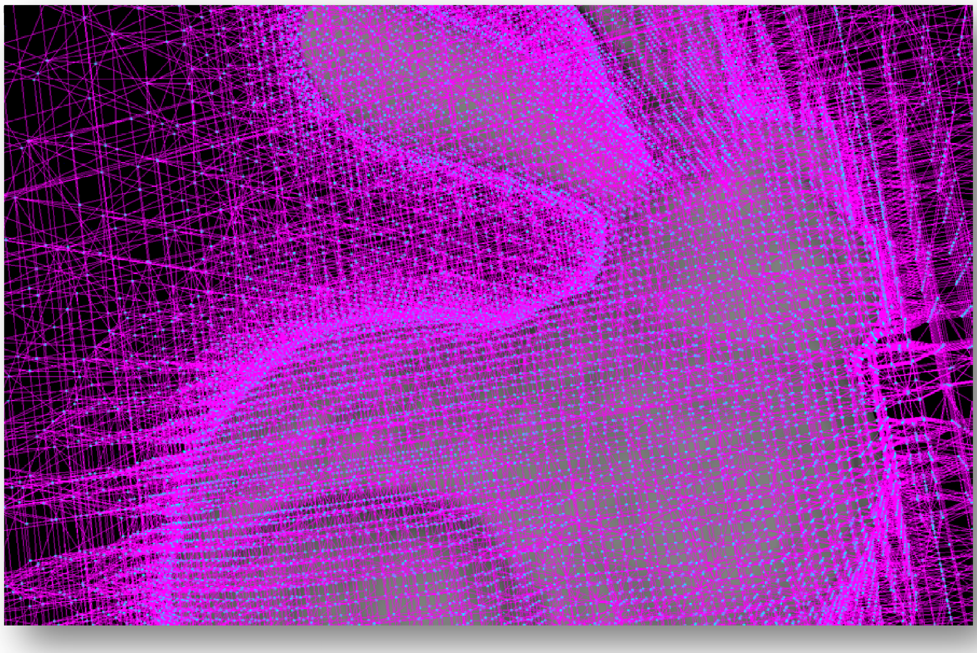


Manifold Contouring of an Adaptively Sampled Distance Field

Master of Science Thesis in the Programme Interaction Design

ELIAS HOLMLID

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, June 2010

Manifold Contouring of an Adaptively Sampled Distance Field

Elias Holmlid

Examiner: Ulf Assarsson

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover:
A portion of the Stanford Bunny contoured from the dual grid (magenta) extracted from an octree. Note that the grid is much denser around the contour of the surface due to an adaptive distance field being used.

Department of Computer Science and Engineering
Göteborg, Sweden June 2010

Abstract

Volumetric data is a convenient representation of shape on many occasions. One application area is remeshing, where a poorly triangulated model is converted to a volumetric representation and then transformed back into a model of better triangle quality. In certain areas, for example medical scans, volumetric data arise naturally. To render the shapes captured by such a scan, a common approach is to convert the volumetric data into a triangle mesh. Since both types of representations are valuable, it is interesting to find reliable and efficient ways of converting between them. Here, we will exclusively look at the conversion from volume data to triangle mesh. Many methods exist for performing such an operation, where one of the most popular is Marching Cubes. The meshes resulting from this algorithm will however have properties often undesirable. Furthermore, the method is not applicable when storing the volumetric data in an adaptive structure, such as an octree. In this report, we will describe how meshes without these undesirable properties can be generated from an octree in a straightforward manner.

Acknowledgements

I would like to thank my supervisors; Henrik Rydberg (Enmesh AB) and Ulf Assarsson (Chalmers Institute of Technology). Also thanks to Stefan Preijert at Volvo for letting me use the Knuckle model.

Contents

1 Introduction **6**
1.1 Background . . . 6
1.1.1 Finite Element Analysis . . . 6
1.1.2 Incentives . . . 7
1.2 Purpose . . . 7
1.3 Method . . . 7
1.4 Materials . . . 8
1.5 Scope and Contributions . . . 8
1.6 Report Structure . . . 8

2 Fundamental Concepts **9**
2.1 Implicit Surfaces . . . 9
2.2 Spatial Hierarchies . . . 9
2.3 Gradient . . . 10
2.4 Manifoldness . . . 11

3 Previous work **12**
3.1 Marching Cubes . . . 12
3.1.1 Algorithm Description . . . 13
3.1.2 Face Ambiguity . . . 14
3.2 Adaptive Distance Fields . . . 16
3.3 Isosurface Extraction on Octrees . . . 16
3.3.1 Dual Marching Cubes . . . 16

4 Implementation **21**
4.0.2 Creating the Dual Vertices . . . 21
4.0.3 Constructing the Dual Grid . . . 21
4.0.4 Applying Marching Cubes to the Dual Grid . . . 26

5 Results **27**

6 Conclusions **28**

7 Discussion **30**

8 Future Work **31**

9 Appendix A: Tools Developed **36**
9.1 MC Table Viewer . . . 36
9.2 ContourViewer . . . 36

1 Introduction

1.1 Background

Geometrical shapes can be stored and processed in a computer in several different ways. One common representation is to approximate the contour of a shape using polygons. For example, the vast majority of 3D console and computer games use models where triangles are the basic building block. A reason for this is that modern graphics cards are heavily optimized toward triangle rendering.

Another way of representing shape is to define it volumetrically. In this case we have a field where each point contains the shortest distance from that point to the surface. An advantage with this representation is that the underlying surface easily can be modified, through operations like Constructive Solid Geometry (CSG) [10].

Since both types of representations are useful, finding reliable and efficient ways of converting between them is an active field of study. In this thesis, we will in particular look at the conversion from volumetric model to triangle mesh. This procedure is commonly referred to as *contouring* [30]. Depending on the application domain, preferred properties and requirements imposed on the conversion results might vary [31]. If the meshes are to be used in real-time rendering for instance, the focus might be on finding an optimal trade-off between visual aspects and performance.

In the field of mechanical engineering, mesh quality is more important, since higher quality meshes reduce numerical errors [27]. The work presented here was undertaken at Enmesh AB, a simulation company based in Gothenburg, specializing in easy-to-use tools for the evaluation of industry designs [1]. We will therefore be more interested in mesh quality rather than speed.

1.1.1 Finite Element Analysis

The main product currently under development at Enmesh is called DesignCheck. Using this program, engineers can quickly verify that the designs they are working on can handle the real world conditions they will face when deployed. DesignCheck is based on the finite element method, a numerical method for solving mechanical engineering problems like heat transfer and structural analysis. Since these kinds of problems involve many complicated factors, solving them analytically is usually not possible, and numerical methods are thus necessary [15].

The finite element method is a divide-and-conquer approach in that a geometrically complex domain is broken down into smaller subdomains of simple geometry. These subdomains are called finite elements and a given problem is independently solved for each element. The elements are then assembled together again in order to obtain the solution over the whole domain [19].

1.1.2 Incentives

The finite elements are defined over a 3D-net, in this case an octree, where each corner of the octree cells contains the shortest signed distance from that point to the underlying surface. This data might then be used to extract a polygonal representation of the surface. There are many reasons why this functionality is needed. One is that we want to be able to render the geometry at a resolution corresponding to the current refinement level of the octree grid [29]. Another is that geometry extracted from CAD files might result in suboptimal triangle meshes. This will enlarge the errors in the solutions. One way to cope with this problem is to convert the original model to a volumetric representation and then extract a new contour. This process is referred to as remeshing [27].

1.2 Purpose

The data structure we start out with is a signed distance field defined over an octree. Our task is to convert this representation into a *manifold triangle mesh*. The definition of a manifold surface will follow, and why it is important that the meshes obey this property.

1.3 Method

Before the software construction work began, a pre-study was made where relevant research material was studied in order to get a broad overview of the field. The starting point was a handful of articles given by the supervisor at the company. During the pre-study, which lasted for about a month, the in-house framework that the implementation was going to be a part of was also studied. This framework was entitled *Box* and we will refer to it by that name from here on.

1.4 Materials

Since Box was implemented in C++, this was the language of choice. Also, a couple of frameworks were included; namely OpenSceneGraph[1], boost[2] and QT[3]. The coding environment used was Visual Studio 2008, and git[4] was chosen for source version control.

A few files with distance fields, as well as a tool for generating such fields for some simple shapes in a desired sampling resolution, were provided by the supervisor at Enmesh.

A tool developed at the company was used for testing various aspects of the extracted surfaces. It was specifically utilized for verifying that the generated meshes became manifold. For visually studying the meshes and manually asserting their quality, the open source program Meshlab[5] was used. Some additional visualization applications were developed during the course of the project. These are discussed in the appendix.

1.5 Scope and Contributions

This report provides an overview on the subject of generating manifold surfaces from an adaptive distance field. The main contributions are implementation details and more thorough descriptions of certain aspects of the utilized algorithms. The hope is to reduce the development time for other software developers who seek to implement these techniques.

There will be no discussion on how to generate the distance fields since a tool for this was already provided in Box. We therefore point interested readers to [2] which provides detailed instructions.

1.6 Report Structure

After introducing some basic terminology and concepts important for this particular field of study, the previous work that our implementation is based on follows. A description of the implementation is then given, and thereafter the results. Conclusions, discussion and possible future improvements end the report.

