APPROVAL SHEET

Title of Thesis: Real-Time High Quality Volume Isosurface Rendering

Name of Candidate: John Werner Kloetzli, Jr
Master of Science, 2008

Thesis and Abstract Approved: __________________________
Marc Olano
Associate Professor
Department of Computer Science and
Electrical Engineering

Date Approved: ________________________________

ABSTRACT

Title of Thesis: Real-Time High Quality Volume Isosurface Rendering

John Werner Kloetzli, Jr , Master of Science, 2008

Thesis directed by: Marc Olano, Associate Professor
Department of Computer Science and
Electrical Engineering

We present a volume format which is capable of representing complex reconstructions as well as a fast isosurface rendering algorithm. Our volume format consists of a set of densely packing tetrahedral polynomials in Bernstein form, each constrained to provide continuity across face boundaries. We define a specific class of tetrahedral partition suitable for our method, along with a least-squares approximation method to generate data in this format as approximations to arbitrary continuous volumes. Our volume format is closed under convolution with scalar volumes, so by representing reconstruction filters in our format we can create convolved volumes easily. We also present a fast rendering algorithm which maintains interactive rates for most volumes up to 128^3 resolution. Our results include analytic and observational error for how our format approximate several common volume reconstructions, as well as space requirements and rendering speed.

Real-Time High Quality Volume Isosurface Rendering

by

John Werner Kloetzli, Jr

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Master of Science
2008

ACKNOWLEDGMENTS

I want to thank Dr. Olano for being a great academic guide and advisor, as well as the entire VANGOGH lab for the technical help and comradeship. I am also thankful to my thesis committee for the time they have given to me, and the entire CS department at UMBC for the high-quality undergraduate and graduate education that they have provided. Most of all I am thankful to God for the opportunities He has given me in the field of Computer Graphics, as well as to my family for supporting me through the stressful times of graduate school.

Volume data sets were provided by The Volume Library (`http://www9.informatik.uni-erlangen.de/External/vollib/`) and VolVis (`http://www.volvis.org/`).

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

1.1 Volume Rendering

Volume rendering is the process of converting volumetric data into meaningful 2D images with the goal of conveying specific information about the data to the viewer. Applications of volume rendering cover a wide range of disciplines including hurricane visualization, medical diagnosis and planning, and smoke and particle simulations. Volume rendering is differentiated from other areas of computer graphics and visualization by the fact that the actual primitive to be rendered is 3D. Computer graphics traditionally deals with 2D surfaces, usually triangular meshes, embedded in 3D space, but volume rendering truly deals with rendering 3D data.

The purpose of volume rendering is to produce images or video of the data which help someone to understand it better. This is contrasted with computer graphics, where generally the task is to appear realistic or artistic but not to convey specific information. Unfortunately, it is very difficult, if not impossible, to generate a quantitative measure for how well information is conveyed which works across many application domains. This is because different applications have very different types of information which it is important to convey, and a rendering method which satisfies the requirements of one application may fail to satisfy the requirements of another. For this reason, it is important to provide rendering techniques which are flexible enough to apply to different applications.

Within volume rendering there are many different data types and formats. In the physical world, data is a continuous 3D distribution (e.g. a human body) but, in order to efficiently deal with such data on a computer, it must be descritized into a finite set of samples. Although there are many different ways of sampling continuous data, the most common for medical volume acquisition is the regular 3D grid, which stores samples along regular intervals in each dimension. Although some applications of volume rendering require complex sample elements such as vectors or tensors, medical volumes generally have samples which are scalar real numbers. Scalar volume data is particularly useful because it can be used to naturally represent volumes of electron density or the resonant response of hydrogen atoms to RF pulses, which are exactly the values produced by Computed Tomography (CT) and Magnetic Resonance Imaging (MRI) scans, respectively. Many scientific simulations also produce volume data sets of density or electron density, which would also benefit from effective visualization. Because of the prevalence of data which is naturally available in the scalar-valued regularly sampled 3D grid, it is reasonable to focus on generating high-quality renderings of this specific format in real-time.

1.2 Reconstruction Filtering

All volume rendering methods have to perform *reconstruction filtering* to produce a continuous function of the discrete sample points. A large body of research deals with filters for reconstruction of continuous volume data, and the level of quality attained by this step plays a huge role in the efficiency with which the resulting image will perform its task of conveying information to the user. This process is generally expensive, however, and most real-time volume rendering methods perform the simplest possible reconstruction filtering: linear blending in each dimension (tri-linear blending).

Tri-linear filtering is almost ubiquitous throughout real-time graphics because of its speed, but the quality disadvantages of such poor filtering are significant. Figure 1.1 shows a comparison of linear and Gaussian reconstruction filtering of an image, pointing out the artifacts of linear reconstruction. Most real-time volume rendering methods not only use tri-linear reconstruction, but require it in order for the method to work. This tight coupling between the reconstruction filter and the rendering method is an unfortunate one, and is difficult to break and maintain high performance.

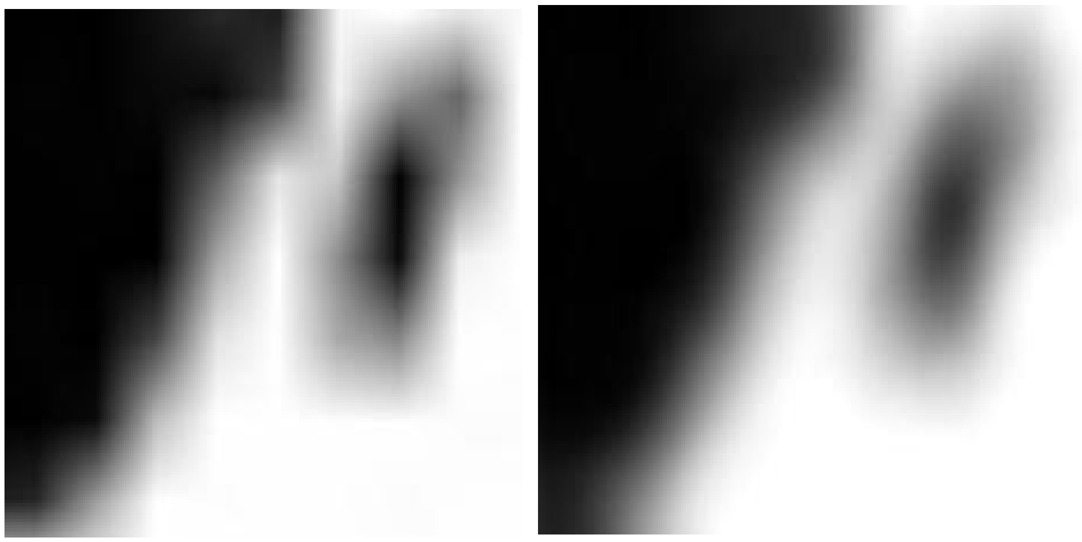

FIG. 1.1. Two reconstructions of a small texture. Bilinear on the left, an approximation of a 3x3 Gaussian with standard deviation .5 on the right. Note the high-frequency linear artifacts on the left, which may hide important information by distracting the user.

The poor quality of linear filtering was demonstrated by Marschner & Lobb (1994), who evaluated a large number of reconstruction filters in the context of volume rendering. They presented evaluation criteria for filters specifically designed for volume rendering, and measured how well several common filters performed in their tests. In their results they mention that linear reconstruction is the least expensive, but also that it is poor quality, and recommend cubic polynomial filters for good quality and windowed sinc filters for high quality. In the end their recommendation was to choose a filter for a specific application based upon the requirements of that application, since no single filter works best in all situations. For this reason it is desirable for our rendering method to support a wide variety of reconstruction filters in order to work well within a large set of applications.

1.3 Isosurface Rendering

Finally, we have to decide the properties of the actual images which the rendering will produce. Our application domain of medical imaging helps us to determine that we want *isosurface rendering*, which is the display of the locus of points of the volume (under reconstruction filtering) which are equal to a target value. For example, consider a continuously reconstructed volume of the density of the human body. If we could identify and display as opaque only the points in that 3D function which had the same density value as bone, we would have an effective visualization of a skeleton. If we were to then change this isovalue density to the density of muscle, our rendering would depict the muscular and skeletal systems of the body. Every density value which is set as the target will produce a continuous surface around all areas in the volume which have at least that high density. This is similar to the way that a contour map in 2D draws a line around all heights greater than the value of the contour. See Figure 1.2 for an example of isosurface rendering.

1.4 Graphics Hardware

The tremendous computational power of *graphics hardware*, which is a special class of computer hardware designed to accelerate rendering tasks, is what allows isosurface rendering to be possible in real-time at all. The amount of data required to be processed in any volume rendering application is tremendous, far outpacing the memory bandwidth available to the traditional CPU. Even if that amount of data could be passed through the

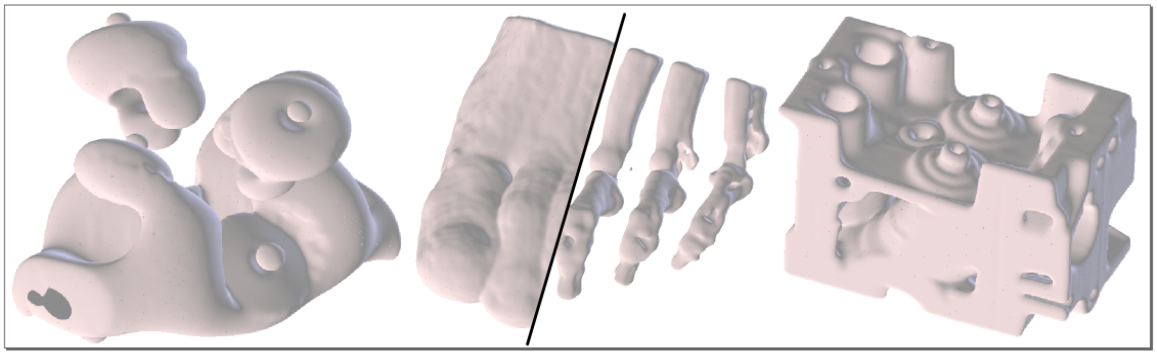

FIG. 1.2. Three isosurface renderings of (from left to right) a molecular simulation, a 3D scan of a foot, and a 3D scan of a mechanical part. The foot (middle image) shows two different isosurfaces corresponding to the density of skin and the density of bone.

