DIRECT EXTRACTION OF NORMAL MAPS FROM VOLUME DATA

A Thesis

Presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Mark Edward Barry

February 2007

AUTHORIZATION FOR REPRODUCTION OF MASTER'S THESIS

I reserve the reproduction rights of this thesis for a period of seven years from the date of submission. I waive reproduction rights after the time span has expired.

_____

Signature

_____

Date

APPROVAL PAGE

TITLE: Direct Extraction of Normal Maps from Volume Data

AUTHOR: Mark Edward Barry

DATE SUBMITTED: February 2007

Dr. Zoë Wood	
Advisor or Committee Chair	Signature
Dr. Gene Fisher	
Committee Member	Signature
Dr. Chris Buckalew	
Committee Member	Signature

Abstract

Direct Extraction of Normal Maps from Volume Data

by

Mark Edward Barry

Extracting a high-resolution contour surface from volume data yields highly complex meshes often consisting of millions of individual polygons. Such large models are difficult to work with, requiring large amounts of disk space and extensive rendering time. While reducing a mesh's complexity is desirable for performance reasons and often visually acceptable, the fine details are lost. Subtle surface features such as bumps and cracks built with large numbers of fine polygons are replaced with fewer, larger polygons. "Painting" the surface with textures or normal maps is a common method of retaining the small details of a surface while getting away with a simpler underlying mesh.

Extracting a surface from volume data, simplifying the surface, and applying a normal map is a multi-step process. First, the high-resolution mesh must be extracted from the volume data and saved. From that a mesh simplification algorithm is run, producing the low-resolution counterpart. Finally the normal map for each low-resolution polygon is generated, involving a costly nearest-point search problem that scales with mesh complexity.

This thesis describes a method of *directly* extracting a simplified contour surface along with detailed normal maps from volume data in one fast and integrated process. A robust dual contouring algorithm is used for efficiently extracting a

high-quality "crack-free" simplified surface from volume data. As each polygon is generated, the normal map is immediately applied. An underlying octree data structure reduces the search space required for high to low-resolution surface normal mapping. The process yields simplified meshes fitted with normal maps that accurately resemble their complex equivalents.

Contents

Contents vi

List of Tables vii

List of Figures viii

1 Introduction 1

2 Previous Work 4

2.1 Isosurface Extraction From Volume Data 4

2.2 Adaptive Contouring 6

2.3 Dual Contouring 8

2.4 General Mesh Simplification 10

2.5 Retaining Surface Detail With Normal Maps 12

2.6 Texture Mapping Progressive Meshes 13

3 Initial Exploration 15

3.1 Extracting Normal Maps From Implicit Surfaces 15

3.1.1 Extracting The Implicit Surface 16

3.1.2 Applying Normal Maps To The Implicit Surface 17

3.2 Adaptive Contouring of Volume Data With Normal Map Extraction 20

3.2.1 Interpolating Scattered Normals 23

4 Dual Contouring With Normal Map Extraction 25

5 Results 29

6 Conclusion and Future Work 35

Bibliography 38

List of Tables

5.1 Performance data 31

List of Figures

1.1 Contouring a two-dimensional grid of data 2

2.1 Contour surface intersecting a cube 5
2.2 Contour surface approximation by collapsing cubes 8
2.3 Dual meshes 9

3.1 Matrix equation for projecting points onto a triangle 18
3.2 Projecting fine-level contour vertices onto a contour triangle . . . 19
3.3 Normal mapped implicit surface 19
3.4 Adaptively contoured dragon model 21
3.5 Delaunay Triangulation of projected contour vertices 23
3.6 Adaptively contoured dragon model with normal maps 24

5.1 Normal mapped dragon model with 43,850 quads 31
5.2 Normal mapped dragon model with 16,388 quads 32
5.3 Normal mapped dragon model with 558 quads 32
5.4 Normal mapped dragon model with 65 quads 32
5.5 Normal mapped mythical creature model with 10,950 quads . . . 33
5.6 Normal mapped mouse embryo model with 3,035 quads 33
5.7 Normal mapped human head model with 1,406 quads 34

Chapter 1

Introduction

Volume data is a set of sampled data points arranged on a uniform grid in three-dimensional space often used to model an object. A simple way of describing a three-dimensional object is by assigning a binary value to every point in the volume data set. This binary value indicates whether the point is positioned on the *interior* or *exterior* of the object. The object's surface (also known as its *contour*) lies between adjacent data points of differing binary value. In other words, traversing from an exterior point to an interior point implies a passage through the object's surface. By extracting this surface as a set of polygons, the object can be viewed and manipulated on a computer display. In computer graphics, an accurate representation of surfaces is important. For example, the use of a lighting and shading model for rendering requires knowledge of surface location and orientation.

Volumetric data has the benefit that it can describe interior structure. In medical imaging, three-dimensional samplings of human body tissue obtained through magnetic resonance (MR) or computed tomography (CT) scans are captured as volume data. Volume data has also been used in the process of scanning

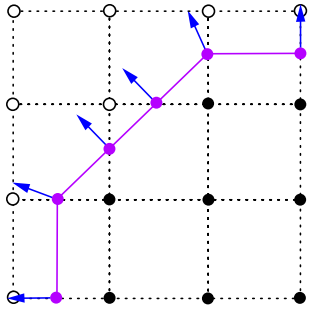

Figure 1.1: Example of contouring a two-dimensional grid of data. The solid dots along the dotted lines represent interior points while the hollow dots represent exterior points. The solid line represents the approximate contour surface. The solid dots along the contour represent the contour vertices.

physical objects to electronically capture their shape [8].

Viewing the object or structure that volume data describes requires extracting a contour surface. Today's graphics rendering pipelines are setup to draw, illuminate, and shade the simplest form of a two-dimensional surface known as a polygon. Typically an extracted contour surface of high detail will contain a very large number of polygons. The interconnection of vertices that forms polygons may also be referred to as a *mesh*. Though a highly detailed contour surface is often desirable, a high cost is incurred in terms of its storage space and rendering performance. This is why it becomes necessary to approximate the original mesh by reducing the extracted surface's size and complexity. Usually a balance is created between the mesh's complexity (or "resolution") and how accurately it approximates the original object. A low-resolution mesh likely will not capture the fine detail but will require less storage space and will render faster.

Still, it is desirable to achieve the benefits of both worlds: fine detail with high performance. A technique known as *normal mapping* is one way to achieve this goal. Normal mapping essentially "paints on" the fine details that are otherwise

not captured on a low-resolution surface.

The goal of this thesis is to present a new method for quickly generating simplified contour surfaces with high-resolution detail using normal maps. The current approach in achieving this solution specifies a lengthy, multi-step process. This thesis demonstrates a novel method of greatly shortcutting the current process by directly extracting a simplified surface along with normal maps in one integrated process.

The process is also very fast - capable of extracting normal mapped meshes of various resolutions in the time of one second on average, giving the user the ability to quickly choose a mesh at the desired level of detail. Yet even normal mapped low-resolution meshes, with 92% fewer polygons, appear nearly identical to their high-resolution equivalents (see Figure 5.2). Furthermore, because of the drastic difference in size, the low-resolution mesh can render almost fourteen times faster - a critical requirement for real-time interaction.