[1]www.openscenegraph.org/
[2]www.boost.org/
[3]qt.nokia.com/
[4]git-scm.com/
[5]meshlab.sourceforge.net/

2 Fundamental Concepts

2.1 Implicit Surfaces

One of the most basic terms in the subject area for this thesis is *implicit surfaces.* An implicit surface is defined as the set of points satisfying the *implicit function*[6] $f(x, y, z) = c$. c is called the *isovalue* and the surfaces are often referred to as *isosurfaces* [28].

A point p is on one side of the surface if $f(p) < c$ and on the other side if $f(p) > c$. All points where $f(p) = c$ therefore lie exactly on the surface. The convention is to classify negative values as being outside the surface and positive values as being inside the surface [13]. In order to adhere to this convention, we can always rewrite the implicit function as $f(x, y, z) - c = 0$ [25]. This will make matters simpler for us since we can always assume that the surface partitions the universe into negative and positive space. In the following we will therefore always assume that the surface is located where the distance function evaluates to zero.

There are basically three different ways in which an implicit function can be specified; as a mathematical function, procedural method or by discrete samples. An example of an implicit mathematical function is the unit sphere, which is the set of all points satisfying $f(x, y, z) = x^2 + y^2 + z^2 = 1$. For a procedural method (also referred to as a black box function), the evaluation is done algorithmically.

In this work, f is specified by discrete samples, i.e. we have a grid of values approximating f at the vertices of the grid. This is commonly referred to as a *scalar field.* Adjacent points form cubes, also referred to as *voxels* or *cells.* Intermediate values are calculated by trilinear interpolation over each voxel [13]. As we have mentioned, our dataset specifically describes the minimum signed distance from each point to the surface. In the rest of the report, the terms scalar field, signed distance field and volume data are interchangeable.

2.2 Spatial Hierarchies

If the samples are laid out uniformly spaced, all voxels will be of equal size, and we denote these grids *uniform grids.* However, for reasons that will become evident later, it is often desirable to sample the distance field non-

[6]Functions of this type are defined as being implicit because they do not describe how to obtain the value of a dependent variable explicitly from an independent variable. In contrast, an explicit function gives us a mapping from an independent variable to a dependent variable, i.e. an equation of type $y = f(x)$ where we have a direct way of evaluating y through x [37].

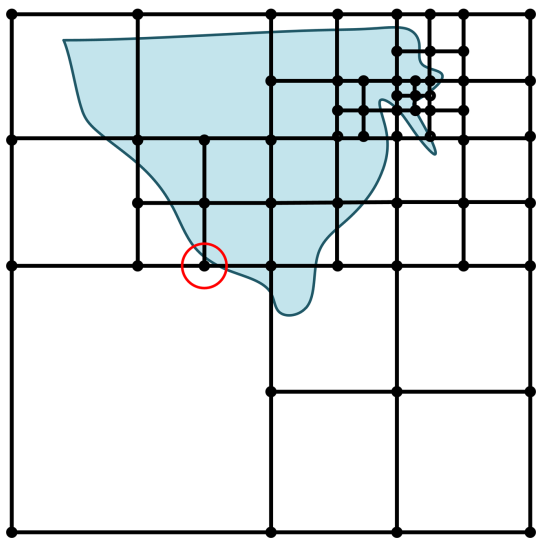


Figure 1: Example of a quadtree that is *adaptively* refined in order to capture features of the underlying surface. If the subdivided cell giving raise to the vertex marked out by the red circle is not subdivided, this is also a *restricted* quadtree.

uniformly. We call these structures *adaptive grids*. A number of alternatives exist for how to store such data [9] and in the work presented here, octrees are used. Specifically, we work with *restricted* octrees, since this is what is provided by Box. Restricted octrees differ from unrestricted octrees in that adjacent voxels cannot differ by more than one level of refinement [18].

2.3 Gradient

The *gradient* of a scalar field is a vector field, where each vector points in the direction of the greatest rate of increase from that point, and the length of the vector gives the magnitude of that increase [35]. At every point along the surface, f is constant and equals the isovalue. The gradient component for all vectors tangent to the surface is thus zero, and it follows that the gradient vectors are parallel to the surface normals at these points [33].

If we follow the convention that points with negative distance values are located outside the surface, and points with positive distance values are located inside the surface, we need to flip the gradient vectors if they should be used as surface normals. This is because the scalar field grows *toward* the surface.

Formally, gradients are defined as the partial derivatives of the function f. For this to work, the function must be continuous and differentiable. For scalar fields, where f is not evaluated as a function in the standard sense, other ways of calculating the gradient are used. In [13], Bloomenthal states two ways of performing these calculations, along with the error estimate for both.

2.4 Manifoldness

Usually, our starting point is a body generated from a 3D Computer Aided Design (CAD) program. Since the designs are created for real life production, it is important that they are described in a mathematically exact way. In CAD software, this is often achieved using NURBS curves [34].

Due to the exact description of the original model, we do not want to introduce unnecessary errors when constructing our meshes. Such an error would for example be to duplicate an edge in the triangulation since this leads to an unconnected model, and thus a non-closed surface. Certain algorithms that we might want to execute on the surface later also require a closed and well-behaved triangulation [29].

We are now touching upon the mathematical concept of *manifolds*. Every manifold has a dimension, which in our case is 2, since we are producing surfaces in 3D space[7] [16]. We will thus more specifically be interested in 2-manifolds. The terms manifold and 2-manifold will hereafter be used interchangeably.

The formal definition of a manifold surface is that for every point on the surface, the neighborhood is homeomorphic to a disk. With neighborhood we simply mean an arbitrarily small region surrounding a point. The meaning of a shape being homeomorphic to another shape is that they can be transformed into each other by stretching and bending, but no tearing [11]. It is then said that a homeomorphism exists between the two shapes and that they are topologically equivalent [36]. A classic example of this concept is the topological equivalence that holds between a coffee cup and a donut[8].

A consequence of the definition is that the intersection between any two triangles in the mesh has to either be the empty set (i.e. no intersection), a common vertex, a common edge or a common face [4]. Thus, there will be no overlapping triangles. Additionally, there can be no boundary triangles or holes in the surface since this would oppose the demand that all points should be locally homeomorphic to disks. The surface will therefore be bounded and closed, i.e. it will be watertight. For completeness, it should be noted that

[7]In mathematics, there is a difference between a sphere, which describes a surface and thus is of dimension 2 and a solid ball which has a volume and therefore is of dimension 3.

[8]This animation shows the concept:
http://upload.wikimedia.org/wikipedia/commons/2/26/Mug_and_Torus_morph.gif

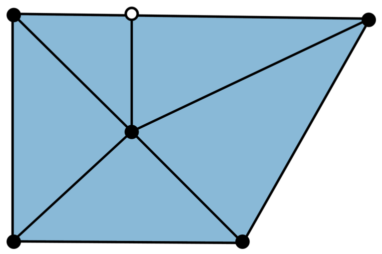


Figure 2: The non-filled vertex is a so called *hanging node*.

the concept of manifold with boundary exists, and that each point then is approximated either with a disk, or with a half-disk [13].

Before ending the discussions on manifolds, we also note that the definition prohibits the existence of *hanging nodes* (see figure 2). A hanging node refers to the occurrence of a vertex lying on the edge of another triangle [20]. This is undesirable since it introduces a discontinuity [14].

3 Previous work

In the following, we will first look at how contouring can be done on uniform grids. We will look at one such algorithm, Marching Cubes, which is arguably the most popular one and also one of the earliest. The algorithm as described in the original paper might however produce meshes with properties often undesirable, and since its publication, much research has been devoted to improve the algorithm [10]. We will specifically discuss one drawback with standard Marching Cubes, namely *face ambiguity*. The reason why we focus on this particular problem is because it leads to non-manifold geometry.

For representing shape, uniform grids are usually not the best choice [9]. We will show the gains by instead using an adaptive grid, specifically an octree. Unfortunately, contouring methods designed for uniform grids are not easily adapted to octrees. Marching Cubes is attractive due to its simplicity and efficiency, and many attempts have therefore been made to extend the algorithm to work on adaptive structures [25]. The method implemented in this thesis, Dual Marching Cubes, will be discussed. We will also describe why it was chosen instead of other candidate methods.

3.1 Marching Cubes

In 1987, William E. Lorensen and Harvey E. Cline published a now classic paper where they describe an algorithm for converting a scalar field into a triangle mesh. They were at this time working for General Electric. The company was then developing a new rendering system for visualization of volumetric data from medical scans. At a seminar, one of the developers in this project proposed a challenge for the attendants; to come up with a technology that could replace the current rendering technique with a polygon based approach.

Lorensen and Cline were participating in the seminar, and took on the challenge [7]. After a while they came up with a solution, and named it Marching Cubes. The reason for the name is because after processing each voxel (cube), the algorithm marches on to the next [8].

Marching Cubes was not the first algorithm for extracting an isosurface [24]. It has however become one of the most popular methods, most likely due to its simplicity and efficiency [25] [28]. The algorithm was patented, but since the patent has expired, the algorithm is now free to use [3].

3.1.1 Algorithm Description

Since each cube consists of eight corners and each corner is either inside (positive) or outside (negative) the surface, there are 256 (2^8) different basic ways for how the surface might intersect a cube. If we know the relation of each corner to the surface, we also know which edges of each cube that are intersected by it. Since we have a scalar value specifying the signed distance to the surface at each corner, we can then interpolate over each edge with different signs at its two vertices to find out where the scalar value equals zero. This is the point of intersection. The connectivity of the triangles formed by the vertices on the cube edges are determined by a triangle lookup table. Note that due to the fact that polygons cannot extend outside their parenting voxels, meshes produced by Marching Cubes contain no hanging nodes and no overlapping geometry.