CPU, it could not possibly finish the massive amount of computation required to perform rendering in real-time. It would be possible to execute volume rendering on a cluster of CPUs, but, because of low demand, such clusters are expensive and difficult to maintain.

Graphics hardware does not have any of these disadvantages. In order to avoid the memory bandwidth problems, graphics hardware has an internal memory store large enough to hold the problem data set which typically has an order of magnitude greater bandwidth than the CPU. Graphics hardware also has literally hundreds of individual processors which run in parallel, resulting in several teraflops of processing power for current models. Even all of this would not be enough to solve the rendering problem efficiently, however, so the real power of graphics hardware is to execute *latency hiding.* Latency is the time required by the memory system to fetch a piece of data from the memory store and pass it to the processor. The GPU architecture is designed to hold literally tens of thousands of independent computing tasks all executing at the same time, so when one task needs to fetch memory there is a very high chance that another task is ready to execute while that memory is being fetched, making memory access virtually free (assuming there is enough computing work to hide the latency of the memory system). Finally, because of high demand from the video game market, these processors are inexpensive, usually costing a few hundred dollars, and fit into standard computers without requiring special software or maintenance. Any isosurface rendering technique will have to be able to run on graphics hardware in order to have the possibility of real-time performance.

1.5 Preconvolution

In Section 1.2 we showed that no single reconstruction filter works for all applications of isosurface rendering, and therefore a rendering method for use across many application domains will have to support a wide variety of filters. However, developing a single rendering algorithm to support many or all of these different filters is a difficult problem. The reason for this is that convolution is very computationally expensive, and not reasonable to compute on-the-fly for large and/or complex filters. All current real-time isosurface rendering methods use filters of the smallest possible size, and many of them require poor-quality linear filtering.

It seems that what we want in this situation is to separate the convolution step, which is very expensive, from the actual rendering step. If we had some intermediate format to store the volume between these two stages, we could perform the convolution step as a pre-process and only deal with the intermediate form during the rendering step. We refer to this process of storing the convolved data set in an intermediate format as *preconvolution*, since the process of generating the intermediate, or *preconvolved,* form of the data can be performed before the actual rendering. In addition, we should be able to support multiple reconstruction filters with the same rendering technique, as long as the preconvolved format is capable of representing the volume data under different reconstructions. If we are able to render the preconvolved format quickly, then we can get a large performance benefit from preconvolution as well.

In order for a particular preconvolved format to be useful, it will have to satisfy a range of different criteria. First, we will have to render the preconvolved data quickly, so it will need to be designed to work efficiently with graphics hardware. Second, it should be possible to generate the preconvolved data as the result of convolution with a wide range of filters. This would give us the ability to support multiple applications which prefer different reconstruction filtering without changing how the rendering algorithm works. If we can find a preconvolved format which provides this level of speed and flexibility, it would greatly extend the usefulness of the core rendering algorithm for it.

1.6 A Generalized Preconvolved Volume Format

In this thesis we present a preconvolved volume format, called the *BT volume*, which is easy to generate through convolution, fast to render using graphics hardware, and able to represent volumes reconstructed with many different filters. The BT volume format is a 3D superspline, which means that it is composed of many adjacent 3D splines primitives which have been constrained in order to provide continuity across the whole volume. The spline primitive we use is the cubic Bézier tetrahedron, which is a 3D spline with 20 weights defined within a bounding tetrahedron. We found that the cubic spline gave the best balance between computational efficiently and representative power, although all of our work could be generalized up to fourth-order splines. BT volume data sets exist in the same cuboid space of the scalar volume data that they represent, dividing each unit cube of this domain into a number of tetrahedra which exactly partition it. Every pair of adjacent Bézier tetrahedra in the volume are constrained to have a level of continuity in order to prevent cracks.

We will present several algorithms for creating and rendering BT volume data. First, we will show how to create a BT volume representation of an arbitrary data set using a least-squares method. Although this is useful for approximating volumes directly, we would prefer to use the convolution framework, so next we show that if we have a reconstruction filter already approximated as a BT volume, then convolution of that filter with a scalar volume data set produces a BT volume. It is important to note that this process is not an approximation, but exact convolution. This fact, when coupled with the least-squares direct fitting method, give us a powerful generalized framework for generating BT volumes as the result of convolution - first approximate a reconstruction filter with a BT volume using least-squares, then perform convolution with a volume data set to produce the final BT volume for rendering. We will present several common reconstruction filters for volume data and their BT volume approximations, evaluating the quality of approximation for each. Finally, we will describe an optimized rendering algorithm which can render a complete volume at around ten frames a second.

Chapter 2

RELATED WORK

Many isosurface rendering techniques have been developed to cover a wide range of applications. Making images from 3D data is a fundamental problem for computer graphics applied to scientific or medical visualization. All of the different techniques can be classified into two categories based upon the speed with which they operate; Offline methods are too slow for the user to directly interact with, while interactive techniques are designed to generate more than three or four frames a second to enable user interaction.

Offline rendering techniques have been around longer than interactive techniques, mainly because it has only been in the last ten years that computer hardware has been fast enough for any type of volume rendering. They can be divided into two main categories according to the high-level description of the algorithm: Techniques which use *ray-tracing* (described in Section 2.1) follow a single viewing ray through the volume to determine if and where it intersects the isosurface, while *surface construction* methods (Section 2.3) focus on extracting a single isosurface in a 2D representation which can then be rendered using standard techniques. The most important of these methods, called *Marching Cubes* (Lorensen & Cline, 1987), is still one of the most popular isosurface rendering methods used today.

As computers have gotten faster and special-purpose graphics hardware has become more popular, volume rendering methods have been developed which take advantage of this speed. Most interactive isosurface rendering methods can be viewed as acceleration of ray-tracing (Section 2.2) or marching cubes (Section 2.4) in order to allow user interaction. There is a third category, which we refer to as *hybrid* methods (Section 2.5), which can be viewed as a combination of surface extraction and ray-tracing. The main advantage of surface-based algorithms is speed - an extracted isosurface triangular mesh can be rendered very quickly using standard graphics hardware. Ray-tracing, on the other hand, generally gives better quality results because triangular meshes are not exact representations of the isosurface. Hybrid methods work by extracting a coarse boundary of the isosurface (like surface extraction), but then perform per-pixel refinement on each primitive of this coarse representation to produce high-quality results (like ray-tracing). This can be viewed as dividing the rendering process into two phases: a coarse surface extraction phase which generates the mesh, and a high-quality refinement phase, generally some form of ray-intersection for each pixel, which generates the final image.

2.1 Ray-Tracing

One of the first methods which rendered surfaces from volumetric data directly was developed by Levoy (1988), who described a method for computing opacity at each voxel depending on the density values at the voxel and how close they were to the target density. He then accumulated the opacity values along each viewing ray to generate the final surface, and computed lighting using a normalized gradient as the normal vector. Danskin & Hanrahan (1992) presented several hierarchical acceleration methods for volume ray-tracing algorithms, showing that efficient volume ray-tracing requires efficient data structures to allow skipping of empty space. More recently, Sramek & Kaufman (2000) introduced the *distance transform* to allow even more efficient traversal of regular volume data.

2.2 Interactive Ray-Tracing

Many real-time applications of ray-tracing have been developed in the last few years, so only a few of the most important methods will be discussed here. The first interactive isosurface rendering methods were made possible by the advent of supercomputers, and focused on the problem of efficient computation on specific computing systems (Parker *et al.*, 1999a,b). DeMarle *et al.* (2003) extended this method to render very large data sets interactively across a cluster of supercomputers. Their system showed how to organize processing nodes in the system to efficiently balance memory and processing power, and worked with data sets up to several gigabytes.

The bulk of interactive ray-tracing techniques have been designed to work on Graphics Processing Units (GPUs) which are in the general consumer market. Unlike specialized supercomputers, GPUs are mainstream processors which are relatively inexpensive and provide acceleration of standard graphics libraries.

2.3 Surface Extraction

As opposed to ray-tracing, where the volume data is being rendered directly, surface extraction algorithms attempt to find an approximation of a particular isosurface in a form which is easier to render. Even though all the algorithms being presented in this section were developed for offline rendering, performance can still be critical: a rendering time of seconds is greatly preferred to one of hours. In a typical application a single isosurface is rendered many times, so it is worth while to generate a fast intermediate form for the isosurface of interest. Surface extraction methods attempt to do exactly this by extracting the surface in a more compute-friendly format than a full volume render would be. Because triangles are easy to render quickly, most methods use triangular meshes as the intermediate form.

Marching Cubes (Lorensen & Cline, 1987) is the most important algorithm that belongs to this category. It is still very widely used, either directly or in one of its derivative forms, even today. The basic idea of marching cubes is to divide the input volume into discrete cubes between sample positions. By assuming linear reconstruction filtering, each cube which contains a piece of a given isosurface can easily be identified because the sample values at the cube vertices must span the target isosurface value. For each of these cubes independently they create a triangle mesh, but in such a way that they all align correctly at boundaries. The original method precomputed all possible configurations of inside/outside along with triangulations which would stitch all adjacent cubes together without any holes, and, from a single cube which straddled the target density value, began to "march" through the volume, constructing the mesh one cube at a time.