Chapter 2

Previous Work

2.1 Isosurface Extraction From Volume Data

The *Marching Cubes* algorithm [18] is one of the earliest methods of extracting a polygonal surface from discrete volume data. The uniform grid of volume data points is essentially a three-dimensional array of cubes. Each of a cube's eight corners is a point associated with a scalar value. Based on the value of the isosurface being contoured, each of the cube's eight corners is considered either inside or outside of the contour surface being extracted. The algorithm "marches" through each cube and generates polygons for each portion of the contour that intersects that cube (see Figure 2.1). Contour vertices for the isosurface are generated at points where the contour intersects the cube's edges. The polygons generated in each cube are combined to produce the final contour surface mesh. Since volume data is a set of scalar values in three-dimensional space, a three-dimensional gradient can be computed for the volume space. Therefore each contour vertex generated also has a gradient vector associated with it, representing the contour surface's normal at that point. The small arrows shown in

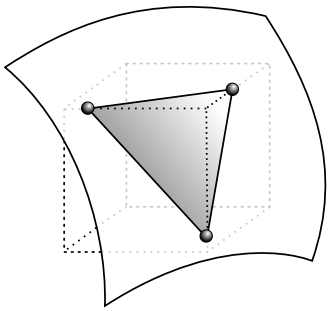

Figure 2.1: Triangle generated where the contour surface intersects a cube. The small spheres represent the contour vertices.

Figure 1.1 and Figure 3.2 represent the contour vertices' normals.

The Marching Cubes algorithm can generate a high-quality mesh that captures a contour's fine details. But often, large numbers of polygons, ranging in the millions, must be generated in order to capture the fine intricacies of a contour. Such large meshes are difficult to work with - requiring substantial disk space, memory, and load time. This complexity more importantly impacts the rendering performance when visually displaying and viewing the mesh on a computer system.

If the finest level of detail is not important to a user, combining groups of eight neighboring cubes into larger cubes will generate a lower resolution mesh. In this way the volume space is partitioned into a fewer number of cubes to be traversed, thus generating a fewer total number of polygons. This idea of combining cubes can be extended to any level of desired detail. The main problem with this general approach is that correct surface topology may not be preserved at all levels of detail.

The *Extended Marching Cubes* algorithm [16] is an improvement over the Marching Cubes algorithm. Instead of restricting the placement of contour vertices to cube edges, extra contour vertices are also generated *within* cubes. This

results in higher-quality meshes that capture features such as sharp edges residing within cubes. The creators of this approach also propose enhancing volume data sets by sampling directed distances (vectors) to a surface rather than scalar distance-to-surface values [11].

Wood et. al. propose a method of extracting semi-regular meshes from volume data [26]. This has the advantage that the semi-regular mesh can be adaptively subdivided according to varying contour surface complexity. This approach also makes it easy to generate meshes at varying resolutions - meshes with few polygons to approximate the rough shape, or meshes with a very high number of polygons to capture the finest details.

In all these surface extraction methods, the downside is they require generating and rendering a high-resolution mesh in order to display the contour surface's finest details. Again, high-resolution meshes require more storage space and take more time to render. This thesis though focuses on a method of extracting a low-resolution mesh while still retaining the finest visual details.

2.2 Adaptive Contouring

A popular method for dealing with geometric models is a multi-resolution approach, where a higher number of polygons may be generated for complex areas of a surface while a fewer number of polygons may be generated in the less complex areas of a surface. For example, accurately capturing a smooth, planar portion would require only a few polygons while a bumpy, jagged portion would require many more. In this way a balance is reached whereby the fine details are preserved while the overall polygon count is minimized. Extracting a contour surface in this manner is known as adaptive polygonization or adaptive

contouring.

A very logical way of organizing and working with a three-dimensional uniform grid of data points is with an *octree* structure [19] [25] [24]. As mentioned previously, groups of eight neighboring cubes (an "octant") can be collapsed together to form a larger cube (see Figure 2.2). These larger cubes can be again collapsed together to form even larger cubes, and so on. The octree as applied to a volume space is structured such that the "root" cube of the tree is the bounding box for the entire volume space. Subdividing the root cube results in eight "child" cubes - forming a parent-child relationship. Each child cube can be recursively subdivided further, all the way to the finest sampling of the volume space ("leaves"). The idea behind adaptive contouring is that an octant may be collapsed into a single cube if the intersecting contour remains accurately preserved. In other words an octant may be collapsed if the surface generated from a collapsed octant accurately approximates the same combination of surfaces generated from the un-collapsed octant. The advantage of collapsing an octant is that it will generate fewer polygons than will eight un-collapsed cubes (see Figure 2.2). This process of collapsing cubes may be performed recursively on all branches of the octree, adapting to the complexity of the contour being extracted. The other advantage to using an octree structure is that empty cubes - those that which the contour surface does not intersect - may be pruned away thus reducing the size of the octree.

The problem with this method of adaptive contouring on simplified octrees is that the resulting polygonal mesh is not "water-tight". The result is instead a visually unacceptable mesh riddled with "cracks". This arises from the discontinuities between polygons generated in cubes at differing levels in the octree. The problem of generating a closed polygonal mesh from a simplified octree has been

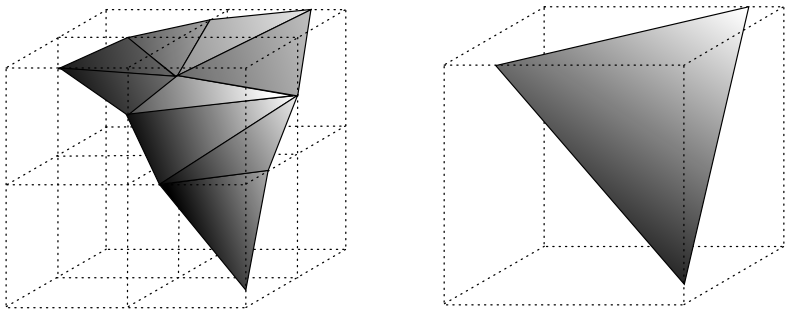

Figure 2.2: Left: a contour surface intersecting eight neighboring cubes (octant), generating nine triangles. Right: the collapsed cube approximation, generating one triangle.

extensively studied [5] [9] [25]. In general the solutions proposed are not perfect or require certain restrictions on how the octree is simplified.

2.3 Dual Contouring

An adaptive contouring method such as Marching Cubes generates contour vertices on cube edges at points where the contour surface intersects. From these vertices, polygons are constructed that lie within cubes. Dual contouring methods instead generate a contour vertex *within* a cube's interior. Polygons are then constructed from neighboring contour vertices, thus spanning cube boundaries. A mesh generated in this manner is considered "dual" to one generated with Marching Cubes in the sense that vertices in one will correspond to polygon faces in the other and vice versa (See Figure 2.3). Dual contouring methods generally produce more accurate meshes because vertices are not constrained to cube edges but instead are free to move within a cube's interior.

Ju et. al. have developed an *adaptive* dual contouring method that produces a "crack-free" contour surface from a simplified octree [15]. Unlike other adaptive

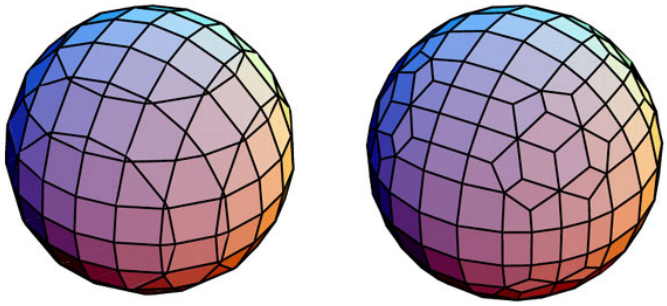

Figure 2.3: Left: mesh generated using Marching Cubes. Right: the left mesh's dual. **(Credit: [15])**

contouring methods, this dual contouring method does not impose restrictions on how an octree is simplified nor requires any sort of crack patching. In addition, it performs as well as [16] in terms of preserving distinct features such as sharp edges.