Voxel Triangulation Describing the triangulation for all the 256 cube configurations is both tiresome and error prone. By utilizing *rotational* and *reflective* symmetry, Lorensen and Cline identified 15 basic cases. Starting out with these 15 cases, all the remaining 241 combinations can then be generated algorithmically [8]. However, since the base cases have to be pinned down manually, the process is still subject to human error.

As a consequence, more systematic approaches have been developed. Bhaniramka et al. gives convex hull based algorithms in [21] and [22] that generates lookup entries automatically for *any* dimension. Given the time frame available for this project, it was decided that such approaches would be too time consuming to implement. Moreover, such methods were mainly

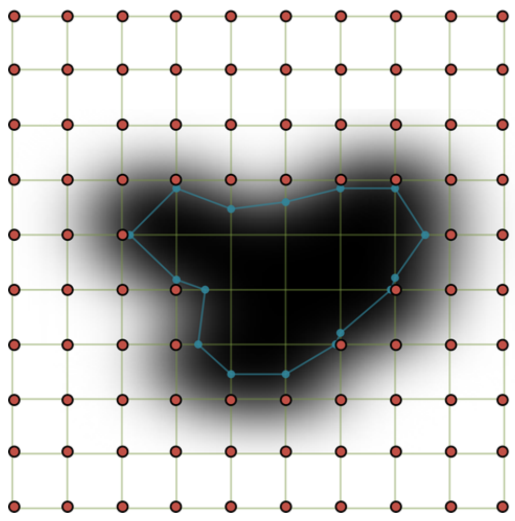


Figure 3: Marching Cubes in two dimensions, called Marching Squares [6]. The red dots indicate vertices that are outside the surface, the blurred black shape is the distance field, and the blue lines represent the extracted contour.

developed for constructing isosurface patches when the table sizes become unmanageable for manual handling; in four dimensions for example the number of table entries becomes $2^{16} = 65536$.

3.1.2 Face Ambiguity

One of the major problems with standard Marching Cubes is that adjacent cube faces might be ambiguous. For a face to be ambiguous, it needs to have two opposite vertices marked as outside, and the other two marked as inside. Referring to figure 4, we see that this is true for certain faces of cases 3, 6, 7, 10, 12 and 13. Figure 5 shows what will happen at the interface between two adjacent cubes sharing such a face; we get holes in the surface and thus non-manifold geometry. There have been many attempts to solve this problem, where some involve face tests for resolving the ambiguities, for example [32]. However, a direct solution is to note that it is the reflective symmetry that is the source of the problem. By extending the 15 base cases to 23 and only make use of rotation when constructing the remaining cases, the problem is avoided [24]. This, together with the absence of intersecting geometry, makes it possible to use Marching Cubes for extracting manifold contours.

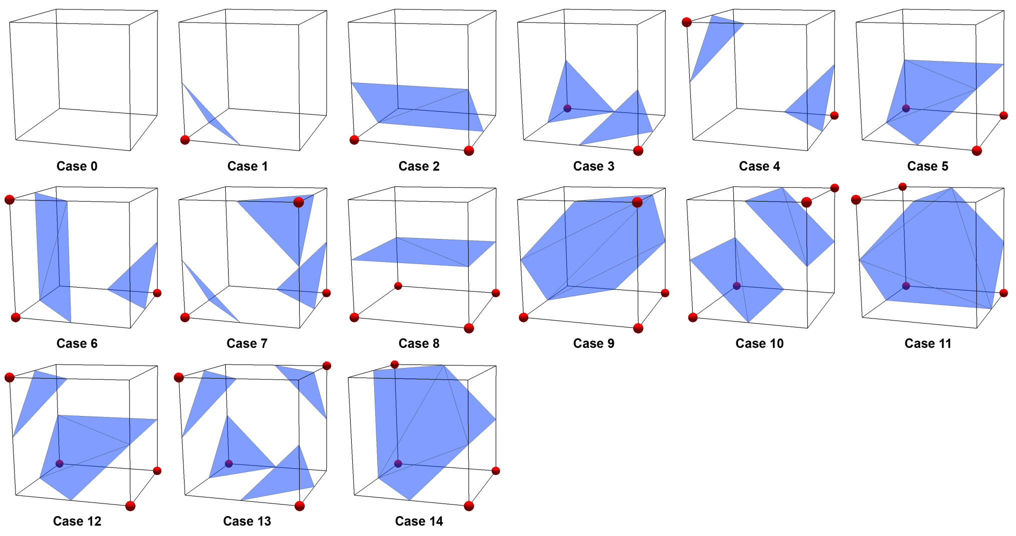


Figure 4: The 15 base cases of Marching Cubes. Filled vertices have a negative sign, i.e. they are outside the surface.

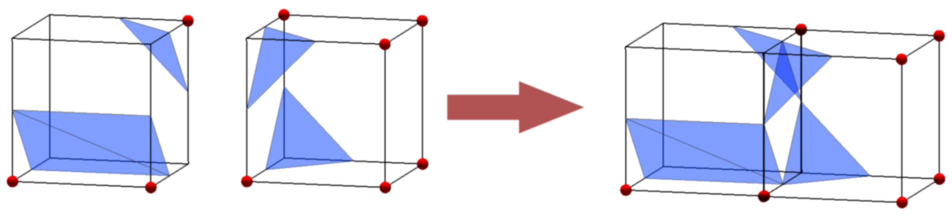


Figure 5: Two adjacent cubes sharing an ambiguous face, leading to a crack in the surface.

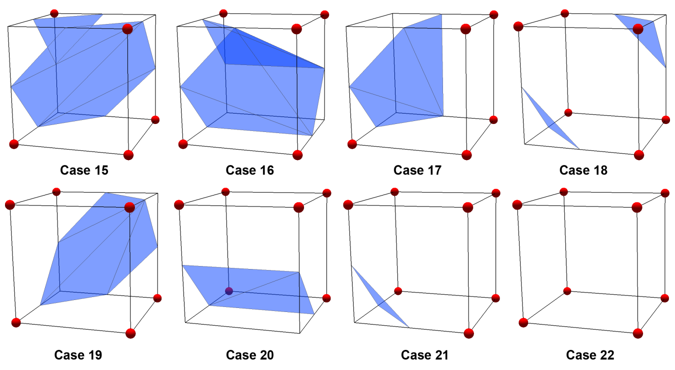


Figure 6: The additional 8 cases for avoiding face ambiguity.

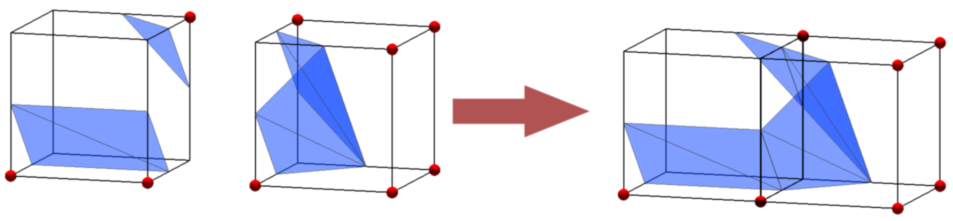


Figure 7: The face ambiguity shown in figure 5 resolved by enhancing the lookup table. In this particular example, the addition of base case 19 solves the problem.

3.2 Adaptive Distance Fields

Distance fields laid out uniformly have been extensively used, but have a major drawback; the grid has to be very fine grained in order to capture thin features and sharp edges of the underlying surface. Ultimately, the grid size becomes unmanageable in terms of memory [9]. Furthermore, if we increase the grid density to capture these features, we will as a consequence end up with a grid that is excessively fine grained at areas where the surface exhibits little or no change (flat areas for example) [25].

An alternative to uniform grids is to construct an adaptive distance field, where the surface is more densely sampled at interesting locations, i.e where the surface exhibits rapidly changing curvature. In [9], Frisken et al. provides a discussion on how to construct such a field. An octree is a natural way for structuring the resulting data, and is what we will use here.

3.3 Isosurface Extraction on Octrees

As we have seen, there are good reasons for adaptively sampling the distance field. Unfortunately, applying Marching Cubes directly to the cells of an octree yields poor results. The problems arise at the interfaces between neighboring voxels at different subdivision levels. This leads to cracks in the surface. When applying Marching Cubes to a uniform grid, the same decisions are taken on both sides of a face and the surface will therefore always be connected [18].

3.3.1 Dual Marching Cubes

Numerous articles have been written on the subject of making Marching Cubes work with adaptive grids. For our purposes, the optimal choice seemed to be the Dual Marching Cubes method, developed by Scott Schaefer and Joe Warren in 2004. The idea is to construct a new grid from the octree,

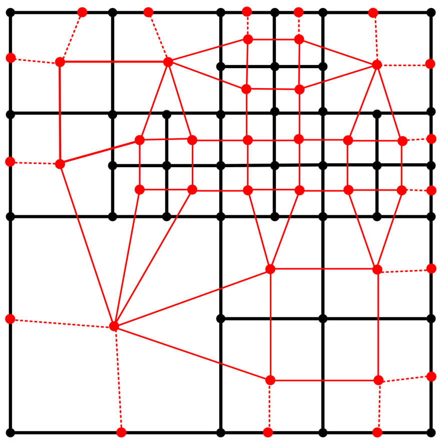


Figure 8: The connectivity of the dual grid (red) constructed from a quadtree (black). The dashed lines form cells connected to the boundary of the octree. These boundary cells are not generated in our implementation since they were unneeded; the isosurfaces we worked with never crossed octree voxels at the boundary.

where the properties of this new grid make it possible to contour it with Marching Cubes. As have been shown above, if the face ambiguity problem is solved, Marching Cubes can be used for extracting crack-free surfaces. Since Marching Cubes only works for grids where there are no differences in subdivision depth, this new grid must enjoy these qualities. Figure 8 shows that this indeed is the case.