Although Marching Cubes generates the lowest quality isosurfaces of any of the techniques discussed here, it has several advantages which have prolonged its usefulness. First, it is very fast and relatively easy to implement. Second, it produces a triangle representation of the isosurface which can be rendered quickly using traditional methods. There have been many extensions to the basic method, including Marching Tetrahedra (Treece, Prager, & Gee, 1999) and Marching Diamonds (Anderson, Bennett, & Joy, 2005), among others. One interesting modification of marching cubes by Gerstner & Rumpf (1999) extended marching tetrahedra by using a recursive nested tetrahedra volume which allowed them to arbitrarily subdivide the extracted mesh, eventually reaching a smooth reconstruction.

Triangle meshes are the most common isosurface representation format, but others also exist. Theisel (2002) pointed out that, from a filtering perspective, Marching Cubes is very irregular and the triangular meshes it produces introduce severe linear artifacts. To fix this, they generate a mesh of rational cubic Bézier patches which exactly represent the isosurface generated from tri-linear reconstruction.

2.4 Real-Time Surface Extraction

Surface extraction methods which are fast enough to allow user interaction have been developed recently. Acceleration of these methods refers to accelerating the process of extracting the triangle mesh of the isosurface, and not rendering the extracted mesh (which is a trivial problem). Pascucci (2004) presented a method for extracting an isosurface from a tetrahedral mesh by exploiting the transform capabilities of the *vertex shader* on graphics hardware. The vertex shader is a programmable stage in the graphics pipeline which has the capability of transforming vertices which they use to transform a polygonal representation of the isosurface intersection with each tetrahedron into the correct position. Every isosurface intersection with a tetrahedron can be approximated by either one or two triangles without changing the overall topology of the surface. Because the graphics hardware performs all of these steps in parallel this technique is fast. He also presented a nested tetrahedral decomposition to allow refinement by subdividing the tetrahedral mesh before extracting the isosurface, which effectively increased the resolution of the extracted mesh.

Klein, Stegmaier, & Ertl (2004) presented an extension of this work which allowed the resulting tetrahedral mesh to be stored for future frames or post-processing. They achieved this by exploiting the then-new hardware capability of multiple views of data buffers. They render the isosurface vertices into a render buffer as colors, and then re-interpret the buffer as a vertex array. By storing the render buffer they can render a single isosurface many times without recalculating the mesh, and by reading the texture back and extracting the vertex data they can recover the vertex buffer on the CPU for further processing or storage. Reck *et al.* (2004) presented a similar method which used an *interval tree* to speed up isosurface extraction. Because the range of vertex values for each tetrahedron must contain the isosurface value in order for that tetrahedron to participate in the isosurface, they store a tree data structure of these intervals. This allows them to very quickly determine only the set of tetrahedra which participate in the isosurface and then extract the triangle mesh from them. Because participating tetrahedra are usually a very small percentage of the total number of tetrahedra, their method provided large speedups.

This idea was extended by Goetz, Junklewitz, & Domik (2005) to work with cubes instead of the more simple case of tetrahedra. Their contribution was to develop efficient ways of generating the triangular approximation of the isosurface within contributing cubes on graphics hardware, which is a non-trivial problem. This method was later extended (Johansson & Carr, 2006) to include correct normal generation for the mesh as well as a higher-quality intersection approximation and acceleration using an interval tree. Finally, Tatarchuk, Shopf, & DeCoro (2007) presented an enhanced version of this basic technique. Their method is a hybrid between tetrahedron- and cube-based extraction techniques. First, they create a cubic grid of values. Using the latest hardware capability of *geometry shaders* (which allow addition of new geometry in the middle of the graphics pipeline) they use interval methods to extract only the contributing tetrahedra. Finally, they perform triangular isosurface extraction from this tetrahedral grid. In order to reduce linear artifacts in the extracted mesh, they perform cubic interpolation between sample points instead of linear.

2.5 Hybrid Methods

Hybrid methods have been developed in order to exploit the advantages of both surface extraction and ray-tracing. Graphics hardware is optimized for rendering triangular meshes, so surface extraction methods which produce such mesh representations of an isosurface can take advantage of this speed. However, triangle mesh approximations of isosurfaces have visible discontinuities at reasonable resolutions, while higher resolution meshes become large and slow to render. Ray-tracing methods do a great job of producing smooth, artifact-free images because each pixel is traced independently. However, tracing each ray through the volume is a non-trivial task which can take a significant amount of time, even when spatial data structures are employed to accelerate the process. Hybrid methods try to capture the advantages of both these methods, while avoiding their disadvantages. The process is to have surface extraction of high-level primitives, which are then refined further using a pixel-exact method such as ray intersection. Because not as many hybrid methods exist, we will discuss several techniques which do not specifically render isosurfaces.

One of the first examples of a hybrid volume rendering method (Shirley & Tuchman, 1990) created a tetrahedral representation of the volume. They noted that the 2D projection of tetrahedra onto the screen space is a set of one to four triangles, and that, for orthogonal view transformations, they could interpolate values across these triangles for each intersecting viewing ray instead of recomputing them. Specifically, the length of a specific ray through a position on the tetrahedra can be calculated by computing the lengths at each vertex and interpolating the values across the triangle faces. They exploit this by pre-integrating the volume at the vertex positions and interpolating across the triangles to fill the remainder of the pixels. Because each integral only has to be computed within a given tetrahedra, computing the vertex values is much faster than trying to trace through a larger volume, and does not require any acceleration data structures. Further, the interpolation step allows them to fill all the pixels which intersect a given tetrahedra with a small number of integration steps.

More recently, Sadowsky, Cohen, & Taylor (2005) presented a more advanced hybrid technique based upon similar ideas. They also create a tetrahedral grid and interpolate values across screen-space triangle faces, but, instead of directly interpolating the integral result, they interpolate coordinates which allos them to calculate the integral even under perspective projection. They also extended the representative power of each tetrahedral primitive by using a Bézier tetrahedron spline primitive instead of linear interpolation. This gave their method the ability to render very smooth results even on low-resolution data. In order to extend this idea to isosurface rendering, Loop & Blinn (2006) developed a closed-form expression for ray-intersection with any isosurface of a Bézier tetrahedron. They also created screen space triangles and interpolate the depth values, evaluating the isosurface formula at each pixel. They were able to render quadratic, cubic, and quartic Bézier tetrahedra, but any higher order primitives would not work because there is no closed-form solution for the roots of quintic and higher polynomials. Although their rendering algorithm produces very high quality isosurfaces for any mesh of Bézier tetrahedra, they did not present any way of generating volume data in their format, so their work has seen limited application.

Finally, Rossl *et al.* (2003) presented an offline rendering method which used a specific tetrahedral decomposition of quadratic Bézier tetrahedra to render smooth isosurfaces of volumetric data. Their method used the idea of a *super spline*, which is a mesh of splines with continuity constraints across spline boundaries. Although the number of free parameters is very large with an unconstrained spline grid, adding continuity constraints greatly reduces the number of legal splines in the grid. They showed how all the spline weights in a specific tetrahedral grid could be completely determined from a small subset, which could be set to approximate the volume. The referred to this method as *repeated averaging*, since the continuity constraints provided enough information to reconstruct all of the weights through repeated averaging of the initial subset. They used interval culling to select only tetrahedra which intersected a given isosurface, rendering them with direct ray-intersection equations. Because of the spline smoothness properties, their method generated higher quality isosurfaces than most surface extraction methods, but their repeated averaging scheme was a heuristic and did not represent a common reconstruction filter.

Chapter 3

BACKGROUND

This section provides some background information necessary to describe the BT volume rendering method presented later. First, Section 3.1 gives a formal definition of the regularly sampled scalar field volume format which is used through the rest of this thesis. Section 3.2 provides a description of discrete convolution, which is the basic mathematical tool used to perform reconstruction filtering.

3.1 Regular Scalar Fields on the Integer Lattice

For the purposes of this paper, we will discuss only regularly sampled scalar-valued 3D fields. Each data point in these volumes is called a *sample point* and contains a single real number, usually interpreted as density at that point. Each volume has a *domain*, denoted D, which is the smallest cuboid containing all the samples. Each volume also has a *size*, represented as a triple (N_x, N_y, N_z), which is the number of unique sample positions in each dimension. The set of sample values is $\mathbf{I} = \{I_{ijk} : 0 \leq i < N_x$, $0 \leq j < N_y$, $0 \leq k < N_z\}$, where (i, j, k) is an integer identifier for each sample.

In order to simplify our notation, we will implicitly transform each volume so that D overlaps the cuboid $C := (0, 0, 0) \times (N_x - 1, N_y - 1, N_z - 1)$. Under this transformation the *identifier* triplet (i, j, k) for each sample I_{ijk} is also the *position* of that sample. We will call this the *integer lattice* of the volume, since it corresponds to a subset of $\mathbb{Z}^3$. Under this transformation a scalar volume A is defined as a mapping $\mathbf{I} \to \mathbb{R}$. For the rest of this thesis when we refer to a *volume* we will be refering to this definition, unless specifically stated otherwise.

3.2 Discrete Convolution

Reconstruction filtering is the process of creating a continuous function from a discrete volume. Reconstruction requires blending sample points from the volume using a *filter kernel* to define how the sample points should be blended together. The mathematical tool which we use to perform reconstruction filtering is called *discrete convolution*, which, in the context of volumes, is a function of 3D space that sums the reversed filter kernel with a volume. For some volume with domain C and integer lattice $\mathbf{I}$ (see section 3.1), the formula for the convolution of volume $A : \mathbf{I} \to \mathbb{R}$ by a kernel $G : \mathbb{R}^3 \to \mathbb{R}$ at the point $\mathbf{P} \in \mathbb{R}^3$ is the summation over $\mathbf{I}$ given by

$$(A * G)(\mathbf{P}) = \sum_{\mathbf{i} \in \mathbf{I}} A(\mathbf{i}) \cdot G(\mathbf{P} - \mathbf{i}) \tag{3.1}$$

The specific filter used can determine properties about the final reconstruction. For example, Gaussian and B-Spline filters are blurring filters, while Bilinear and Catmull-Rom filters produce aliasing. For a thorough examination of reconstruction filtering for volume data, including an examination of different reconstruction filters, we refer the reader to the work of Marschner & Lobb (1994).