The first step in the Dual Contouring algorithm is to simplify the underlying octree data structure that represents the volume data. To do this, the authors make use of quadric error functions (QEFs) as introduced in [10]. Associated with each leaf cube in the octree is a QEF that describes both the optimal placement of a contour vertex within the cube and the error quantifying how well that vertex approximates the contour surface. An octant may be collapsed if the resulting sum of QEFs yields an error less than a given tolerance. Contour vertices within leaf cubes are calculated and placed so as to minimize the QEF. The error tolerance may be adjusted to vary the final mesh's level of detail. Using a low error tolerance will yield a higher detail mesh with a higher polygon count. A high error tolerance will yield a lower detail mesh with fewer polygons but, of course, with many of the finer details lost.

The second step, after the octree has been maximally collapsed, is to connect the contour vertices to form polygons that in turn make up the final extracted

surface. Wherever the contour surface intersects a cube edge, the contour vertices from the four cubes sharing that edge are connected to form a *quad*. Because the octree is adaptively simplified there may be cases where a collapsed octant is adjacent to an un-collapsed octant. The polygon spanning these cubes is instead generated as a *triangle*. For this thesis, the fact that a polygon is localized to four known octree cubes is very important in the process of creating a normal map for the polygon. The significance of this is detailed in Chapter 4.

Another feature of the Dual Contouring algorithm is its ability to maintain surface topology through various levels of surface detail. A strict approach will disallow any cube collapses that result in a topology change. Alternatively a topology change can be tagged with a high error value and may still be performed if the error tolerance is set high enough. Usually this approach is more practical since preserving surface topology is not always necessary for an adequate approximation.

Though the Dual Contouring algorithm as presented in [15] is very robust and in general generates high-quality meshes, fixes and improvements are discussed in [27], [23], and [14].

2.4 General Mesh Simplification

It is important to note that the methods discussed in the previous sections extract a *simplified* surface *directly* from volume data. A more general approach to contour surface extraction usually does not implement any sort of adaptive contouring but instead involves first extracting a high-resolution surface and later simplifying it as necessary. A high-resolution mesh extracted from a large volume data set may contain hundreds of thousands or even millions of polygons. These

large meshes are usually much too large and cumbersome to work with. So it becomes necessary to reduce the mesh's complexity to a manageable size. A good mesh simplification process will reduce a mesh's polygon count while at the same time maintaining the most prominent features and overall shape. Mesh simplification/decimation is another area of computer graphics that has been extensively studied [12] [10] [13].

There are a couple drawbacks to using a general mesh simplification approach in the context of this thesis work. One is that it implies the extra step of first extracting a high-resolution mesh that may end up being discarded anyways. This initial step requires extra processing time and greater storage space. Once the high-resolution mesh is created, extra loading and processing time is necessary to perform the actual simplification procedure on it. Adaptive contouring methods discussed previously have the benefit that they cut out these steps - instead *directly* extracting the final simplified mesh.

Another drawback is that given a high-resolution mesh and its simplified counterpart, there is no straightforward mapping of surface points from one to the other. This is important in that the goal of this thesis is to use a simplified mesh that still retains the same detailed surface normals present in a high-resolution equivalent. In general, given a point's coordinates on one mesh, no simple function exists that can calculate the corresponding coordinates on the other. Usually, vertices on one mesh end up mapping to a polygon's interior on the other. In order to determine point correspondence, a brute-force, nearest-point search is required. The significance of this is discussed further in the following section.

2.5 Retaining Surface Detail With Normal Maps

A normal map [17] [7] "paints" a planar polygon, otherwise of a uniform normal across the entire surface, with varying normals across the surface. Illuminating the surface for rendering reveals not a uniformly shaded surface, but one with varying shading according to the normal map, making it visually convincing that the surface is not flat, but textured with fine-detail bumps. This is a very powerful technique that allows a simplified mesh, painted with normal maps, to appear as detailed as a high-resolution mesh. Memory requirements, load time, and rendering time scale proportionally with a mesh's polygon count. Applying and rendering normal maps, on the other hand, is a very fast and cheap operation. Therefore it's desirable to use as simple meshes as possible. Modern computer games that demand real-time rendering performance make heavy use of simplified meshes textured with normal maps.

In [6], the authors describe a method by which the details from a high-resolution mesh can be mapped to a low-resolution approximation. Data such as texture values or surface normals found on the high-resolution mesh are captured in texture maps (or normal maps) and applied to the low-resolution approximation. In this way, a simplified mesh is reduced in geometric complexity yet still renders the same details found on its high-resolution equivalent. The process first assumes that a high-resolution mesh has been simplified into a low-resolution approximation. For each polygon in the low-resolution mesh, a texture map is created that captures the details retained in the high-resolution mesh. For each texture map sample taken on the low-resolution mesh, a ray is cast out to determine the nearest corresponding point on the high-resolution mesh. This can be a costly operation that involves searching every polygon in the high-resolution

mesh for a possible intersection with the ray. The time required for this search operation thus scales with the high-resolution mesh's polygon count. The authors of [6] do provide some optimizations by partitioning the search space into cells. The nearest-point search is then only limited to the single cell in which a low-resolution polygon resides. Note that creating the spatial partitioning data structure to optimize the search space requires the redundant creation of a volume data structure if the mesh came from a volume.

Several software applications have implemented the process described above. "Melody" developed by nVidia [20] and "NormalMapper" developed by ATI [2] are two such products. These applications load a high-resolution mesh and a low-resolution counterpart, and following the procedure described in [6], generate the appropriate normal maps. One version of Melody even incorporates the mesh simplification process, giving the user the ability to dynamically vary the level of simplification before generating normal maps.

A limited search space is naturally implemented using octrees in the Dual Contouring algorithm. As described in Chapter 4, collecting normals to create a normal map is limited to "searching" only four octree cubes that a polygon spans.

2.6 Texture Mapping Progressive Meshes

The authors of [22] present a method of constructing a progressive mesh such that all meshes in the progressive mesh sequence share a common texture parameterization. In the case of using a common normal map, all meshes at all resolutions retain the same fine details found on the highest-resolution mesh. In much the same way, this thesis work allows the extraction of meshes at vari-

ous resolutions that use normal maps to retain the same fine details found on a high-resolution equivalent. The main difference is that the progressive mesh approach uses a single normal map for all mesh resolutions whereas this thesis work generates a new set of normal maps for each mesh extracted. The single parameterization approach may be more efficient in the end but takes a substantially longer time to construct for all mesh resolutions. The process described in this thesis instead allows one to immediately switch between varying resolutions very quickly, thus avoiding any lengthy initial construction process.

Chapter 3

Initial Exploration

Initial investigation into this project started with the extraction of normal maps from implicit surfaces. This closely related domain provided a good initial test ground for the project. This initial work is presented to highlight the significance of the final implementation.

3.1 Extracting Normal Maps From Implicit Surfaces

An implicit surface [4] is a two-dimensional surface that exists in three-dimensional space. It is the set of all points in three-dimensional space that satisfy the equation $f(x, y, z) = c$ where c is a constant. Mentioning implicit surfaces is important because the contour surface extracted from a volume data set is akin to the surface calculated from an implicit equation. As mentioned previously, a volume data set is a set of scalar values aligned on a uniform grid spread out in three-dimensional space. In the same way, an implicit equation

defines scalar values in three-dimensional space. The only difference being that volume data is discrete (values defined only at integer coordinates on a uniform grid) but an implicit equation defines a continuous set of values throughout all three-dimensional space. The process of extracting and polygonalizing an implicit surface [3] can be accomplished in much the same manner as extracting a contour surface from discrete volume data, by using the Marching Cubes algorithm for instance.

Working with an implicit equation versus discrete volume data proves much more convenient because of an implicit equation's continuous nature. In addition to scalar values being defined at every point in space, the equation's *gradient* is also defined for every point in space. A continuous gradient translates into a well-defined surface normal vector at every point on an implicit surface, which proves very convenient for extracting normal maps.