As can be seen from the figure, the dual cells are not always square, but might degenerate into other types of quadrilaterals and even triangles. This happens when the refinement depths differ for the quadtree cells holding the binding vertices. In three dimensions, we will get an even larger flora of different polytopes. However, since the algorithm constructing the dual grid actually duplicates the coinciding vertices and thus also duplicates the collapsed edges, the resulting cells are still topologically equivalent to cubes. The cells can thus be contoured using standard Marching Cubes lookup tables [25].

Reasons for choosing Dual Marching Cubes We will now explain why we chose to base our implementation on Dual Marching Cubes. In addition, we will compare certain of its qualities to other algorithms. Obviously, an important factor was the time frame available for the project. As a consequence, algorithms that would require modifications or extensions to the Box framework were considered less interesting.

1. Compatible Data Dual Marching Cubes operates on a scalar field, which is what is provided through the Box interface. Methods like Manifold Dual Contouring [26] and Cubical Marching Squares [10] both produce surfaces of high quality. These methods however rely on edges being tagged with exact intersection points and precise normals, so called *hermite data.*

Even if we could use the scalar field for interpolating the intersection points and evaluating the surface normals through the gradient, it was uncertain whether this would give any good results for such algorithms since neither the interpolated intersection points nor gradient normals would enjoy such high quality as when using hermite data.

2. Flexibility Dual Marching Cubes consists of well defined steps, where the details of the steps are somewhat flexible. The first step for example is to place *dual vertices* inside each voxel of the octree, where these vertices are placed on features of the implicit function. In the Dual Marching Cubes article, this process is referred to as *feature isolation.* In this way, a more accurate representation of the surface will be generated. However, other strategies for vertex placement are possible. For example, the vertices could simply be placed in the center of each voxel.

This is an approach mentioned by the authors of the article, and it was for instance chosen for the Dual Marching Cubes implementation discussed in [18]. The surfaces produced when centering each dual vertex resemble those generated by SurfaceNets [25], an extraction method aimed at producing high-quality meshes for finite element analysis [5]. By choosing this strategy, we could focus on generating manifold surfaces first. Given time, features could be implemented later.

3. Straightforward Solution Many methods make use of some sort of patching for connecting the mesh where cracks have occurred due to differences in node refinement depth. One such method is called Adaptive Marching Cubes. Even if the meshes produced by this algorithm contain no holes, the surface will be discontinuous in areas where patching have occurred. This could probably be solved by executing some sort of post processing on the mesh [23]. However, compared to Dual Marching Cubes, this procedure would be less straightforward and the result more uncertain.

4. No Inter-cell Dependencies Dual Marching Cubes does not introduce inter-cell dependencies. Inter-cell dependency means that the surface extracted inside one voxel might depend on neighboring voxels. This increases algorithm complexity and might severely impact contouring time.

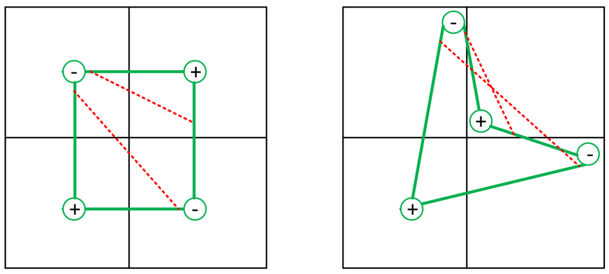


Figure 9: In the left figure, the dual vertices are placed at the centers of the voxels, and the dual cells are always convex. In the right figure, the dual vertices have been moved to better align with the underlying surface, resulting in non convex dual cells. This, in turn, leads to intersecting triangles and a non-manifold surface.

Even if we are not targeting real-time, models might contain several millions of voxels, and introducing inter-cell dependency could be a problem.

Problems and Solutions

1. Non-convex Cells Surfaces of higher quality might be generated if the feature isolation step of Dual Marching Cubes is implemented properly. However, when not placing the dual vertices regularly, the algorithm might produce intersecting triangles due to non-convex dual cells. Figure 9 illustrates this problem. The original Dual Marching Cubes paper did not discuss this drawback, but it was later pointed out in [12]. Since we use the strategy of always centering the dual vertices inside each voxel, the problem is avoided altogether.

2. Sliver Polygons Our dual vertex placement strategy unfortunately leads us to another problem, namely that of sliver polygons. A sliver polygon is a polygon of zero, or very small area [25]. For finite element methods, the stability of the solution is increased if the triangles are more equally sized [5]. Figure 10 shows a close-up of the surface of a contoured sphere containing sliver polygons. The authors of Dual Marching Cubes solve the problem by snapping the dual vertices to lie exactly on the isovalue if it lies within some tolerance. This is done when performing feature isolation.

3. Topological Holes In a paper from 2007, Kazhadan et al. compares an implementation of their own for unrestricted octrees with Dual Marching Cubes [18]. They proceed in the same way as is done is this thesis, namely

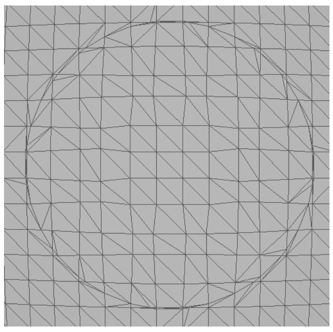


Figure 10: Part of the surface of a sphere containing sliver polygons.

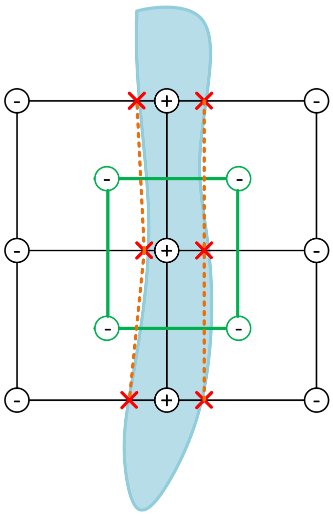


Figure 11: Due to a rapidly fluctuating distance field, the dual grid (green square) misses a portion of the surface (blue shape). The dashed line shows the contour as it is defined for the quadtree.

that they place the dual vertices in the centers of each voxel, thus leaving out the feature isolation procedure. In their results, they show that topological holes might appear at locations where the surface is thin. Note that the surface is still closed though as this is not the same as a crack. To see why this might happen, consider a situation in two dimensions where the surface intersects the quadtree edges in such a way that the dual vertices are all classified as being outside the surface (see figure 11). The new dual edges between the vertices will thus not intersect the surface. We are currently unaware of any solution to this problem.

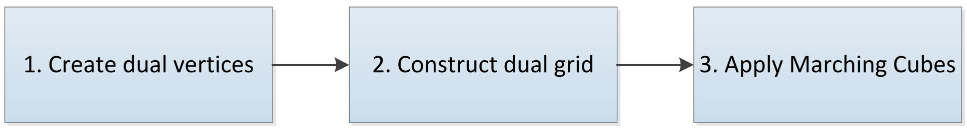


Figure 12: The three steps of our implementation. The output from each step is used as input to the next.

4 Implementation

The implementation is based on the octree traversal algorithm from Dual Marching Cubes for generating a grid dual to the octree. This dual grid is then contoured using a Marching Cubes implementation that avoids face ambiguities. The process can be broken down into three steps, shown in figure 12. Our main contribution in this section will be to provide details for the second step of the algorithm, since the original article on Dual Marching Cubes restricts the discussion to quadtrees.

4.0.2 Creating the Dual Vertices

Box provided an easy way of traversing the octree in a top-down manner, enumerating all the voxels. For each voxel, the center position is first calculated. The gradient is thereafter sampled, inverted and normalized for use as the surface normal. Finally, the distance field at the center point is sampled. This data is stored in a DualVertex class. When building the dual grid, we need to access this data for each voxel we visit. All the instances of DualVertex are therefore stored in a map where lookup is done by voxel.

4.0.3 Constructing the Dual Grid

In this step, the octree is traversed top-down for the second time, in order to bind together the dual vertices belonging to the eight voxels that share a common vertex in the octree. In other words, for each vertex in the octree, we create a topological cube where the corners of that cube are the dual vertices of the voxels sharing that octree vertex.

The procedure is simpler to implement if the dual cells defined for vertices on the octree boundary are avoided. This is because the traversal algorithm finds voxels sharing a common vertex. Since the vertices on the borders are not shared by eight voxels, they have to be handled in a special way. Refer to figure 8 for a visual explanation of border cells.

For the distance fields used, the contour never crossed any octree cells at the boundary of the octree. It was therefore unnecessary to implement support for boundary cells since they would never contribute to the surface anyway. In the paper on Dual Marching Cubes, Schaefer and Warren suggest how the boundaries can be included for quadtrees. Should the border cells be needed later, the approach they describe should be easily extendable to octrees.

Driving the construction of the dual grid are four recursive methods; `nodeProc(n)`, `faceProc(n1,n2)`, `edgeProc(n1,n2,n3,n4)` and `vertProc(n1,n2,n3,n4,n5,n6,n7,n8)`. The `n*` arguments are all nodes in the octree. Figure 13 shows how these methods are called from a subdivided node in nodeProc. The recursion is started by calling `nodeProc` with the root node of the octree as argument.