3.3 Bézier Tetrahedra and Supersplines

Our method will heavily use a specific type of 3D polynomial called the Bézier Tetrahedra (BT for short). The cubic BT are a set of cubic polynomial solids represented in the Bernstein basis where each element of the family is defined within a bounding tetrahedron domain by a set of weights on the twenty basis polynomials. Let the points $\mathbf{v_i} : \{\mathbf{i} \in [1,4]\}$ be the vertices of a tetrahedra $\mathbf{T} \in P(\mathbb{R}^3)$ (three-dimensional Euclidian projective space), and the set of 20 weights $\mathbf{w} = \{w_{ijkl} \,:\, i+j+k+l=3\}$. Note that the weight *positions* are fixed relative to the bounding tetrahedron and only their *value*, which is a scalar, changes. Together, $\mathbf{w}$ and $\mathbf{T}$ define a Bézier Tetrahedron, τ.

In order to actually evaluate the BT, we must go through a little more work. The matrix $\mathbf{M_T} = [\mathbf{v_1}, \mathbf{v_2}, \mathbf{v_3}, \mathbf{v_4}]^T$ is an affine transform into the *barycentric space* of $\mathbf{T}$ (Loop & Blinn, 2006). Given a point $\mathbf{P} = (x, y, z, 1) \in P(\mathbb{R}^3)$ transform it into the barycentric space of $\mathbf{T}$ by $\mathbf{r} = (r, s, t, u) = \mathbf{P} \cdot (\mathbf{M_T})^{-1}$. The formula for evaluation of $\tau := \{\mathbf{T}, \mathbf{w}\}$ is

$$bt(\mathbf{P}) = \sum_{i+j+k+l=3} w_{ijkl} \begin{pmatrix} 3 \\ ijkl \end{pmatrix} r^i s^j t^k u^l \tag{3.2}$$

where $\begin{pmatrix} 3 \\ ijkl \end{pmatrix}$ is the *multinomial function* defined by $\left(\frac{d!}{i!j!k!l!}\right)$ for all $i+j+k+l=d$.

3.3.1 Properties of Bézier Tetrahedra

Bézier tetrahedra have several properties which make them very intuitive to work with.

Weight Positions Every weight w_{ijkl} is associated with position $(i\mathbf{v_0} + j\mathbf{v_1} + k\mathbf{v_2} + l\mathbf{v_3})$ in the barycentric space defined by $\mathbf{T}$. Each weight associated with a vertex of the bounding tetrahedra is equal to the evaluation of the BT at that point, so it is relatively easy to tell what the general topology of a BT is by looking at the corner values.

The Bounding Property All of the weights follow the *bounding property*, which gives limits on the range of the solid based on the weight set. For a given BT with weights $\mathbf{w}$, any value resulting from evaluation of the BT will have to be between the highest and lowest weight values. In other words, for any point $\mathbf{P}$ in the domain of a BT with weights $\mathbf{w}$ the resulting value $Q = bt(\mathbf{P})$ must be between the maximum and minimum values in $\mathbf{w}$. Therefore, one can determine easily if a given BT has an isosurface of a specific level by computing the min and max weights.

Addition of Bézier Tetrahedra Because Bézier tetrahedra are polynomials, we can add them as long as their domain tetrahedra are the same. BT are closed under addition, so the result is another BT which is computed by adding all co-located weight values. This follows directly from the definition in equation 3.2, so we omit a proof.

Total Degree Consider any n-variate polynomial

$$P(\mathbf{x}) = \sum_{i,j,...,k} C_{ij...k}\, x_1^i x_2^j ... x_n^k$$

The *total degree* of this polynomial is given by $\sup(i+j+...+k)$ for all $i, j, ..., k$. According to definition of cubic Bézier Tetrahedra from equation 3.2, the total degree of these splines is also cubic. This fact, which does not hold for tensor product splines, is what allows us to find closed-form ray-intersection equations in Section 3.5.1, which is what allows us to maintain high-performance and achieve interactive rates. Trying to modify our method to work with cubical domain Bézier patches would incur a large performance penalty, since ray-intersection would have to be performed using an iterative method which does not map as well to graphics hardware.

3.3.2 Continuity Constraints

Bézier tetrahedra have constraints which will enforce continuity between any *adjacent* pair. For our purposes a pair of tetrahedra are adjacent when they share a common triangular face but have no overlapping volume, and two BT are adjacent if and only if their bounding tetrahedra are adjacent. Since cubic BT are cubic polynomials we can enforce up to C_3 continuity between any pair, but this would over-constrain the tetrahedra and limit it's usefulness. For this reason, we limit ourselves to C_0 and C_1 continuity since they will help prevent gaps and ensure basic smoothness without limiting the representative power of the splines.

Enforcing C_0 continuity is simple: each pair of co-located weights across the shared face must have equal value. Consider a pair of BT τ and τ' defined by $\{\mathbf{w}, \mathbf{T} = [v_1, v_2, v_3, v_4]\}$ and $\{\mathbf{w}', \mathbf{T}' = [v_1, v_2, v_3, v_4']\}$, respectively. C_0 Adjacency across the common face (v_1, v_2, v_3) is ensured by enforcing $w_{ijk0} = w'_{ijk0}$ for all $i + j + k = 3$. The center of Figure 3.1 shows an example of this.

C_1 continuity is slightly more complex. It can be helpful to think of the weights of a BT in *levels* depending on how far away they are from a given face. For example, weights in τ are at level l from face (v_1, v_2, v_3) if they have the form w_{ijkl}. The left of Figure 3.1 shows a BT with weights in the same level connected, with the far left showing levels counted from the bottom face and the one next to it showing levels counted from the front-right face. In general, C_n continuity across a common face involves weights in the first n levels in each BT relative to the common face. Specifically, C_1 continuity across τ and τ' in achieved by enforcing that

$$w'_{i,j,k,1} = \left(\frac{[i, j, k, 1]}{3} \times \mathbf{M}'_{\mathbf{T}} \mathbf{M}_{\mathbf{T}}{}^{-1} \right) \cdot [w_{i+1,j,k,0},\ w_{i,j+1,k,0},\ w_{i,j,k+1,0},\ w_{i,j,k,1}]$$

The middle term of this equation $\left(\frac{[i,j,k,1]}{3} \times \mathbf{M'_T}\mathbf{M_T}^{-1}\right)$ represents the barycentric position of w'_{ijk1} relative to $\mathbf{T}$. The intuitive explanation is that this equation enforces that all "diamonds" formed by five adjacent weights across the common face - three on the face and one in each of the tetrahedra - lie on the same hyperplane. The bottom right of figure 3.1 shows two adjacent cubic BT with one of these "diamonds" marked with close-dotted lines.

3.3.3 Bézier Tetrahedra in Super Splines

Although individual splines can represent simple shapes, in order to perform useful work we will have to construct a set of splines which all contribute to representing a larger, more complex volume. In order to maintain coherence across the larger volume it is necessary to enforce some level of continuity constraints as described in section 3.3.2. For our application this means either C_0 or C_1 constraints across all BT boundaries, without mixing continuity levels. This construction is called a *super spline* (Rossl *et al.*, 2003), and will be the basis for our intermediate volume format.

3.4 Graphics Programming

The nature of the rendering problem has pushed the design of graphics hardware towards massively parallel stream processing systems. These processors, referred to as *Graphics Processing Units* (GPUs), have a high-level sequential order called the graphics pipeline which defines how individual rendering primitives are processed into pixels. Each stage of the pipeline is implemented in very wide parallel processors which have built-in scheduling for each compute unit. Each stage in this process has a *kernel* program which takes one or more primitives from the previous stage and produces primitives of the next stage. The data between stages is referred to as a *data stream*, and each kernel program