3.1.1 Extracting The Implicit Surface

To extract the implicit surface mesh, the standard Marching Cubes algorithm is used without incorporating any adaptive contouring techniques. First, a section of three-dimensional space is blocked off and subdivided into a uniform array of cubes. For each of the cubes' corner vertices, the three-dimensional x-y-z coordinates are plugged into the implicit equation to calculate its scalar value. Performing Marching Cubes on this three-dimensional array of cubes at the "zero contour" generates the polygonalized implicit surface. To produce a visually accurate model of the implicit surface *without* using normal maps, the three-dimensional space must be divided into smaller cubes so that a finer sampling is possible. From this finer sampling, smaller polygons are generated to capture the

surface's fine detail. But because the final goal is to let the normal maps capture the fine detail rather than the mesh's underlying geometry, the space is subdivide into relatively large cubes. This results in a polygonalized implicit surface that appears rather rough - only approximating the general shape of the surface (see Figure 3.3 center). Importantly, this keeps the surface's total polygon count very reasonable, which translates into better rendering performance.

3.1.2 Applying Normal Maps To The Implicit Surface

To make up for the course appearance of the mesh alone, normal maps are applied to each polygon in the mesh. The normal maps are meant to capture the finer details that a few, large polygons cannot. As the Marching Cubes algorithm generates polygons, the appropriate normal map is immediately applied.

The simplest building block of a polygon is a triangle. Because the mesh data structure implemented here is based on triangles, it makes sense to generate a normal map per triangle. After Marching Cubes generates a triangle, a number of steps occur to generate the normal map for that triangle. First the dimensions of a rectangular normal map must be calculated based on the size of the triangle. Because the triangle is defined in three-dimensional space and the normal map is only of two dimensions, the triangle must be projected onto a two-dimensional space. In simpler terms, a two-dimensional rectangle must be fitted over the triangle.

The triangle's orthonormal basis is computed and used to calculate the projection (see Figure 3.2). The orthonormal basis matrix is composed of three mutually perpendicular row vectors: $\vec{u}$, $\vec{v}$, and $\vec{w}$. One of the vectors must correspond to the triangle's normal vector. The other two form the plane in which

$$\begin{bmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ w_x & w_y & w_z \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} = \begin{bmatrix} p'_x \\ p'_y \\ p'_z \end{bmatrix}$$

Figure 3.1: Matrix equation for projecting points onto a triangle (and its associated normal map). Using orthographic projection, the component p'_z may be ignored on the two-dimensional normal map.

the triangle lies. Choosing to associate the triangle's normal with $\vec{w}$, the vectors $\vec{u}$ and $\vec{v}$ must be computed. There are no constraints on the orientation of $\vec{u}$ nor $\vec{v}$, as long as all three vectors are mutually perpendicular - forming an orthogonal matrix. Although, choosing values for $\vec{u}$ and $\vec{v}$ translates into how the triangle will be oriented on the normal map and determines the normal map's dimensions required to bound the triangle. For most efficient use of space, the $\vec{u}$ vector can be oriented parallel with the triangle's longest edge. The $\vec{v}$ vector can then be computed from $\vec{w} \times \vec{u}$.

The x-y-z coordinate of each of the triangle's vertices is multiplied with the basis matrix to perform the projection. After the projection, the z-component of the triangle's vertices can be dropped and the coordinates treated as two-dimensional. The two-dimensional triangle is bounded with a rectangle representing the normal map. Next the normal map is filled, pixel-by-pixel, with the appropriate normal vector values. These vectors are especially easy to determine due to the fact that the surface is being extracted from an implicit equation. Normals are determined by plugging the three-dimensional x-y-z coordinates associated with each normal map pixel into the equation describing the implicit equation's gradient. Because the gradient is defined continuously everywhere, there is no problem determining

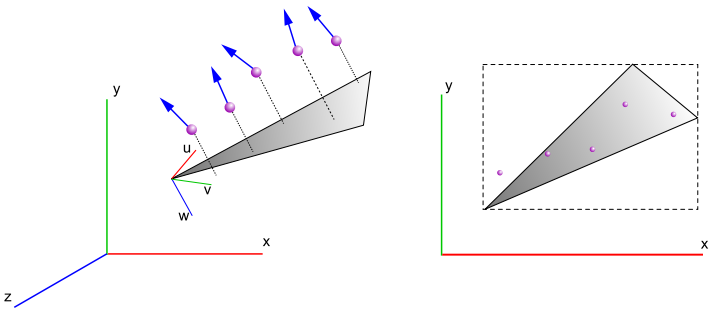

Figure 3.2: Projection of fine-level contour vertices onto a contour triangle. The dotted bounding box on the right represents the normal map.

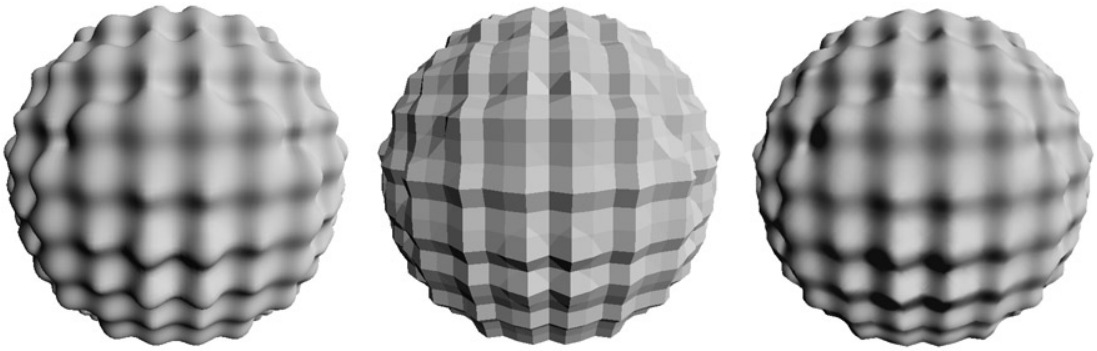

Figure 3.3: Left: high-resolution implicit surface with 138,632 triangles. Center: simplified flat-shaded surface with 8,216 triangles. Right: simplified surface with same geometry as the center surface (8,216 triangles) but with normal maps applied. Implicit equation used: $x^2 + y^2 + z^2 + 0.6\cos(7.4x) + 0.6\cos(7.4y) + 0.6\cos(7.4z) = 20$

normals for all pixels in the normal map.

Though the process of generating normal maps for implicit surfaces works very nicely (see Figure 3.3), the remaining chapters focus on extending this work to a more general domain of discrete volume data.

3.2 Adaptive Contouring of Volume Data With Normal Map Extraction

Successfully extracting a normal mapped implicit surface was an excellent proof of concept that led to the next stage of performing the same procedure but instead on *discrete* volume data. Extracting the polygonalized surface from the volume data first involves creating an octree structure representing the volume space. As mentioned previously, the octree structure is ideal for adaptive contouring. Groups of eight neighboring cubes may be collapsed to form one larger cube, from which a lower polygon count surface may be extracted.