In the implementation, `edgeProc` is divided into `edgeProc_X`, `edgeProc_Y` and `edgeProc_Z`. Similarly, `faceProc` is broken up into `faceProc_XZ`, `faceProc_XY` and `faceProc_YZ`. The reason for this is that we need to keep track of how the nodes we recurse on are spatially related. An alternative is to send an additional argument to the methods, specifying the adjacency relationship. This however led to cluttered code. Since further recursion on a face/edge always happen along the same plane/axis, the methods naturally lend themselves to specialization. In the pseudo-code below we will exemplify with `edgeProc_X` and `faceProc_XY`. The only code differing in the other versions are which nodes are picked out for further recursion.

On many occasions we will have a situation in `faceProc`, `edgeProc` or `vertProc` where one or several nodes are subdivided while one or several nodes are leaves. In that case, the leaf nodes are reused as arguments in further recursive calls. Therefore, when we write that we "call `vertProc` with the nodes sharing the face center vertex" in the pseudo-code for `faceProc_XY` for example, some of these nodes might be duplicated; it is not always the case that we have eight *unique* nodes. This type of recursion on an octree with different refinement depths is the reason why we end up with several different types of polytopes in the dual grid and not only cubes. Following is pseudo-code for the four methods.

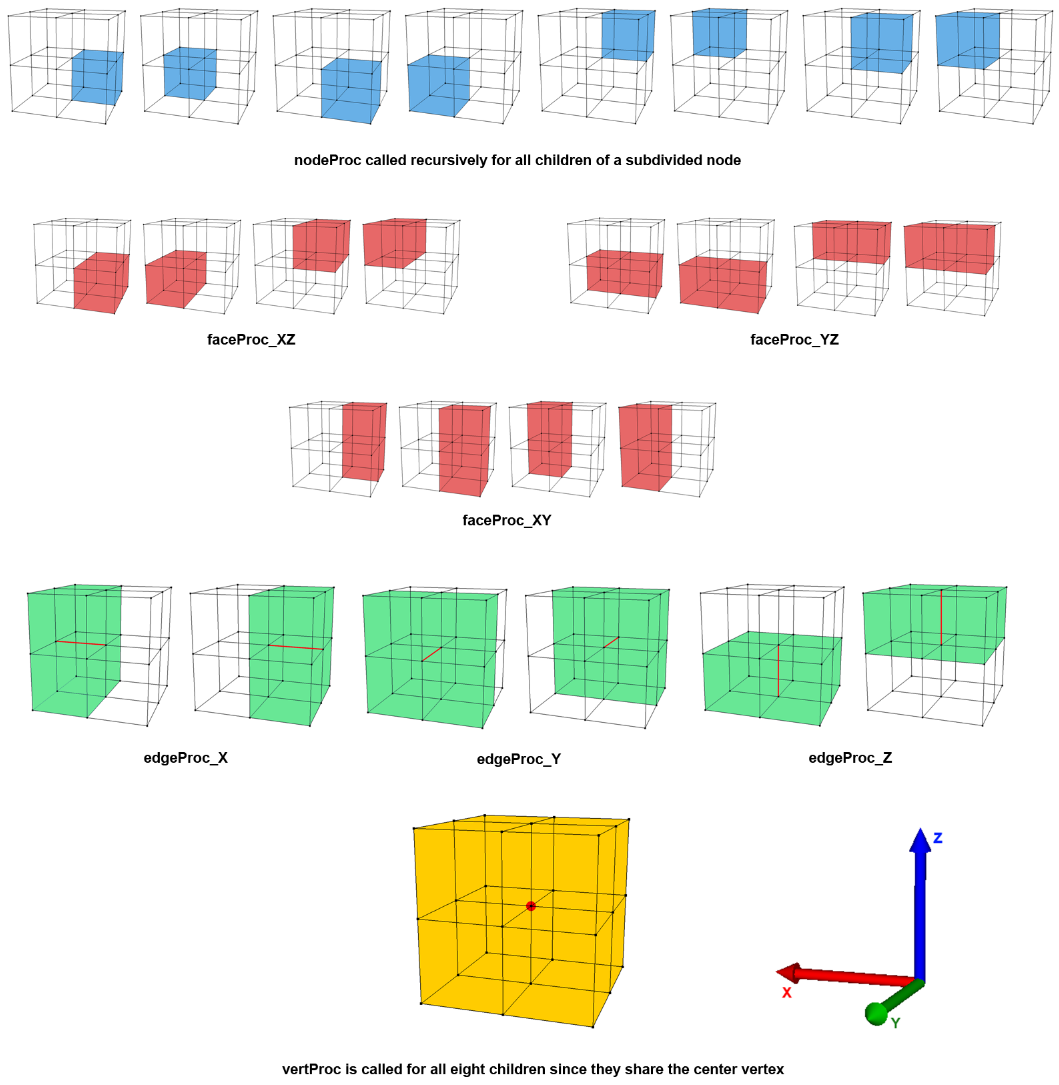

nodeProc called recursively for all children of a subdivided node
faceProc_XZ
faceProc_YZ
faceProc_XY
edgeProc_X
edgeProc_Y
edgeProc_Z
Z
X
Y
vertProc is called for all eight children since they share the center vertex

Figure 13: This image shows how nodeProc, faceProc, edgeProc and vertProc are called from nodeProc for a subdivided node.

Procedure 1 Recursive method nodeProc

```
1: procedure NodeProc($n$)
2:     if $n$ is subdivided then
3:         call nodeProc for all eight children
4:         call faceProc_XZ for children sharing a face in the XZ plane
5:         call faceProc_XY for children sharing a face in the XY plane
6:         call faceProc_YZ for children sharing a face in the YZ plane
7:         call edgeProc_X for children sharing an edge along the X-axis
8:         call edgeProc_Y for children sharing an edge along the Y-axis
9:         call edgeProc_Z for children sharing an edge along the Z-axis
10:        call vertProc with all eight children; they share a common vertex
11:    end if
12: end procedure
```

Procedure 2 Recursive method faceProc_XY

```
1: procedure FaceProc_XY($n1$, $n2$)
   ▷ When at least one of the two nodes sharing a face is subdivided, we get
   four sub-faces and four edges meeting at a single vertex at the center of
   the face
2:     if $n1$ is subdivided or $n2$ is subdivided then
3:         call faceProc_XY for nodes sharing sub-faces
4:         call edgeProc_X for nodes sharing an X-edge
5:         call edgeProc_Y for nodes sharing a Y-edge
6:         call vertProc with the nodes sharing the face center vertex
7:     end if
8: end procedure
```

Procedure 3 Recursive method edgeProc_X

```
1: procedure EdgeProc_X($n1$, $n2$, $n3$, $n4$)
   ▷ When at least one node sharing an edge is subdivided, the edge is split
   into two smaller edges separated by a vertex
2:     if at least one node $n1..n4$ is subdivided then
3:         call edgeProc_X for nodes sharing smaller edges
4:         call vertProc with the nodes sharing the separating vertex
5:     end if
6: end procedure
```

Procedure 4 Recursive method vertProc

```
procedure VertProc($n1$, $n2$, $n3$, $n4$, $n5$, $n6$, $n7$, $n8$)
    if all arguments $n1..n8$ are leaves then
        init DualCell $d$
        for all $n \in n1..n8$ do
            add the DualVertex for node $n$ to $d$
        end for
        add $d$ to DualGrid
    else
        call vertProc using the nodes themselves as arguments if they are leaves. For subdivided nodes, pick the child that shares the vertex we are recursing on.
    end if
end procedure
```

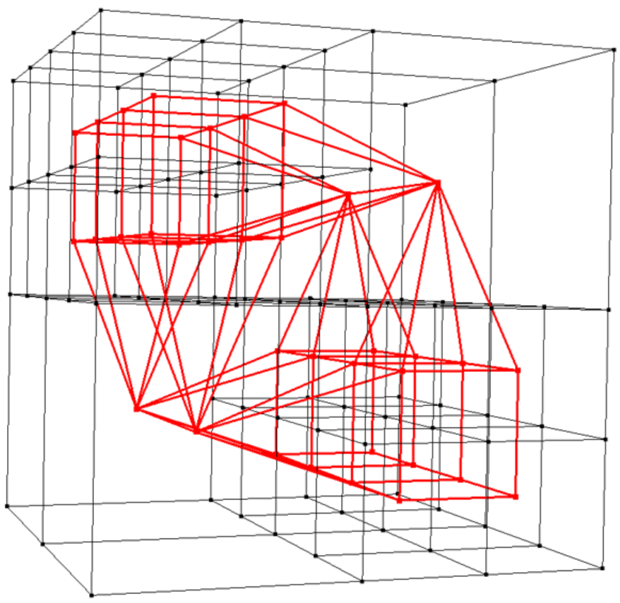


Figure 14: Dual grid (red) created from octree (black).

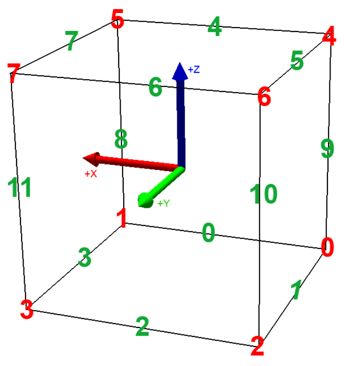


Figure 15: Vertex and edge numbering.