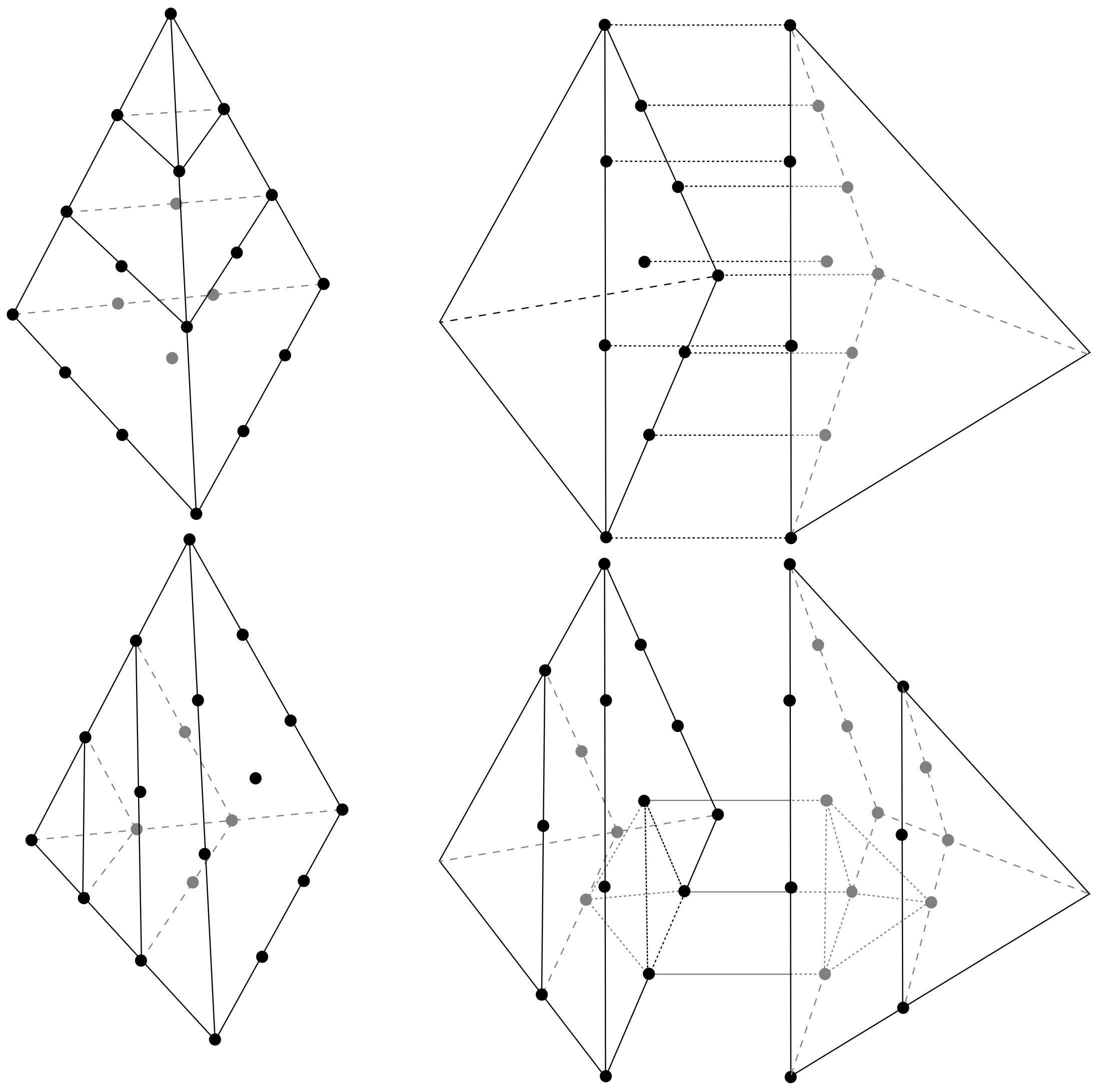

FIG. 3.1. On the left are two diagrams showing a cubic Bézier tetrahedron with the twenty weight positions as black dots. Note that weights can be divided into “bands” which are co-planar - top shows division according to bottom face, bottom shows division according to front-right face. The right-top image is of two adjacent Bézier tetrahedra, slightly separated for clarity. Co-located weights are connected with dashed lines - constraints for C_0 continuity across the splines requires that co-located weights have equal value. The image on the bottom-right shows a single C_1 continuity constraint for the same two tetrahedra - all weights connected with the close dashed lines have to lie on the same hyperplane.

processes the minimum number of elements on its input data stream in order to allow it to produce the primitives on its output stream. Although there are many different stages in the graphics pipeline, several of them have the ability to load user-defined kernels and perform custom operations on them. The three programmable stages are called the *vertex stage*, the *geometry stage* (Blythe, 2006), and the *pixel stage*, and custom kernel programs in these stages are called *shaders*. The left of Figure 3.2 shows the pipeline in full.

The Vertex Stage The first programmable stage in the graphics pipeline, the vertex stage, takes one vertex as input and sends one vertex as output. Individual vertices need to have a position, but can also have other data associated with them, such as color. The standard use of this stage is to transform the vertex into the final position given a transformation defined by a camera. Although each execution instance of a vertex shader only has access to one vertex, small amounts of extra data can be passed into the shader through the use of *constant buffers*, which are available to all of the programmable stages. This allows custom transformations or other data to be passed into every execution instance of the shaders, and can be changed very quickly for every frame rendered.

The Geometry Stage The transformed vertices are then marshaled by the hardware into triangles, which are then passed into the geometry stage. This stage can output zero or more triangles, although the maximum number of triangles which it can output is relatively low (generally a few dozen or less). If the data output from the vertex shader does not define the connectivity required to determine triangles, the geometry shader can input single vertices as well.

The Pixel Stage The triangles output by the geometry shader are then passed to the *rasterizer*, which is a fixed piece of hardware that determines which pixels the triangle covers. These pixel locations are then passed to the pixel stage, which is responsible for either computing a final color or discarding the pixel so that no color is output. Attributes from the vertex data can be interpolated across the triangle for input to each execution instance of the pixel shader. See the left of Figure 3.2 to see a flow diagram of this process. Bold elements represent a single instance of input to the next stage, with grayed-out elements shown for context.

3.5 Rendering Bézier Tetrahedra

Although BT and supersplines of BT are a nice formulation of 3D polynomials into a format which is easily controlled by the input weights, it is really only of use to us if it can be rendered quickly to the screen. It turns out that rendering of BT is a problem which can be solved very quickly using a method presented by Loop & Blinn (2006) which takes advantage of the parallel processing power of graphics hardware. In order to describe this method, it is necessary to introduce a way of representing BT as tensors.

3.5.1 The Tensor Formulation

Tensors provide a generalized understanding of matrix multiplication and dot products through the notion of *contraction*. We will use Einstein Index Notation as described by Blinn (2003) to represent tensor contraction. To summarize, when the same symbol appears as both a superscript and subscript in the same equation, an implied summation is performed at that index, where the superscript represents contravariant indices (a column of a matrix) and subscripts represent covariant indices (a row of a matrix). Consider the BT defined by a tetrahedron $\mathbf{T}$ and weight set $\{\mathbf{w}\}$. We can construct a 4^3 tensor of control points $\mathbf{B}$ by

$$\mathbf{B}_{ijk} = w_{e_i+e_j+e_k} \tag{3.3}$$

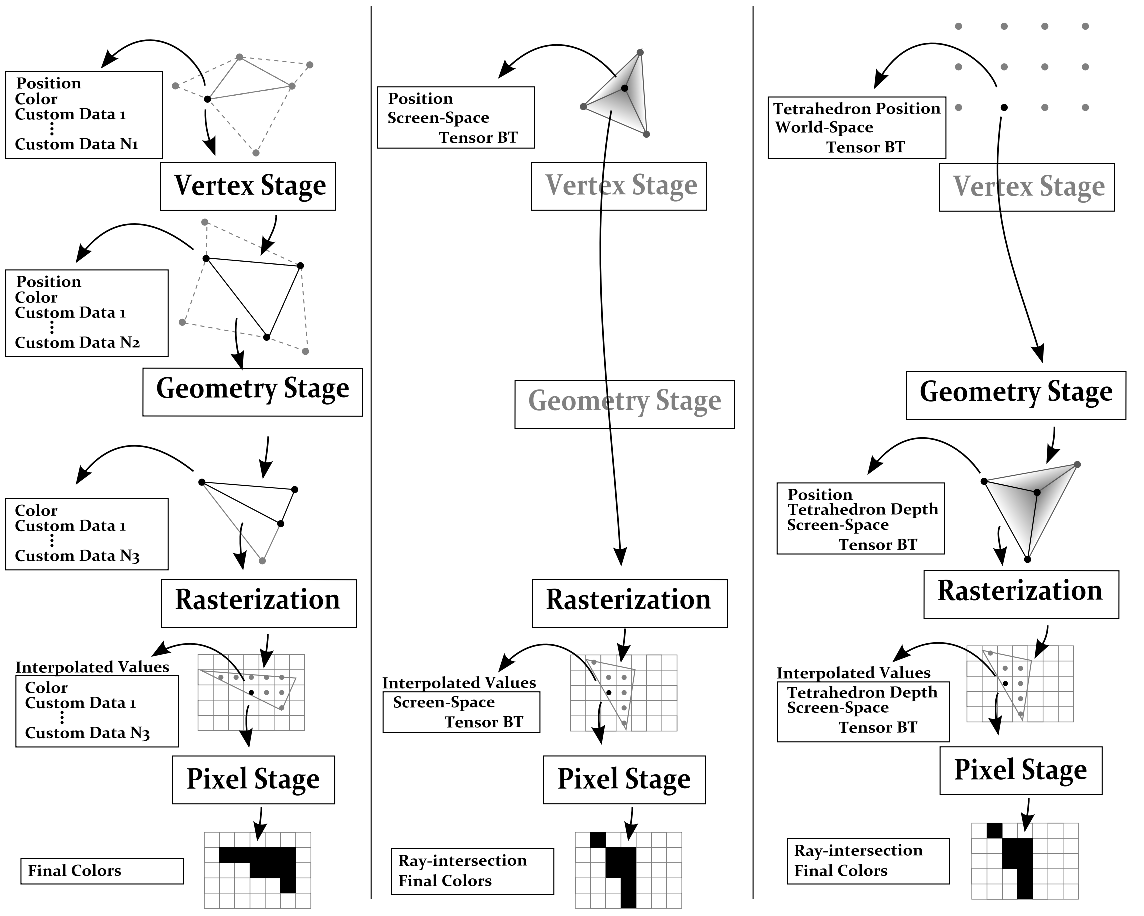

FIG. 3.2. On the left is the generic programmable graphics pipeline. Bold elements represent a single instance of input to the next stage, with greyed-out elements shown for context. Boxes on the left show the components of each element at each stage. *Vertex shaders* transform single vertices. Triangles are then re-assembled as input to the *Geometry shader*, which can output zero or more triangles. The fixed-function *rasterizer* computes pixel coverage for each triangle and passes them to the *pixel shader*, which produces a color. Vertex elements are interpolated before being passed to the pixel shader. The middle diagram shows how Loop & Blinn (2006) render Bézier tetrahedra on the pipeline by pre-computing screen-space tensors, as well as screen-space triangles which interpolate the depth of each viewing ray. They skip the vertex and geometry stages, computing ray-intersection in the pixel shader. Finally, the right shows how they suggest changing their method when geometry shaders are available. In this case only a single vertex is stored for each tetrahedron, which is expanded into screen-space triangles in the geometry shader.

where e_x is a four component vector with a 1 at position x and all other components equal to 0. Cubic BT have a tensor form defined as the three contractions

$$bt(\mathbf{P}) = \mathbf{r}^i\mathbf{r}^j\mathbf{r}^k\mathbf{B}_{ijk} \tag{3.4}$$

with $\mathbf{P}$, $\mathbf{r}$ from equation 3.2. Although this equation is mathematically equivalent to equation 3.2, since $\mathbf{B}$ is much larger than $\mathbf{w}$ (it contains 64 components instead of 20), it will be much less efficient to evaluate. This is because the tensor form contains much redundancy in the weight tensor $\mathbf{B}$ when compared to the weight set $\mathbf{w}$. The reason for using the tensor formation is to take advantage of some useful properties of tensor contraction.