In order to determine if cubes should be collapsed, a certain error metric must be established. An octant that if collapsed would exceed an error threshold will be left alone. An octant will be collapsed as long as the resulting error metric is below a threshold. The error metric used in this implementation is a simple "fine contour vertices to course surface distance" measurement. Certainly other error metrics could be explored but this one served adequately for this experiment. First an octant is "virtually" collapsed and the contour surface intersecting the resulting cube is generated. This is the "course" surface. The Euclidean distance from contour points from all eight cubes in the octant to the course surface is computed. These distances are averaged together and scaled inversely with the cube's size. If the resulting average distance is greater than a threshold, the octant remains un-collapsed and the temporary surface is discarded. Instead, the finer surfaces generated from all eight cubes are retained and added to the final mesh. Otherwise if the averaged distance is less than the error threshold, the course surface is retained and added to the final mesh. This collapsing process essentially "prunes" branches of the octree, resulting in a simplified octree that

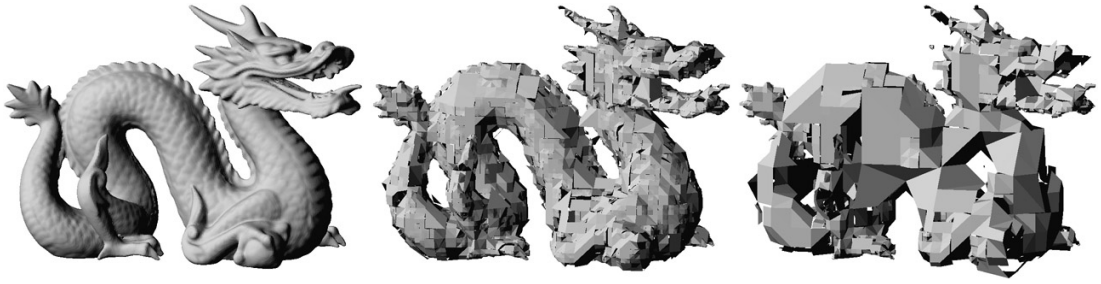

Figure 3.4: Left: high-resolution contour surface with 450,507 triangles extracted using the Marching Cubes algorithm. Center: adaptively contoured surface with 34,343 triangles. Right: adaptively contoured surface with 10,114 triangles.

approximates the volume's contour surface. Using this simple error metric, groups of triangles that are roughly planar are likely to be collapsed into fewer triangles. Thus in areas of high curvature or with distinct features, there are fewer collapsed cubes, resulting in a greater number of smaller triangles. In the dragon model (see Figure 3.4), this is evident around the head and feet, where complexity is high. In the more planar regions, such as on the torso, more cube collapses have occurred, resulting in larger and fewer triangles. This demonstrates the "adaptive" nature of this contouring method.

Though this adaptive contouring method produced very poor meshes, the goal for this step in the project was to demonstrate a proof of concept for the extraction of normal maps from discrete volume data. The process of extracting normal maps from discrete volume data is not too different from that of extracting them from implicit surfaces. The biggest difference though is that the gradient of a *discrete* volume surface is not clearly defined everywhere in the three-dimensional space. To fill each pixel in a normal map with accurate vector values from the contour surface, known normals from contour vertices must be interpolated.

The normal maps act to fill in the fine detail that the simplified contour

surface lacks. The finest detail that can be obtained comes from the contour vertices generated at the finest levels ("leaves") of the octree. Thus the process of generating a normal map involves capturing the normals from these fine-level contour vertices. What makes the use of octrees especially useful in the normal map extraction stage is the fact that all fine-level vertices used to construct a normal map are contained within a known and limited search space. All fine-level vertices needed are contained within the course-level cube from which the course surface is being generated. In other words, all fine-level vertices needed are found within a cube's descendents in the octree. This limited search space is extremely easy and fast to traverse. A cube calls a recursive function for each of its child cubes, traversing down the octree of sub-cubes until reaching a leaf cube. At a leaf cube, contour vertices, if any, are extracted and returned in an array. As all calls to the recursive function return, the arrays of fine-level contour vertices are combined. This combined array of fine-level vertices is the key to creating the normal maps for the current cube's contour surface.

Just as was done for extracting implicit surfaces, a normal map is created for each triangle generated from Marching Cubes. Also following the same process, the triangle generated in three-dimensional space is transformed into two-dimensional space so a normal map can be fitted over it. Just as the triangle is transformed, so is every fine-level vertex. This transformation orthographically projects each fine-level vertex onto the triangle's normal map, with the projected vertices' coordinates rounded to the nearest normal map pixel.

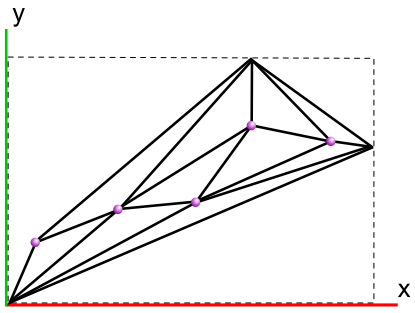

Figure 3.5: Delaunay Triangulation of projected contour vertices from Figure 3.2. The contour triangle and normal map have been translated to the origin. Notice that the Delaunay triangles completely cover the contour triangle, thus defining all normal map pixels used. (Pixels outside the contour triangle are never rendered).

3.2.1 Interpolating Scattered Normals

The projection step scatters "normal-defined" pixels across the normal map but still leaves the remaining pixels' normal values undefined. In order to completely fill the normal map, the normal values at the projected points must be *interpolated* across undefined pixels. The fact that these points are spaced in a random fashion makes interpolation a nontrivial problem. Many methods of interpolating scattered data are available [1] but a *Delaunay Triangulation* approach was initially implemented for this part of the project. The Delaunay Triangulation interconnects a set of scattered points and produces a two-dimensional triangle mesh. It tends to maximize the minimum angle of all the triangles' angles in the triangulation, avoiding any "sliver" triangles. Therefore the Delaunay Triangulation tends to interconnect only the nearest neighboring points. Filling each Delaunay triangle then results in filling in all undefined pixels on the normal map that will be mapped onto the contour triangle. To interpolate normals among points, pixels within each Delaunay triangle are filled according to the Phong shading model [21].

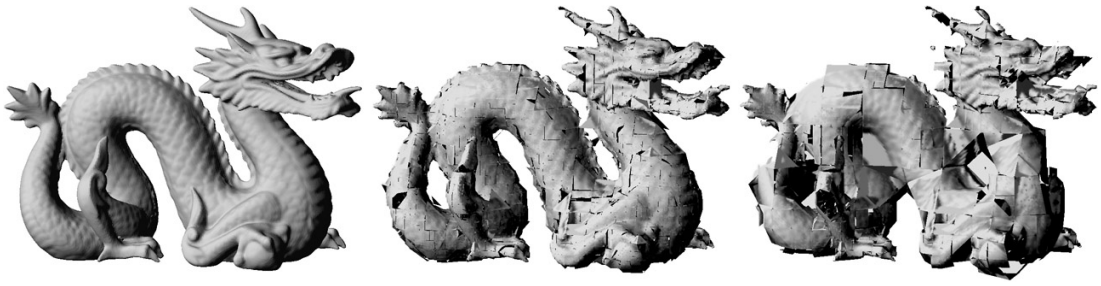

Figure 3.6: Same contour surfaces as in Figure 3.4 but with normal maps applied (center, right).

A Faster Approximation

In short, interpolating projected vertices using the Delaunay Triangulation results in a visually acceptable interpolation - but the process is fairly slow. It turns out that performing the Delaunay Triangulation can be more trouble than it's worth, especially for smaller normal maps where the projected vertices already fill in most the area. Instead of using a robust interpolation method such as the Delaunay Triangulation, a little experimentation led to the discovery of a fast approximation that yields equally good results. This approximation actually does not perform any interpolation at all. It works by first initializing the entire normal map with the contour triangle's normal. Taking no further steps in modifying the normal maps, the resulting normal mapped mesh would appear identical to a flat-shaded mesh without normal maps applied at all. After initialization, the fine-level vertices are projected onto the normal map as before, overwriting any initialized pixels. All normal mapped meshes displayed in the figures use this method.

Chapter 4

Dual Contouring With Normal Map Extraction

Extracting normal maps for adaptively contoured meshes was a success, but a suitable mesh extraction algorithm was still necessary. In addition to being a robust adaptive contouring algorithm, Dual Contouring discussed in Section 2.3 is ideally structured for easy extraction of normal maps as well.