4.0.4 Applying Marching Cubes to the Dual Grid

The Marching Cubes implementation makes use of two different lookup tables. The first one, `EdgeTable`, is an array of 12-bit numbers, where the indexing into the array is done with an 8-bit number. Each bit in the number used as index corresponds to a vertex in the cube. If a bit is 1, that vertex is outside or exactly on the surface. A bit set to 0 means that vertex is completely inside the surface. `EdgeTable` consists of entries for all 256 voxel configurations, where each 12-bit number specifies which of the 12 edges that are intersected by the surface for that particular configuration. The numbering scheme we chose is shown in figure 15. The second table, `TriTable`, is indexed by the same 8-bit number as `EdgeTable`. Each entry is an array of 15 integers and each triplet of numbers in the array specifies which of the intersected voxel edges to connect to form a triangle. The reason why the array is of size 15 is that the maximum number of triangles for a voxel is five.

The workflow of our Marching Cubes implementation is:

1. Construct an 8-bit index k based on the vertex classifications of the cell.

2. Use bitmasking on the number fetched at index k of `EdgeTable` to pick out which edges in the voxel that are intersected.

3. For each edge, interpolate between its endpoints to calculate the intersection point and normal.

4. Finally, store the connectivity of the generated vertices, specified at index k of `TriTable`.

5 Results

For the tested input files, the algorithm has so far not failed in generating manifold meshes. This has been verified by an in-house tool for asserting mesh quality, and also by manually inspecting the meshes in Meshlab. Avoiding reflective symmetry when generating the Marching Cubes lookup tables removed some small cracks which otherwise occurred. In figure 16 we see a close up of the same section of two meshes, where the left one was generated from 15 base cases, and the right one from 23. Most meshes contain sliver polygons.

The time it takes to create the meshes is acceptable. Of the total time approximately 50% is spent creating the dual grid, 40% is spent placing the dual vertices, and 10% is spent running Marching Cubes. For models of larger size than the ones we have tested with, memory consumption might lead to problems.

Table 1: The time is the total contouring time (all the three steps of the algorithm) and the memory usage specifies the maximum amount simultaneously allocated while extracting the contour.

Model	Num voxels	Num faces	Time (s)	Memory usage (MB)
Sphere	14232	9740	0.36	13
Bunny	88019	60900	2.3	51
Cube	105400	47516	3.1	61
Console	117958	93462	3.2	68
CSG	994092	791346	31	548
Knuckle	2066772	1596964	62	1080

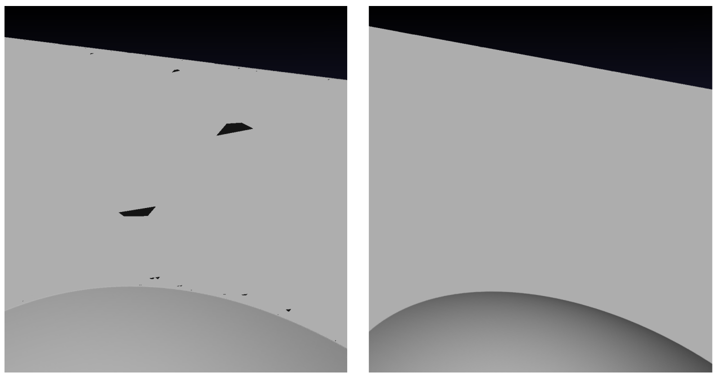


Figure 16: Cracks appear when using Marching Cubes lookup tables making use of reflective symmetry. By only exploiting rotational symmetry, the problem disappears.

6 Conclusions

By using the simple strategy of placing the dual vertices uniformly spaced, and by avoiding reflective symmetry when generating the Marching Cubes lookup tables, Dual Marching Cubes always generates manifold surfaces. Because of how the dual grid is defined, some parts of the contour defined on the octree might however be left out in the dual grid. This leads to topological holes in the final surface. We have not observed this problem for the surfaces in our test suite, but there is currently nothing in our implementation preventing it from happening.

In the paper describing Dual Marching Cubes, the authors mention that their Marching Cubes algorithm is an extension of Marching Cubes adjusted to work on the dual grids. No such extension was needed; in our implementation we contour the cells exactly as would have been done on a grid of axis-aligned cubes of uniform size.

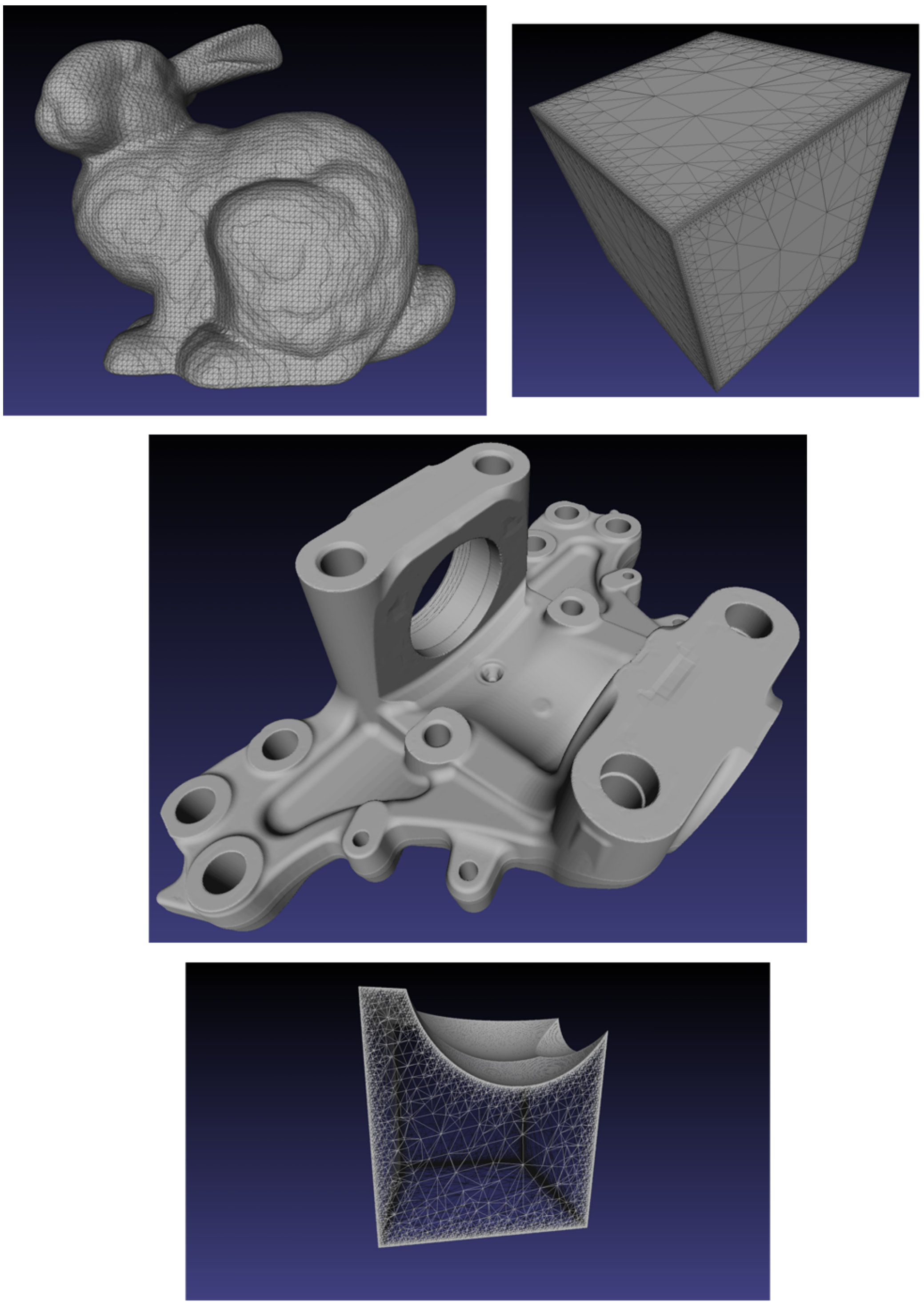


Figure 17: Some extracted contours. From left to right, top to bottom: Bunny, Cube, Knuckle and CSG (referring to the models listed in table 1).

7 Discussion

Much literature has been written on the subject of extending Marching Cubes to avoid ambiguities, and also for making the algorithm work on adaptive distance fields. Even if the algorithm has been around for well over twenty years, and its drawbacks have been well described and studied, the large amount of methods proposed for solving these problems[9] hints that it is difficult to construct an algorithm that solves them all while not introducing new ones. For example, the feature isolation step of Dual Marching Cubes aims to solve the problem that sharp edges become rounded in the original Marching Cubes algorithm. However, allowing the dual vertices to move freely inside their parent voxels might lead to overlapping triangles and a non-manifold surface.

Choosing a contouring method might be hard since so many variants exist, often producing seemingly similar results. If a requirement exists on watertight surfaces and the distance field is defined over an octree, we definitely recommend the approach presented in this report. It is reasonably fast for non-realtime applications, and is relatively easy to implement. It however proved to be somewhat harder than originally thought to extend the quadtree traversal algorithm described by Schaefer and Warren to three dimensions. Hopefully, the pseudo-code given in this report might speed up the process for other developers who want to implement and evaluate the algorithm.

[9]We have only described the problems most relevant for this particular study and refer to [24] which provides an excellent overview of the research done on the subject of Marching Cubes.