Tensor-ray Intersection Tensor notation also allows efficient calculation of ray-intersection with the zero isosurface of a BT in tensor form, which is defined by setting equation 3.4 equal to zero. Consider a univariate cubic polynomial in Bernstein basis that corresponds to a single ray through a BT, given by

$$\sum_{i=0}^{3} \binom{3}{i} (1-v)^{3-i} v^i a_i = 0$$

where $v \in [0,1]$ corresponds to the portion of the ray inside the tetrahedron bounds. Finding the roots of this equation will tell us where the ray intersects the zero-isosurface of the spline, but first we have to calculate the coefficients a_i from the tensor form. Consider two points $\mathbf{p}$, $\mathbf{q}$ which both lie on the bounding tetrahdron of a given BT τ and form the line $(1-v)\mathbf{p} + v\mathbf{q}$ through the tetrahedron. The coefficients a_i can then be written

$$
\begin{aligned}
a_0 &= \mathbf{p}^i \mathbf{p}^j \mathbf{p}^k \bar{\mathbf{B}}_{ijk} \\
a_1 &= \mathbf{p}^i \mathbf{p}^j \mathbf{q}^k \bar{\mathbf{B}}_{ijk} \\
a_2 &= \mathbf{p}^i \mathbf{q}^j \mathbf{q}^k \bar{\mathbf{B}}_{ijk} \\
a_3 &= \mathbf{q}^i \mathbf{q}^j \mathbf{q}^k \bar{\mathbf{B}}_{ijk}
\end{aligned}
\tag{3.5}
$$

Since this equation is in Bernstein form it conforms to the same bounding property that Bézier Tetrahedra have, namely that if all coefficients are the same sign then the polynomial cannot have any roots in the $[0, 1]$ range. This allows us to perform early-termination tests to avoid the expensive root finding algorithm.

In order to actually find the roots we use the method developed by Blinn (2006), which, although not the fastest root-finding algorithm available, is the most numerically stable algorithm we are aware of. Numerical stability is very important for our application since we will be performing this calculation on graphics hardware, which does not efficiently support double-precision floating point representations.

Tensor Transformation While $\mathbf{T}$ is in 3D projective Euclidean space, $\mathbf{w}$ and $\mathbf{B}$ are in the barycentric space defined by $\mathbf{T}$, so evaluation of any point requires first transforming the point into barycentric coordinates and then evaluating equation 3.2 or 3.4 with the transformed point. It turns out that we can actually transform the weight tensor into the space of the evaluation point and perform the evaluation in that space without transforming the point. The weight tensor $\mathbf{B}$ after transformation into Euclidean space, denoted $\bar{\mathbf{B}}$, can be calculated by

$$
\bar{\mathbf{B}}_{ijk} = \mathbf{W}_i^l \mathbf{W}_j^m \mathbf{W}_k^n \mathbf{B}^{lmn} \tag{3.6}
$$

where $\mathbf{W} = \mathbf{M}_{\mathbf{T}}^{-1}$ (see equation 3.2).

Intersecting Arbitrary Isosurfaces We can use tensor transformation of Bézier Tetrahedra to simplify ray-intersection with any isosurface level. In Section 3.5.1 we showed how to efficiently render the zero-isosurface of a tensor-form BT. It turns out that we can transform a BT so that any level isosurface is equal to the zero-isosurface under the transformation. Consider the case where we set Equation 3.4 equal to a constant c. In order to perform this transformation we need to identify a space where one of the polynomial basis functions is a constant term which can "absorb" c, and transform the resulting weights back into the original space. By inspection, Euclidian space will serve this purpose well since the polynomial basis functions will be in power basis, which has a constant term corresponding to position $(i, j, k, l) = (0, 0, 0, 3)$.

Consider the tensor form BT $\mathbf{B}$ which we want to transform into $\hat{\mathbf{B}}$ so that the c-isosurface of $\mathbf{B}$ becomes the 0-isosurface of $\hat{\mathbf{B}}$. We can evaluate equation 3.6 to get the Euclidian space tensor, where Equation 3.3 shows us that the only value in the which contains the weight from position $(0, 0, 0, 3)$ is $\bar{\mathbf{B}}_{3,3,3}$, so we can subtract c from that term to get the new tensor, which can be transformed back into the original space.

3.5.2 Rendering Bézier Tetraherda

Now that we have developed the mathematical basis required to perform ray-intersection with any isosurface of a BT spline solid, we have to design an actual rendering pipeline which executes the intersection efficiently on graphics hardware. Our contributions in this area are extensions of a method first described by Loop & Blinn (2006), which we will describe in detail here. Because there was no hardware support for geometry shaders when they published their paper, they presented two methods - one for hardware which they had access to and one looking forward to upcoming hardware. These two methods are depicted as the center and right columns of Figure 3.2, respectively.

Screen-Space Triangles The basic idea of both methods is to create screen-space triangles with depth information at each vertex which is then interpolated across the triangles, similar to previous tetrahedron-based methods (Shirley & Tuchman, 1990; Sadowsky, Cohen, & Taylor, 2005). The simplest method for decomposing a tetrahedron into screen-space triangles contains two cases: the three-triangle case when one vertex is contained (in 2D screen space) within the convex hull of all four vertices, and the four-triangle case where all four vertices contribute to the convex hull. Although some tetrahedron orientations will produce one or two screen-space triangles, they can be expressed as degenerate versions of these two cases. Since these cases are very rare, it is reasonable to ignore them. Example three- and four-triangle orientations can be seen in Figure 3.3.

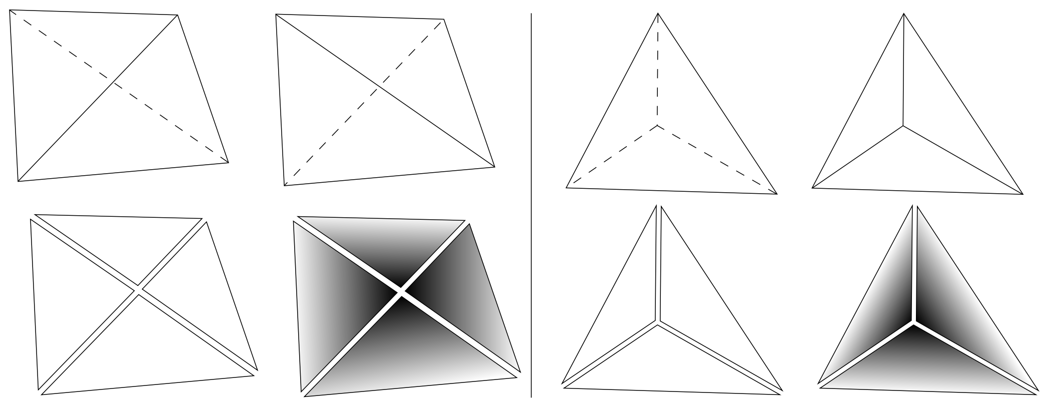

FIG. 3.3. Three- and four-triangle screen-space projections of tetrahdera. Two four-triangle orientations appear on the top left, with two three-triangle orientations on the top right. Below them are the screen-space triangle projections, along with a visualization of depth interpolated across each triangle from vertex values. Individual triangles have been separated slightly to clearly differentiate them.

The purpose of this construction is to allow each instance of the pixel shader to determine the points $\mathbf{p}$, $\mathbf{q}$ from Equation 3.5 for ray-intersection calculation, which is performed in screen-space.

Transforming the BT Tensor to Screen-Space The transformation into screen space is defined by a 4x4 transformation matrix called the World-View-Projection matrix, denoted $\mathbf{WVP}$. Because the vertices of the bounding tetrahedron $\mathbf{T}$ are in Euclidean space already, applying the World-View-Projection transformation directly will transform them into screen space. The BT weights, however, are in the barycentric space defined for Equation 3.2 above, so we will need to multiply the transformation from barycentric coordinates into Euclidean space ($\mathbf{M_T}$) with the World-View-Projection transformation. The inverse of this composite transformation provides us with an Equation for the barycentric coordinates $\mathbf{r}$ of a screen space point $\mathbf{P_s}$ given by

$$\mathbf{r} = \mathbf{P_s} \cdot (\mathbf{M_T} \cdot \mathbf{WVP})^{-1} = \mathbf{P_s} \cdot \mathbf{W}$$

where $\mathbf{W}$ is the inverse composite transform. We can calculate the transformed weight tensor $\bar{\mathbf{B}}$ by equation 3.6.

Hardware Mapping Loop and Blinn presented two different ways of mapping these steps onto hardware. The first technique, which is the one they implemented, pre-computes the screen-space weight tensors on the CPU. They also pre-computed the screen-space triangles on the CPU and passed the tensor information into the GPU through vertex data for the ray-intersection calculations. The pixel shader is able to reconstruct the points $\mathbf{p}$, $\mathbf{q}$ for ray-intersection from the current position (which is interpolated from the vertex positions) and the depth position, giving $\mathbf{p} = [x, y, z, 1]$ and $\mathbf{q} = [x, y, z + depth, 1]$. Since these values are in screen-space and the tensor values are in screen-space, they are able to evaluate the ray-intersection (Equation 3.5) directly. The disadvantage of this method is that a lot of computation has to be done on the CPU as a pre-process, limiting the scalability of their method. This makes it more of a proof-of-concept than a reasonable rendering architecture.