Dual Contouring builds the final mesh by generating quads, and in some cases triangles. The implementation treats triangles as quads having two vertices fused together. Connecting four minimizing vertices found within four cubes in the octree generates a quad. Subsequently, generating the normal map for a quad is trivial. In the adaptive contouring method of Section 2.2, normals for the low-resolution surface were obtained by collecting all the fine-level contour vertices contained within one cube in the octree. The only difference to this approach for Dual Contouring is that the generated polygon (quad) spans *four* cubes. Therefore generating the normal map requires collecting all fine-level contour

vertices from the four cubes involved. A recursive function like that mentioned in Section 3.2 is simply called on all four cubes and the resulting arrays of contour vertices are combined. These fine-level contour vertices are then projected onto the quad's normal map. As before, the remaining undefined pixels in the normal map may be interpolated using the Delaunay Triangulation or filled in with the faster approximation method. It's very clear that the "search" problem discussed in Section 2.4 and Section 2.5 is trivial when using an octree.

The following outlines an executive summary of the final algorithm for extracting normal maps from volume data using Dual Contouring.

1. Convert raw volume data to octree structure. This is a one-time step performed only if needed.
    (a) Prune away "empty" cubes that do not contain any contour vertices (i.e. cubes that the contour surface does not intersect). This one-time step saves storage space and load time.
    (b) Calculate, and store as part of the octree, all contour vertices in the leaf cubes. These are the fine-level contour vertices representing the finest sampling of the contour surface. When a low-resolution quad is extracted, these are collected to generate its normal map.
2. Simplify octree.
    (a) Set error tolerance, determining the extracted mesh's resolution. Low error tolerance = high-resolution mesh. High error tolerance = low-resolution mesh.
        i. Specify either strict or relaxed topology safety checking. Strict topology safety checking will never allow cube collapses that re-

sult in changed topology. Relaxed topology safety checking will allow a topology change if the penalty incurred is below the error tolerance.

(b) Simplify octree according to Dual Contouring's use of the QEF metric. This step generates, and calculates positions for, the contour vertices. In the next step, these contour vertices are connected to form quads.

3. Extract contour surface and normal maps.

(a) Allow the Dual Contouring algorithm to extract quads using the contour vertices generated while simplifying the octree. Add each new quad to the final mesh.

(b) For each quad created, generate its normal map.

i. Calculate the quad's orthonormal basis.

ii. Multiply the coordinates of each quad vertex by the orthonormal basis, thus transforming the quad into two-dimensional space.

iii. Collect all fine-level contour vertices for the quad.

A. For each of the four octree cubes that the quad spans, recursively traverse to the leaves of the octree to collect the finest-level contour vertices.

B. Merge all four collections into one.

iv. Project all fine-level contour vertices onto the quad.

A. Multiply the coordinates of each fine-level contour vertex by the quad's orthonormal basis. Drop the result's z-component to obtain the two-dimensional coordinates.

v. Bound the projected quad with a rectangular normal map.

A. Calculate the height and width of the normal map. Use the same sampling rate used for the volume space, where one unit equals the length of a finest-level cube.

B. Shift the quad, normal map, and all projected contour vertices so that the normal map's x and y coordinates are positive.

vi. Interpolate among the projected contour vertices to fill undefined normal map pixels.

A. Optional: calculate the Delaunay Triangulation of all projected contour vertices. Fill each Delaunay triangle using the Phong shading model.

B. Optional: use the fast approximation discussed in Section 3.2.1

Chapter 5

Results

In computer graphics, visual results are often subjective. One of the measures of success though is how well the low-resolution normal mapped surface resembles the original high-resolution surface without normal maps. Some of the results here are expected, such as a higher polygon count mesh bearing a closer resemblance to the original high-resolution mesh than a lower polygon count mesh would. Another factor to consider is the speed at which a mesh, along with its normal maps, can be extracted from the volume and rendered. Extraction time is a less important number because this operation is presumably not performed as often as rendering. In most common uses, a user will extract a mesh then spend some time viewing it. Each little mesh rotation or translation the user requests requires re-rendering. A user expects the display to be smooth and responsive. The worst-case scenario is extracting the dragon mesh from a 356 x 161 x 251 volume, along with normal maps, at the highest resolution. This operation takes a little over 3 seconds. Extracting simpler meshes (which would be the more frequent request) takes less time - an average of 1 second for the meshes shown in the following figures. Being that these times are so low, it makes it easy for a user

to quickly switch between varying levels of mesh resolution. A user may wish to begin with a low-resolution mesh for fast display and cursory examination. Then as the user desires more geometric accuracy, a higher resolution mesh may be quickly extracted. It is very important to emphasize that a low-resolution mesh is *directly extracted*. The process is *not* a matter of taking a high-resolution mesh and simplifying it.

A low rendering time is one of the main motivations for using normal mapped surfaces at all. Recall that one goal of this thesis work is the ability to view detailed contour surfaces very quickly. Table 5.1 compares the rendering performance of high-resolution meshes to low-resolution, normal mapped, meshes. It is clear that rendering low-resolution meshes is a lot faster, yet they retain a significant amount of detail.

For the dragon example, even on the lowest resolution meshes, the dragon's scales are still clearly defined. The flat-shaded meshes alone do not come near to displaying that kind of detail. Figure 5.1 - Figure 5.7 compare a low-resolution and normal mapped mesh to its high-resolution equivalent. At the lowest resolutions, maintaining correct topology of the original high-resolution mesh begins to fail. But even as the quality of the mesh breaks down, the applied normal maps still pick up the fine details. In the extreme example of Figure 5.4, the flat-shaded mesh alone hardly resembles the dragon model. Distinct features such as the sharp bend in its torso are lost in a blob of a few polygons. But even with the poor underlying mesh geometry, applying normal maps immediately restores most of the finer details. The sharp bend in the dragon's torso, along with its scaly skin, becomes evident again.

Figure 5.5 - Figure 5.7 demonstrate that the normal map extraction process works beautifully for a number of different volume data sets. In particular, figures

	Quad Count			Render Time (ms)		
	Hi-Res	Lo-Res	Reduction	Hi-Res	Lo-Res	Speedup
Figure 5.1	225,467	43,850	80.6%	360	90	4.0
Figure 5.2	225,467	16,388	92.7%	360	26	13.8
Figure 5.3	225,467	558	99.8%	360	1	360
Figure 5.4	225,467	65	99.97%	360	0.3	1200
Figure 5.5	150,823	10,950	92.7%	245	22	11.1
Figure 5.6	64,896	3,035	95.3%	103	6	17.2
Figure 5.7	56,637	1,406	97.5%	91	3	30.3

Table 5.1: Performance data for the examples shown in Figure 5.1 - Figure 5.7.

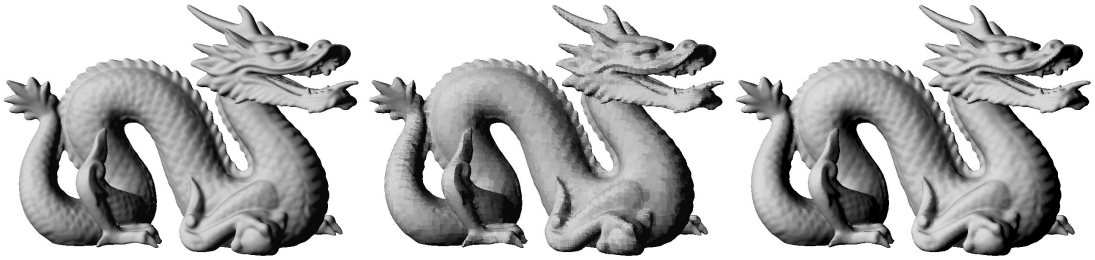

Figure 5.1: A dragon extracted from a 356 x 161 x 251 volume. Left: high-resolution dual contour surface (225,467 quads). Center: low-resolution, flat-shaded, dual contour surface (43,850 quads). Right: same as center but with normal maps applied.

of the mouse embryo and human head demonstrate that the process works equally well for medical imaging applications.