8 Future Work

Sliver Polygon Elimination Apart from making the models look unattractive, meshes containing sliver polygons are not optimal from a finite element method perspective. When discussing the problem, we pointed out that Schaefer and Warren solve the problem in the feature isolation step of Dual Marching Cubes. Their solution makes use of the somewhat involved procedures developed for finding the features of the implicit function. If feature isolation was to be integrated, it would be natural to use their approach for eliminating sliver polygons. If not, this approach might be too complicated for a relatively simple problem.

A different solution is proposed in [28]. Here, an extended Marching Cubes lookup table is used with 6561 (3^8) entries instead of the standard 256. A vertex of a voxel might then be classified as lying *on* the surface in addition to being outside or inside. When the distance field value is close enough to the isovalue, the value is snapped to exactly equal the isovalue (this contrasts the approach used in the Dual Marching Cubes paper where it is the *position* of the dual vertex that is snapped). We see no reason why this method should not work with the polytopes of the dual grid as well. It should also be easy to test this approach since the authors have released a tool for building these extended lookup tables[10].

Minimizing Memory Usage In the current implementation, memory usage is a problem for larger models. When generating the surface from the largest octree we have been testing on, the application peaks at over one gigabyte of RAM. Certain volumetric datasets might be up to sixty times larger, for example the high resolution scan of Michelangelo's David from the Digital Michelangelo Project [17].

However, this problem could be improved upon by collapsing steps two and three of the algorithm. That is, instead of creating the dual grid and storing it for Marching Cubes later, we would contour each cell directly in `vertProc` and also immediately write the resulting polygons to disk. We can thereafter dispose the memory for the dual cells directly. Should this not be enough, we can also calculate the positions of the dual vertices and sample the gradient and distance field in `vertProc`. We then no longer need to keep more than eight dual vertices in memory at the same time. The field sampling procedure is however computationally expensive. Processing each octree voxel more than once might therefore have a major impact on total contouring time.

[10]http://www.cse.ohio-state.edu/research/graphics/isotable/

References

[1] Enmesh AB. http://www.enmesh.se (2010-05-06). Retrieved 2010-05-06.

[2] Bærentzen Jacob Andreas and Aanæs Henrik. Computing discrete signed distance fields from triangle meshes. Technical report, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, 2002.

[3] Erem Burak and Dedual Nicolas. Surface construction analysis using marching cubes. http://www.ndedual.com/wp-content/uploads/marchingcubes.pdf. Retrieved 2010-05-21.

[4] Forest Clement, Delingette Herve, and Ayache Nicholas. Removing tetrahedra from a manifold mesh. In *Computer Animation*, page 225, 2002.

[5] de Bruin P. W., Vos F. M., Post F. H., Frisken-Gibson S. F., and Vossepoel A. M. Improving triangle mesh quality with surfacenets. In *Lecture Notes in Computer Science*, pages 69–102. Springer Berlin / Heidelberg, 2000.

[6] Lingrand Diane, Charnoz Arnaud, Gervaise Raphaël, and Richard Karen. The marching cubes. http://users.polytech.unice.fr/∼lingrand/MarchingCubes/algo.html (2002-05-31). Retrieved 2010-06-01.

[7] Lorensen William E. Marching cubes. http://www.marchingcubes.org/index.php/Marching_Cubes (2007-12-01). Retrieved 2010-06-08.

[8] Lorensen William E. and Cline Harvey E. Marching cubes: a high resolution 3d surface construction algorithm. *ACM SIGGRAPH Computer Graphics*, 21(4):163–169, 1987.

[9] Frisken Sarah F., Perry Ronald N., Rockwood Alyn P., and Jones Thouis R. Adaptively sampled distance fields: A general representation of shape for computer graphics. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 249–254, 2000.

[10] Chien-Chang Ho, Fu-Che Wu, Bing-Yu Chen, Yung-Yu Chuang, and Ming Ouhyoung. Cubical marching squares: Adaptive feature preserving surface extraction from volume data. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2005)*, 24(3):537–545, 2005.

[11] O'Rourke Joseph. *Computational Geometry*. Cambridge University Press, Cambridge, CB2 2RU, UK, second edition, 1998.

[12] Manson Josiah and Schaefer Scott. Isosurfaces over simplicial partitions of multiresolution grids. *Computer Graphics Forum (Proceedings of Eurographics)*, 29(2):377–385, 2010.

[13] Bloomenthal Jules. Implicit surfaces. http://www.unchainedgeometry.com/jbloom/pdf/impencyc.pdf (2010-05-13). Retrieved 2010-05-13.

[14] KTH. Finite elements fall 07. http://www.csc.kth.se/utbildning/kth/kurser/DN2260/fem07/07_project/femproj07.pdf (2007-08-31). Retrieved 2010-05-13.

[15] Logan Daryl L. *A First Course in the Finite Element Method*. Brooks/Cole, Pacific Grove, CA 93950 USA, third edition, 2002.

[16] Lee John M. *Introduction to Topological Manifolds*. Springer-Verlag, New York, US, 2000.

[17] Levoy Marc. The digital michelangelo project. http://www.graphics.stanford.edu/projects/mich/ (2009-08-11). Retrieved 2010-06-04.

[18] Kazhdan Michael, Klein Allison, Dalal Ketan, and Hoppe Hugues. Unconstrained isosurface extraction on arbitrary octrees. In *Proceedings of the fifth Eurographics symposium on Geometry processing*, pages 125–133, 2007.

[19] Reddy J. N. *An Introduction to the Finite Element Method*. McGraw-Hill, 1221 Avenue of the Americas, New York, NY 10020, third edition, 2006.

[20] CFD Online. Mesh adaptation. http://www.cfd-online.com/Wiki/Mesh_adaptation (2007-05-15). Retrieved 2010-05-13.

[21] Bhaniramka Praveen, Wenger Rephael, and Crawfis Roger. Isosurfacing in higher dimensions. In *Proceedings of the conference on Visualization*, pages 267–273, 2000.

[22] Bhaniramka Praveen, Wenger Rephael, and Crawfis Roger. Isosurface construction in any dimension using convex hulls. *IEEE Transactions on Visualization and Computer Graphics*, pages 130–141, 2004.

[23] Shu Renben, Zhou Chen, and Kankanhalli Mohan S. Adaptive marching cubes. *The Visual Computer*, 11(4):202–217, 1995.

[24] Newman Timothy S. and Yi Hong. A survey of the marching cubes algorithm. *Computers & Graphics*, 30:854–879, 2006.

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

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

[27] Oren Sifri, Sheffer Alla, and Gotsman Craig. Geodesic-based surface remeshing. In *12th International Meshing Roundtable*, pages 189–199, 2003.

[28] Raman Sundaresan and Wenger Rephael. Quality isosurface mesh generation using an extended marching cubes lookup table. *Computer Graphics Forum*, 27(3):791–798, 2008.

[29] Henrik Rydberg (supervisor at Enmesh AB); private correspondence.

[30] Ju Tao, Schaefer Scott, and Warren Joe. Convex contouring of volumetric data. *The Visual Computer*, 19(7-8):513–525, 2003.

[31] Karkanis Tasso and Stewart A. James. Curvature-dependent triangulation of implicit surfaces. *IEEE Computer Graphics and Applications*, 21(2):60–69, 2001.

[32] Lewiner Thomas, Lopes Hélio, Wilson Vieira Antônio, and Tavares Geovan. Efficient implementation of marching cubes' cases with topological guarantees. *journal of graphics, gpu, and game tools*, 8(2):1–15, 2003.

[33] Weisstein Eric W. Gradient. http://mathworld.wolfram.com/Gradient.html (2010-04-30). Retrieved 2010-05-21.

[34] Wikipedia. Computer-aided design. http://en.wikipedia.org/wiki/Computer-aided_design (2010-05-13). Retrieved 2010-05-13.

[35] Wikipedia. Gradient. http://en.wikipedia.org/wiki/Gradient (2010-05-21). Retrieved 2010-05-21.

[36] Wikipedia. Homeomorphism. http://en.wikipedia.org/wiki/Homeomorphism (2010-05-03). Retrieved 2010-05-13.

[37] Wikipedia. Implicit and explicit functions. http://en.wikipedia.org/wiki/Implicit_and_explicit_functions (2010-05-13). Retrieved 2010-05-13.

9 Appendix A: Tools Developed

It was realized from the beginning that much time would be devoted to tracking down the causes of errors in the produced surfaces. To help with this, two applications were developed during the course of the project. They will briefly be described in this appendix along with some screenshots.

