, pp. 395–404. [16](#)
- [Nea04] NEALEN A.: An as-short-as-possible introduction to the least squares, weighted least squares and moving least squares methods for scattered data approximation and interpolation. URL: <http://www.nealen.com/projects/130> (2004), 150. [25](#)
- [PK15] PÄTZOLD M., KOLB A.: Grid-free out-of-core voxelization to sparse voxel octrees on gpu. In *Proceedings of the 7th Conference on High-Performance Graphics* (2015), ACM, pp. 95–103. [23](#), [24](#), [28](#), [42](#)

- [PSL*98] PARKER S., SHIRLEY P., LIVNAT Y., HANSEN C., SLOAN P.-P.: Interactive ray tracing for isosurface rendering. In *Visualization'98. Proceedings* (1998), IEEE, pp. 233–238. [30](#)
- [RCBW12] REICHL F., CHAJDAS M. G., BÜRGER K., WESTERMANN R.: Hybrid Sample-based Surface Rendering. In *VMV 2012* (2012), The Eurographics Association, pp. 47–54. [7](#)
- [WL16] WEISS K., LINDSTROM P.: Adaptive multilinear tensor product wavelets. *Visualization and Computer Graphics, IEEE Transactions on* 22, 1 (2016), 985–994. [17](#)
- [WMLT07] WALTER B., MARSCHNER S. R., LI H., TORRANCE K. E.: Microfacet models for refraction through rough surfaces. In *Proceedings of the 18th Eurographics conference on Rendering Techniques* (2007), Eurographics Association, pp. 195–206. [33](#)

A | The M matrix in Mathematica

(*First degree Bernstein*)

$\lambda_0[p_x_, p_y_, p_z_] := (1 - p_x)(1 - p_y)(1 - p_z)$

$\lambda_1[p_x_, p_y_, p_z_] := p_x(1 - p_y)(1 - p_z)$

$\lambda_2[p_x_, p_y_, p_z_] := (1 - p_x)p_y(1 - p_z)$

$\lambda_3[p_x_, p_y_, p_z_] := p_x p_y(1 - p_z)$

$\lambda_4[p_x_, p_y_, p_z_] := (1 - p_x)(1 - p_y)p_z$

$\lambda_5[p_x_, p_y_, p_z_] := p_x(1 - p_y)p_z$

$\lambda_6[p_x_, p_y_, p_z_] := (1 - p_x)p_y p_z$

$\lambda_7[p_x_, p_y_, p_z_] := p_x p_y p_z$

$m = \{\{\lambda_0[p_x, p_y, p_z], \lambda_1[p_x, p_y, p_z], \lambda_2[p_x, p_y, p_z], \lambda_3[p_x, p_y, p_z],$

$\lambda_4[p_x, p_y, p_z], \lambda_5[p_x, p_y, p_z], \lambda_6[p_x, p_y, p_z], \lambda_7[p_x, p_y, p_z]\}\}$;

$mt = \text{Transpose}[m]$;

$mtm = mt.m$;

$M = \text{Map}[\text{Integrate}[\#\#, \{p_x, 0, 1\}, \{p_y, 0, 1\}, \{p_z, 0, 1\}] \&, mtm, \{2\}] // \text{MatrixForm}$

B | Expanding scalar fields

In order to keep this algorithm relatively simple and fast, we use a breadth-first algorithm that works on a grid of values. A brief overview of the algorithm:

- Mark all completed points
- Use a breadth-first search to find the direct neighbors
- For every neighbor, find the closest distance to the surface
- Repeat until all points are completed

B.1 Completed points and direct neighbors

We first mark all grid points that contain distance values as completed and add them to a list. Using a breadth-first search we collect all points that are direct neighbors of the completed grid values. A direct neighbor is defined as a point that is axis-aligned with a completed point, which ensures that the distance between them is always equal to the size of the voxel edge.

B.2 Smallest distance to the surface

B.2.1 Four voxel faces

We define the normal as $n = |n_p - c_p|$ where n_p is the location of the neighbor, and c_p the position of the completed point. Using this normal, there are four possible voxel faces connected to the completed point, which can be found by using the two orthogonal directions aligned to the grid.

B.2.2 Smallest distance

When we find a face that contains only completed values, we can calculate the smallest distance to the surface. The distance is composed of two values: d_1 : the distance to the voxel face itself, and d_2 : the signed distance field inside the voxel.

d_1 can be calculated using trigonometry, the distance to one of the points on the voxel face is either the size of a voxel edge, the diagonal of a voxel face, or the diagonal of the voxel cube.

To calculate complete distance, we have to compare the distance of the four corners of the voxel face. Since it is a bounded linear equation, the minimum has to be one of the corners.

We first calculate the distance d to the surface over the axis-aligned edge. We then calculate the difference in distance for both d_1 and d_2 for all four points of the voxel face, by subtracting d from every value.

We first assume the smallest distance is over the axis-aligned edge to the completed point. We calculate the smallest distance by calculating $d_2 - d_1$ for all corners. We pick the corner that has the smallest distance which is also ≤ 0 .

B.2.3 Correctness

The expanded values are correct only under the following assumptions. First, the distance to the surface inside a completed voxel should match the voxel size, otherwise the voxel size cannot be compared to the distance values.

To guarantee global correctness of the expanded values, we have to make sure that for any expanded point, there is no other surface closer to this point. We define a square area with an edge length x in terms of voxel-edges ($x = 2$ means that the area is 2 by 2 voxels). We have to sample at least as many neighboring voxels x in every axis around the area in which we want correct values, to ensure that neighboring distances are calculated correctly with our breadth-first expansion.

The algorithm can be terminated when all values inside the area are found. The maximum iterations can be no longer than the manhattan distance in terms of x .