Figure 5.2: A dragon extracted from a 356 x 161 x 251 volume. Left: high-resolution dual contour surface (225,467 quads). Center: low-resolution, flat-shaded, dual contour surface (16,388 quads). Right: same as center but with normal maps applied.

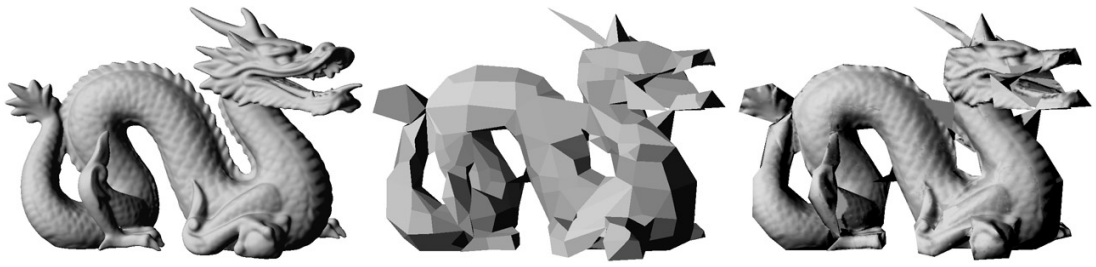

Figure 5.3: A dragon extracted from a 356 x 161 x 251 volume. Left: high-resolution dual contour surface (225,467 quads). Center: low-resolution, flat-shaded, dual contour surface (558 quads). Right: same as center but with normal maps applied.

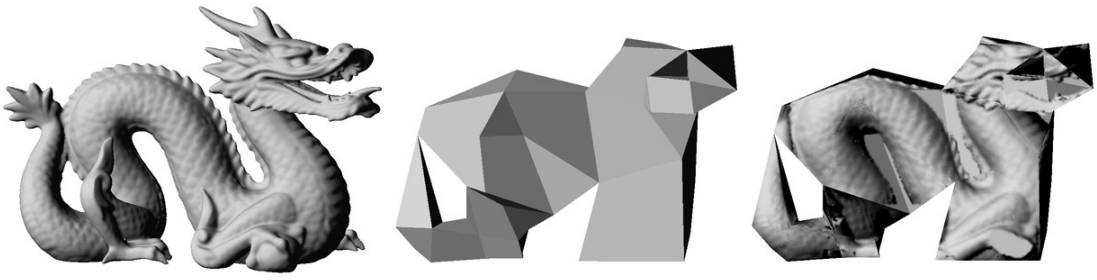

Figure 5.4: A dragon extracted from a 356 x 161 x 251 volume. Left: high-resolution dual contour surface (225,467 quads). Center: low-resolution, flat-shaded, dual contour surface (65 quads). Right: same as center but with normal maps applied.

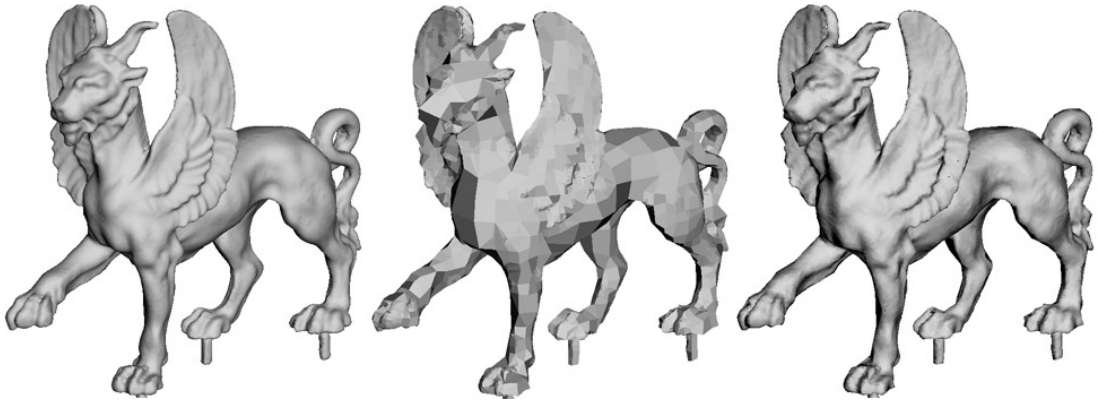

Figure 5.5: A mythical creature extracted from a 316 x 148 x 332 volume. Left: high-resolution dual contour surface (150,823 quads). Center: low-resolution, flat-shaded, dual contour surface (10,950 quads). Right: same as center but with normal maps applied.

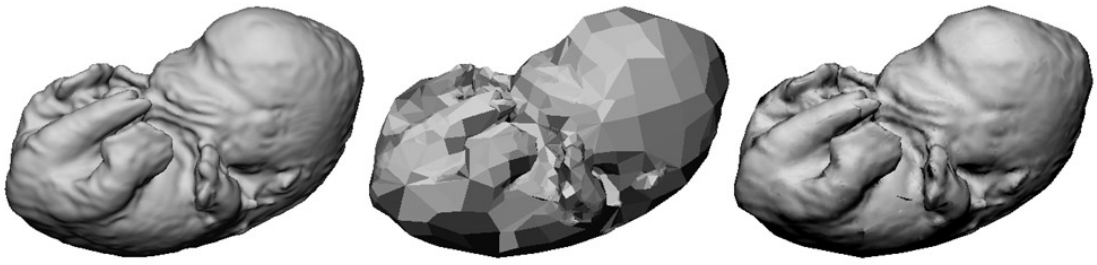

Figure 5.6: A mouse embryo extracted from a 256 x 128 x 128 volume. Left: high-resolution dual contour surface (64,896 quads). Center: low-resolution, flat-shaded, dual contour surface (3,035 quads). Right: same as center but with normal maps applied.

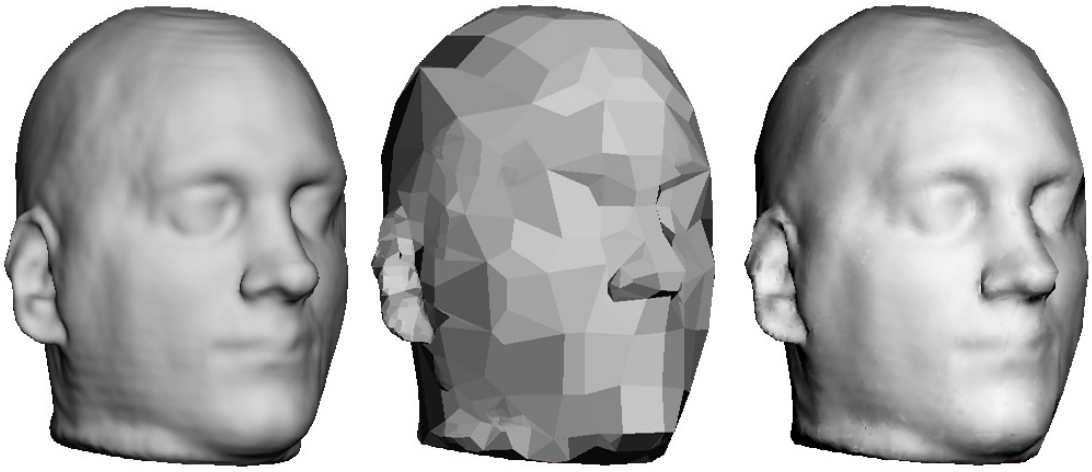

Figure 5.7: A human head extracted from a 130 x 128 x 128 volume. Left: high-resolution dual contour surface (56,637 quads). Center: low-resolution, flat-shaded, dual contour surface (1,406 quads). Right: same as center but with normal maps applied.