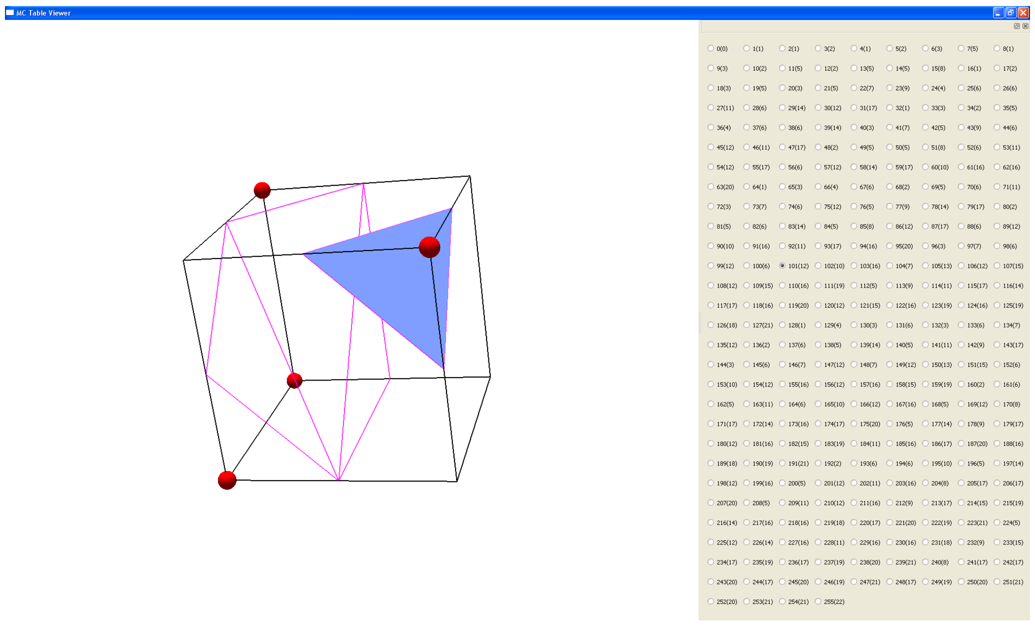
9.1 MC Table Viewer

Even if test code was developed for making sure that the transform methods for calculating the Marching Cubes tables from the base cases were working as intended, being able to evaluate the results visually was considered valuable. For this purpose, an application where each of the 256 entries could be viewed independently was created. In parenthesis after each entry, the base case from which that entry was derived is printed. This makes it easy to go through all entries derived from a certain base entry to make sure it remains correct through the transformations. The triangle faces are rendered in two passes, where the first pass fills the faces with light-blue color, and the second pass renders the magenta outlines. Backface culling is turned off for the outlines and turned on for the filled triangles. This makes it possible to always see the triangles (if they are not depth culled), and to distinguish which are front facing and which are back facing. This was a helpful feature, since there sometimes were problems with triangles having the wrong winding order.

9.2 ContourViewer

This purpose of this application was to be able to view the octree, the dual grid, and the extracted isosurface simultaneously. During development, many of the surface errors were due to some error in the construction of the dual grid. It was then useful to be able to see how the dual grid was connected in relation to the octree.

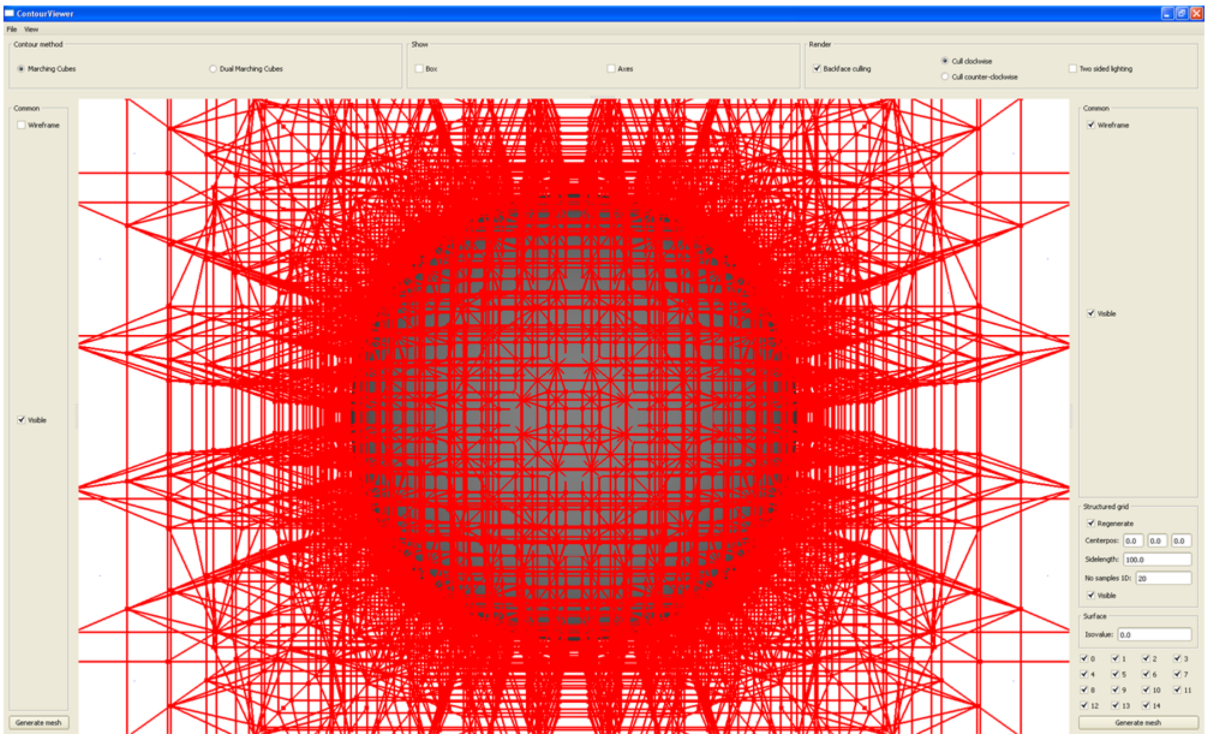
For tracking down errors, small octrees however had to be used. Only after a few subdivisions of the octree, it became hard to see how the dual grid connected the voxels. Some sort of picking would have been useful, so that only certain voxels/dual grid cells of interest could be selected, for example at regions where the surface exhibited erroneous features.



MC Table Viewer

0(0)	1(1)	2(1)	3(2)	4(1)	5(2)	6(3)	7(5)	8(1)
9(3)	10(2)	11(5)	12(2)	13(5)	14(5)	15(8)	16(1)	17(2)
18(3)	19(5)	20(3)	21(5)	22(7)	23(9)	24(4)	25(6)	26(6)
27(11)	28(6)	29(14)	30(12)	31(17)	32(1)	33(3)	34(2)	35(5)
36(4)	37(6)	38(6)	39(14)	40(3)	41(7)	42(5)	43(9)	44(6)
45(12)	46(11)	47(17)	48(2)	49(5)	50(5)	51(8)	52(6)	53(11)
54(12)	55(17)	56(6)	57(12)	58(14)	59(17)	60(10)	61(16)	62(16)
63(20)	64(1)	65(3)	66(4)	67(6)	68(2)	69(5)	70(6)	71(11)
72(3)	73(7)	74(6)	75(12)	76(5)	77(9)	78(14)	79(17)	80(2)
81(5)	82(6)	83(14)	84(5)	85(8)	86(12)	87(17)	88(6)	89(12)
90(10)	91(16)	92(11)	93(17)	94(16)	95(20)	96(3)	97(7)	98(6)
99(12)	100(6)	[✓ 101(12)]	102(10)	103(16)	104(7)	105(13)	106(12)	107(15)
108(12)	109(15)	110(16)	111(19)	112(5)	113(9)	114(11)	115(17)	116(14)
117(17)	118(16)	119(20)	120(12)	121(15)	122(16)	123(19)	124(16)	125(19)
126(18)	127(21)	128(1)	129(4)	130(3)	131(6)	132(3)	133(6)	134(7)
135(12)	136(2)	137(6)	138(5)	139(14)	140(5)	141(11)	142(9)	143(17)
144(3)	145(6)	146(7)	147(12)	148(7)	149(12)	150(13)	151(15)	152(6)
153(10)	154(12)	155(16)	156(12)	157(16)	158(15)	159(19)	160(2)	161(6)
162(5)	163(11)	164(6)	165(10)	166(12)	167(16)	168(5)	169(12)	170(8)
171(17)	172(14)	173(16)	174(17)	175(20)	176(5)	177(14)	178(9)	179(17)
180(12)	181(16)	182(15)	183(19)	184(11)	185(16)	186(17)	187(20)	188(16)
189(18)	190(19)	191(21)	192(2)	193(6)	194(6)	195(10)	196(5)	197(14)
198(12)	199(16)	200(5)	201(12)	202(11)	203(16)	204(8)	205(17)	206(17)
207(20)	208(5)	209(11)	210(12)	211(16)	212(9)	213(17)	214(15)	215(19)
216(14)	217(16)	218(16)	219(18)	220(17)	221(20)	222(19)	223(21)	224(5)
225(12)	226(14)	227(16)	228(11)	229(16)	230(16)	231(18)	232(9)	233(15)
234(17)	235(19)	236(17)	237(19)	238(20)	239(21)	240(8)	241(17)	242(17)
243(20)	244(17)	245(20)	246(19)	247(21)	248(17)	249(19)	250(20)	251(21)
252(20)	253(21)	254(21)	255(22)					

Figure 18: A screenshot of the MC Table Viewer application.



ContourViewer

File View

Contour method: [✓ Marching Cubes] Dual Marching Cubes

Show: Box Axes

Render: [✓ Backface culling] [✓ Cull clockwise] Cull counter-clockwise Two sided lighting

Common: Wireframe [✓ Visible]

Common: [✓ Wireframe] [✓ Visible]

Structured grid: [✓ Regenerate]
Centerpos: 0.0 0.0 0.0
Sidelength: 100.0
No samples 1D: 20
[✓ Visible]

Surface
Isovalue: 0.0

[✓ 0] [✓ 1] [✓ 2] [✓ 3] [✓ 4] [✓ 5] [✓ 6] [✓ 7] [✓ 8] [✓ 9] [✓ 10] [✓ 11] [✓ 12] [✓ 13] [✓ 14]

Figure 19: Using ContourViewer for viewing the dual grid and the extracted surface simultaneously.