Their second method, which they did not implement for lack of hardware support, stored a single vertex with all of the BT coefficients as user data. The performed no pre-processing on the CPU, transforming the tensor into screen-space and generating the depth-interpolating screen-space triangles in the geometry stage. The pixel stage was the same. In theory this method is much preferred to the first one because it does not perform any pre-process on the CPU and therefore scaled much better. Both of these methods, however, transform the weight tensor every frame, which is an expensive operation for the CPU or the GPU.

Chapter 4

METHOD

The foundational construction of this work, the *BT volume*, is a continuous 3D function representation based upon the concept of tetrahedral super-splines introduced in Section 3.3.3. It is described in detail in Section 4.1, followed in Section 4.2 by a description of how to generate a BT volume as the best least-squares approximation of a given function. One of the most important features of BT volumes is that the set of BT volumes is closed under convolution with a scalar-valued volume data set. In other words, if we have a BT volume (approximating, for instance, a reconstruction filter) and any scalar-valued volume data set, their convolution will also be a BT volume. Using this fact, we can compute a BT volume which is the exact reconstruction of an arbitrary data set as long as the reconstruction filter we use is also represented as a BT volume. This property of the BT volume format, which is described in detail in Section 4.3, is what allows us to separate the convolution and rendering steps in our algorithm. Finally, Section 4.4 describes how to actually render a BT volume efficiently using graphics hardware. Most of this work has was published in 2007 (Kloetzli, Olano, & Rheingans, 2008).

4.1 BT Volumes

This Section presents a formal definition of the BT volume, along with some justification for the usefulness of its design. Section 4.1.1 describes how we divide a volume domain into unit cubes, which we call *voxels*. This is followed in section 4.1.2 by a description of general tetrahedral partitions and the more specific subset of *κ-partitions*, along with some useful properties of κ-partitions. The formal definition of a BT volume as a superspline on a κ-partition is given in section 4.1.3.

4.1.1 Voxels and Shift Invariance

Given a volume A with domain C and integer latice **I**, a *voxel* $A_{i,j,k}$ of the volume is defined as any cube $(i, j, k) \times (i + 1, j + 1, k + 1)$ of $A * G$ for some reconstruction filter G. We would like to have similar definition of a voxel for a BT volume, but BT volumes are continuous functions and not discrete volumes, so we would prefer to have a definition which does not require the reconstruction filter G. Therefore we forget the reconstruction filter and define a voxel $V_{i,j,k}$ of a BT volume V as the cube $(i, j, k) \times (i+1, j+1, k+1)$ of the volume. This may seem like a simplistic definition, but it turns out that voxels will help us to capture the idea of *shift invariance*, which will be needed later to perform convolution with BT volumes.

4.1.2 Tetrahedral Partitions and κ-partitions

BT volumes are comprised of individual Bézier splines on tetrahedra. Since each of the spline primitives is defined only within its bounding tetrahedron, we need to have a way of dividing the domain C of the BT volume into tetrahedra. This type of division is called a *tetrahedral partition* of C, and is commonly used in volume rendering (Gerstner & Rumpf, 1999; Loop & Blinn, 2006; Treece, Prager, & Gee, 1999; Anderson, Bennett,

& Joy, 2005; Rossl *et al.*, 2003). Notation in this section was developed by us to formally define partitions which are applicable to our method. Useful tetrahedral partitions will be *coverings* and *packings*; that is, every point in the domain will be covered by at least one tetrahedron, and no two tetrahedra will have overlapping volume (note that adjacent boundaries must overlap). In the general case tetrahedral partitions may be completely unstructured, but in general regular patterns of tetrahedra are found to be more useful. Carr, Moller, & Snoeyink (2006) presented an overview of regular tetrahedral partitions. Our method will require a special type of regularity in the partition which has not been formally defined in previous work.

We introduce the notation of a *κ-partition* as a special class of tetrahedral partition which meets the extra constraints imposed by our method. In addition to covering and packing, κ-partitions must meet two additional constraints: they must be valid *voxel exact partitions* and they must be *shift invariant*. Exact partitions are covering partitions that are contained within the bounds of the volume; every point in the volume has at least one tetrahedron covering it and every tetrahedron in the partition is contained inside the domain space. Visually, this means that no tetrahedra are "sticking outside" of the domain. *Voxel exact* partitions must include as subsets exact partitions of each voxel cube in the domain. Finally, shift invariance requires that all voxels have partitions that are the same when shifted so that the voxel domains are coincident. Figure 4.1 shows several examples of triangular partitions in order to demonstrate these properties in 2D, as well as a valid κ-partition.

Shift-Invariance Examined Shift invariance is an essential property for our purposes which will be used later, so this section provides a more rigorous examination of its implications. Define the *cannonical mapping* of a κ-partition, denoted $\kappa_{ijk} : V_{ijk} \rightarrow H$ (where H is a set of tetrahedra which partition the voxel $V_{0,0,0}$), to be

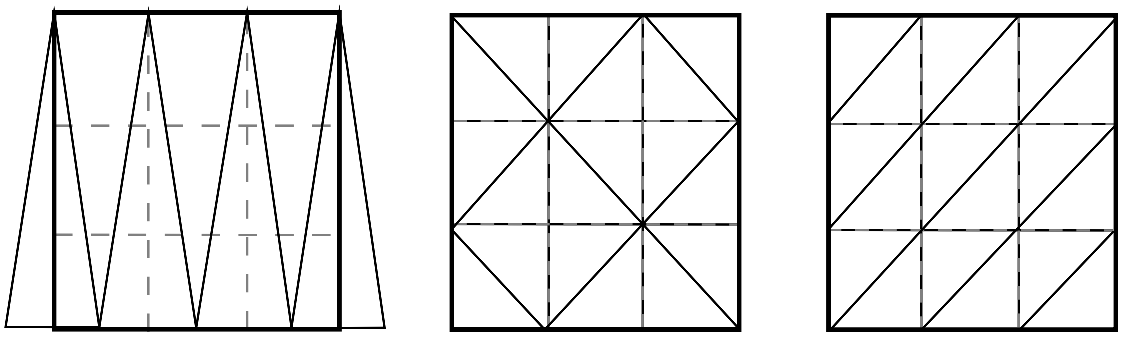

FIG. 4.1. Three example triangular partitions of a square domain. Voxel boundaries are dotted grey lines. Left: *Covering* and *packing* triangular partition. Center: A partition which is *regular*, *exact* (*covering* and *contained*), and *voxel exact* (each voxel is has an *exact* sub-partition), but is not *shift invariant* and so not a κ-partition. Right: A valid κ-partition.

$$\kappa_{ijk}(\mathbf{p}) = \mathbf{T_h} : (\mathbf{p} - \lfloor \mathbf{p} \rfloor) \in \mathbf{T_h} \in H$$

The purpose of the canonical mapping is to demonstrate that each unique tetrahedron in the canonical voxel $V_{0,0,0}$ has a related tetrahedra in every other voxel V_{ijk} which differs only by the translation $[i, j, k]$. Consider a tetrahedral κ-partition $\bigtriangledown$ where $\bigtriangledown_{ijk}^{\mathbf{T_h}}$ denotes the Bézier tetrahedron in voxel V_{ijk} which is related in this way to tetrahedron $\mathbf{T_h}$ in the canonical set H. We will use this notation later to perform summations over Bézier tetrahedra which only differ by a translation.

Test κ-partitions Many common tetrahedral partitions are κ-partitions. We will limit our attention to the three specific κ-partitions shown in Figure 4.2, *freudenthal* with six tetrahedra per cube, two equivalent orientations of *face-divided* with 12 tetrahedra per cube, and *face-centered* with 24 tetrahedra per cube. Note that the minimal 5-tetrahedra partition of a cube is not a valid κ-partition because adjacent cubes have to be inverted in order to make their triangle faces align correctly, which makes the resulting partition not shift invariant.

There are several desirable properties which make certain tetrahedral partitions better than others (Carr, Moller, & Snoeyink, 2006). Of relevance to our application are *symmetry*, *minimality*, and *error* (which Carr, Moller, & Snoeyink (2006) refer to as *exactness*). Symmetric partitions are invariant under cardinal rotation and mirroring transformations, and ensure that directional artifacts are not present in the final output. The freudenthal and face-divided partitions are not symmetric, although the artifacts introduced by the freudenthal partition are much more severe than face-divided partitions.

Minimal partitions contain a small number of tetrahedra, and are desirable because they reduce further processing required later in the method. Freudenthal partitions have the smallest number of tetrahedra possible for κ-partitions, while face-centered and face-divided contain twice and four times that number, respectively. Finally, error determines how closely a BT volume based on a given partition can approximate a given function. As you would expect for our three partitions, this is mainly determined by the number of tetrahedra in each, because this determines the resulting representative power of the BT volume.

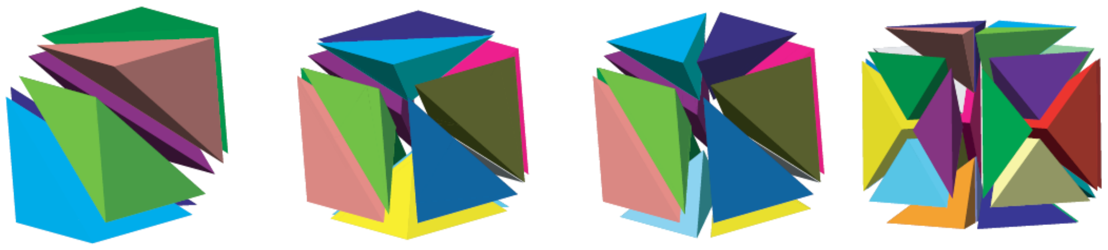

FIG. 4.2. The three partitions used in our work. Image from Carr, Moller, & Snoeyink (2006). From left to right: Freudenthal with 6 tetrahedra, the two different face-divided orientations with 12 tetrahedra each, and face-centered with 24 tetrahedra.