Chapter 6

Conclusion and Future Work

This thesis describes a method of directly extracting normal mapped contour surfaces from volume data. This method greatly shortcuts the current multi-step process of extracting a high-resolution mesh, simplifying it, and generating normal maps. Instead, a high-resolution mesh is *never* extracted. A low-resolution mesh is extracted directly from the volume data, with normal maps immediately generated and applied. The process of generating normal maps is greatly streamlined due to the use of an octree data structure and the Dual Contouring algorithm. The visual results of this process are of excellent quality - the low-resolution normal mapped meshes closely resemble their high-resolution equivalents. This process is also very fast - generating and displaying a normal mapped surface in less than a few seconds.

Though this thesis describes a method that is fast and yields excellent visual results, there is always room for improvement. Internally, there are many ways that the program's code and data structures can be optimized. For example, recall the fact that Dual Contouring generates vertices inside octree cubes and that these vertices become quad corners. Every time a normal map is generated

for a quad, the fine-level vertices within four cubes that the quad intersects must be collected. Since more than one quad may intersect a single cube, the process of collecting the fine-levels vertices is performed more than once. This is an inefficiency that may be remedied by memoizing the collection of fine-level vertices within each cube. This would be a trade-off of memory for speed.

One inefficiency in the system as it stands is that each polygon allocates its own normal map. Therefore for thousands of polygons, thousands of separate normal maps are generated. In addition, the space efficiency is very poor. In the case of a rectangular normal map applied to a triangular polygon, the optimal space usage would be 50%. Half the pixels in the normal map don't get mapped onto the triangle. Ideally the collection of normal maps could be packed into one or a few *atlases* [6] [22]. Though the result would be a more efficient use of memory space, the packing process is computationally intensive and would only be worth the time if producing a final output.

Though this thesis work uses Dual Contouring's QEF metric for mesh simplification, the use of other error metrics could be explored. The QEF metric aims to preserve the most prominent features such as sharp edges. Perhaps preserving certain sharp edges would not be necessary if they were instead captured in a normal map. An optimal tradeoff between capturing details in a normal map versus in the underlying mesh could be reached by finding a suitable error metric.

The work presented in this thesis could be extended to use in computer games. Modern computer games already rely on simplified meshes detailed with normal maps to provide both performance and high-quality graphics. The authors of the Dual Contouring algorithm went on to develop a computer game that implements their work. The game environment consists of objects that are destructible - meaning that the underlying geometry of the objects may be modified during

game play. They accomplish this by representing all objects as volumes and performing Constructive Solid Geometry (CSG) operations on them. Immediately after an object's underlying volume has been modified, the Dual Contouring algorithm extracts and displays a new contour surface for that object. A problem as it stands now is that surfaces are not able to display very much detail without bogging down the game. As discussed previously, it takes many fine polygons to represent a surface's intricacies, which in turn will slow down rendering and thus game play. That is unless of course fewer polygons are used and the fine intricacies are captured with normal maps. Referencing the rendering times in Table 5.1, it would be virtually impossible to include any of the high-resolution meshes in a real-time game. But it would certainly be reasonable to include the normal mapped equivalents. As objects in the game are modified via CSG operations, new geometry and new normal maps would be generated. In some cases, depending upon the level of "destruction", existing geometry would not need to be modified at all, but instead only the normal maps.

Bibliography

[1] I. Amidror. Scattered data interpolation methods for electronic imaging systems: a survey. *Journal of Electronic Imaging*, 11:157–176, April 2002.

[2] ATI NormalMapper. http://ati.amd.com/developer/tools.html.

[3] J. Bloomenthal. Polygonization of implicit surfaces. *Comput. Aided Geom. Des.*, 5(4):341–355, 1988.

[4] J. Bloomenthal and B. Wyvill, editors. *Introduction to Implicit Surfaces*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[5] P. Cignoni, F. Ganovelli, C. Montani, and R. Scopigno. Reconstruction of topologically correct and adaptive trilinear isosurfaces. *Computers and Graphics*, 24(3):399–418, 2000.

[6] P. Cignoni, C. Montani, R. Scopigno, and C. Rocchini. A general method for preserving attribute values on simplified meshes. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 59–66, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[7] J. Cohen, M. Olano, and D. Manocha. Appearance-preserving simplification. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer*

graphics and interactive techniques, pages 115–122, New York, NY, USA, 1998. ACM Press.

[8] B. Curless and M. Levoy. A volumetric method for building complex models from range images. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 303–312, New York, NY, USA, 1996. ACM Press.

[9] S. F. Frisken, R. N. Perry, A. P. Rockwood, and T. R. Jones. Adaptively sampled distance fields: a general representation of shape for computer graphics. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 249–254, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[10] M. Garland and P. S. Heckbert. Surface simplification using quadric error metrics. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 209–216, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[11] S. F. F. Gibson. Using distance maps for accurate surface representation in sampled volumes. In *VVS '98: Proceedings of the 1998 IEEE symposium on Volume visualization*, pages 23–30, New York, NY, USA, 1998. ACM Press.

[12] P. S. Heckbert and M. Garland. Survey of polygonal surface simplification algorithms. Technical report, 1997.

[13] H. Hoppe. Progressive meshes. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 99–108, New York, NY, USA, 1996. ACM Press.

[14] IEEE Computer Society. *Intersection-free Contouring on An Octree Grid*. IEEE, 2006.

[15] T. Ju, F. Losasso, S. Schaefer, and J. Warren. Dual contouring of hermite data. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 339–346, New York, NY, USA, 2002. ACM Press.

[16] L. P. Kobbelt, M. Botsch, U. Schwanecke, and H.-P. Seidel. Feature sensitive surface extraction from volume data. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 57–66, New York, NY, USA, 2001. ACM Press.

[17] V. Krishnamurthy and M. Levoy. Fitting smooth surfaces to dense polygon meshes. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 313–324, New York, NY, USA, 1996. ACM Press.

[18] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169, New York, NY, USA, 1987. ACM Press.

[19] D. Meagher. Geometric modeling using octree encoding. In *Computer Graphics and Image Processing*, pages 129–147, 1982.

[20] nVidia Melody. http://developer.nvidia.com/object/melody_home.html.

[21] B. T. Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, 1975.

[22] P. V. Sander, J. Snyder, S. J. Gortler, and H. Hoppe. Texture mapping progressive meshes. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 409–416, New York, NY, USA, 2001. ACM Press.

[23] S. Schaefer, T. Ju, and J. Warren. Manifold dual contouring. *IEEE Transactions on Visualization and Computer Graphics*, 2007. to appear.

[24] R. Shekhar, E. Fayyad, R. Yagel, and J. F. Cornhill. Octree-based decimation of marching cubes surfaces. In *VIS '96: Proceedings of the 7th conference on Visualization '96*, pages 335–ff., Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[25] J. Wilhelms and A. V. Gelder. Octrees for faster isosurface generation. *ACM Trans. Graph.*, 11(3):201–227, 1992.

[26] Z. J. Wood, P. Schröder, D. Breen, and M. Desbrun. Semi-regular mesh extraction from volumes. In *VIS '00: Proceedings of the conference on Visualization '00*, pages 275–282, Los Alamitos, CA, USA, 2000. IEEE Computer Society Press.

[27] N. Zhang, W. Hong, and A. Kaufman. Dual contouring with topology-preserving simplification using enhanced cell representation. In *VIS '04: Proceedings of the conference on Visualization '04*, pages 505–512, Washington, DC, USA, 2004. IEEE Computer Society.