4.1.3 The BT Volume Definition

Given all of the background material we have presented, the definition of a BT volume is simple: associate a cubic Bézier spline with every tetrahedron of a κ-partition $\bigtriangledown$. As a notational shortcut, we will represent evaluation of the BT volume $\bigtriangledown$ as

$$\bigtriangledown(\mathbf{p}) = \bigtriangledown_{\lfloor \mathbf{p} \rfloor}^{\kappa_{\lfloor \mathbf{p} \rfloor}(\mathbf{p})}(\mathbf{P})$$

where κ is the canonical mapping from section 4.1.2. Figure 4.3 shows an example BT volume approximating a 3D Gaussian, with wireframe drawn to distinguish tetrahedral boundaries.

4.2 Approximating Continuous Volumes as BT Volumes

In order to generate a BT volume $\bigtriangledown$ to approximate an arbitrary continuous scalar-valued volumetric function $G : D \in \mathbb{R}^3 \to \mathbb{R}$ we have to create a κ-partition for D as described in section 4.1.2. The specific partition used will determine how close our approximation can be to the original function, since it will determine the number and distribution of weights per voxel (We assume that G has already been scaled so that D aligns to the desired integer lattice).

Once we generate our tetrahedral partition, we have to complete the definition of $\bigtriangledown$ by determining the optimal weights so that $\bigtriangledown$ approximates G as closely as possible. Since BT are collections of polynomials, a simple least-squares approach works well. We determine a set of sample points distributed evenly in the barycentric coordinates of each bounding tetrahedron, transforming the points into Euclidean space and evaluate the target function value. In our experience a single sample at each weight location (ten per tetrahedron) is sufficient to produce good results. We then construct a least-squares matrix with one element per matrix, computing the optimal weights.

There are several boundary conditions which can be enforced to help make the resulting BT volume well-behaved. First, we can ensure that the volume goes to zero smoothly around its entire border by forcing the first two rows of weights along the border to be zero. We can ensure C^0 continuity by performing one large least-squares system which contains one unknown for each weight in $\bigtriangledown$. Finally, each pair of adjacent tetrahedra introduce six new constraints to ensure C^1 continuity. Since each of these constraint types is linear, we use standard linearly-constrained least-squares. Because the number of terms in the least squares matrix can get very large for even small volumes, we used an iterative least-squares technique instead of singular value decomposition.

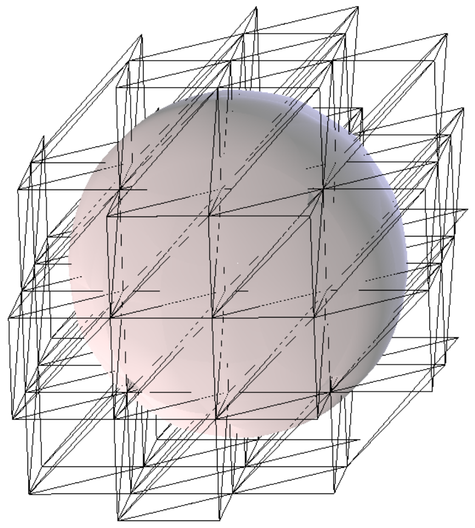

FIG. 4.3. An isosurface rendering of a BT Volume with the 6-tetrahedra freudenthal κ-partition approximating a Gaussian reconstruction filter, where lines denote tetrahedron boundaries. Note that tetrahedra which would not contribute to this particular isosurface are culled before rendering. This volume enforced smooth borders and C^0 continuity.

4.3 BT Volume Convolution

Although the least-squares method performs reasonably well for approximating arbitrary 3D functions, it also has several drawbacks. First, it takes a long time to solve the least-squares matrix, especially when the smooth boundary constraints are enforced, taking a few minutes for a volume as small as $6x6x6$. Also, least squares error reduction does not fit into the convolution framework easily since it does not allow many different reconstruction filters. It might be possible to minimize a different error metric to support this, but it would undoubtedly be very slow as well. What we really want is a new framework for creating BT volume data which does fit the reconstruction filtering model, which, it turns out, is possible.

Our goal is produce a BT volume $\diamond$ as the result of convolution of a scalar-valued discrete volume A and a reconstruction filter $\bigtriangledown$ (which is already approximated as a BT volume). Since a volumetric reconstruction filter is simply a scalar-valued volume function, we can create a BT Volume approximation by the method from the previous section. We can substitute $\bigtriangledown$ into the formula for discrete convolution given in Equation 3.1 to get

$$(A * \bigtriangledown)(\mathbf{P}) = \sum_{\mathbf{i} \in \mathbf{I}} A(\mathbf{i}) \cdot \bigtriangledown(\mathbf{P} - \mathbf{i})$$

Since $\mathbf{I}$ is the set of samples on the integer lattice, $\mathbf{P} - \mathbf{i}$ will be shifted by integer amounts in each direction from $\mathbf{P}$. Therefore, for all $\mathbf{P} - \mathbf{i}$ for a given $\mathbf{P}$,

$$\kappa_{ijk}(\mathbf{P} - \mathbf{i}) = \mathbf{T_h}$$

This means that every evaluation of $\bigtriangledown$ in the summation will fall in the same barycentric position (r, s, t, u) of a tetrahedron which is a shifted version of $\mathbf{T_h}$. Substituting Equation 3.2 for $\bigtriangledown$ in Equation 4.1 gives

$$(A * \bigtriangledown)(\mathbf{P}) = \sum_{\mathbf{i} \in \mathbf{I}} A(\mathbf{i}) \sum_{i+j+k+l=3} w^{\mathbf{i}}_{ijkl} \binom{3}{ijkl} r^i s^j t^k u^l$$

Since only the weights $\mathbf{w^i}$ depend on the outer summation (see Section 3.3.1 about addition of Bézier tetrahedra), we can re-arrange the order of the summations, leaving only the terms which depend on the summation over $\mathbf{i}$ inside the summation, to get

$$= \sum_{i+j+k+l=3} \binom{3}{ijkl} r^i s^j t^k u^l \left(\sum_{\mathbf{i} \in \mathbf{I}} A(\mathbf{i}) w^{\mathbf{i}}_{ijkl} \right) \tag{4.1}$$

By inspection this is the equation of a BT with the weight component equal to the entire second summation, so we have achieved our goal of deriving $\diamond$. Note that $\diamond$ and $\bigtriangledown$ will both have the same κ-partition, although they can have different dimensions.

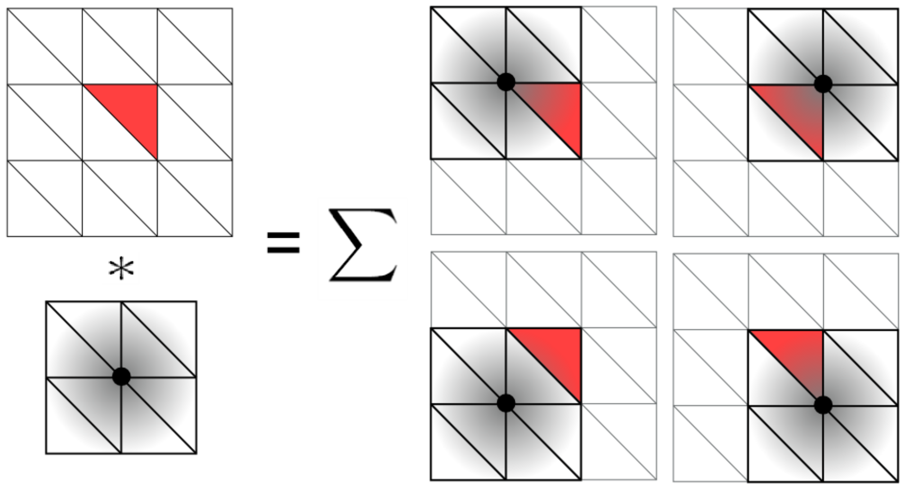

FIG. 4.4. We model convolution as a summation of BT solids. In order to convolve one tetrahedron of a BT volume, pictured above in red, with a BT volume filter with the same κ-partition, we need to sum together the contributions of overlapping BT in all possible translations of the filter, each scaled by the sample value at that filter position.

The intuitive explanation behind this derivation is to consider the view of convolution as a sum of kernel functions centered on each sample point and weighted by the sample value. Since the volume samples are evenly spaced and κ is the same for each voxel, each tetrahedron in the result will be covered by one and only one tetrahedron from *each* piecewise BT kernel in the sum (see Figure 4.4). So each tetrahedron of the full volume can be expressed as a sum of BT from the kernels. BT are closed under addition, so the result of the convolution is a single BT for each tetrahedron in the full volume.

This is the main result from our work and allows us to represent a volume convolved with a BT volume filter as a BT volume. The power of this result lies in the fact that BT volumes can be rendered in real time, thus allowing us to render high quality convolved volumes exactly, assuming we can represent the reconstruction filter as a BT volume.

4.4 Rendering BT Volumes

Our rendering algorithm is a modification of the second method proposed by Loop & Blinn (2006), which we have described in Section 3.5.2. The basic idea is to store each Bézier tetrahedron in the super spline as a single vertex, expanding it to screen-space triangles in the geometry shader and calculating ray-intersection in the pixel shader. Figure 4.6 shows the entire rendering pipeline, which consists of three stages.

Since our goal is to make the rendering as fast as possible, we perform some steps as a pre-process in order to minimize the work required per frame. Assume that we start with a BT $\diamond$. First, we compute the tensor form for each Bézier tetrahedron (Equation 3.3) and transform into Euclidian space (Equation 3.5.1). We compress the transformed splines back into the tetrahedral indexed form, storing them as custom data in a single vertex buffer. The position of each vertex is set to the center of the voxel that the tetrahedron belongs to. The following sections describe how we use graphics hardware in the three accelerated passes of our rendering algorithm.

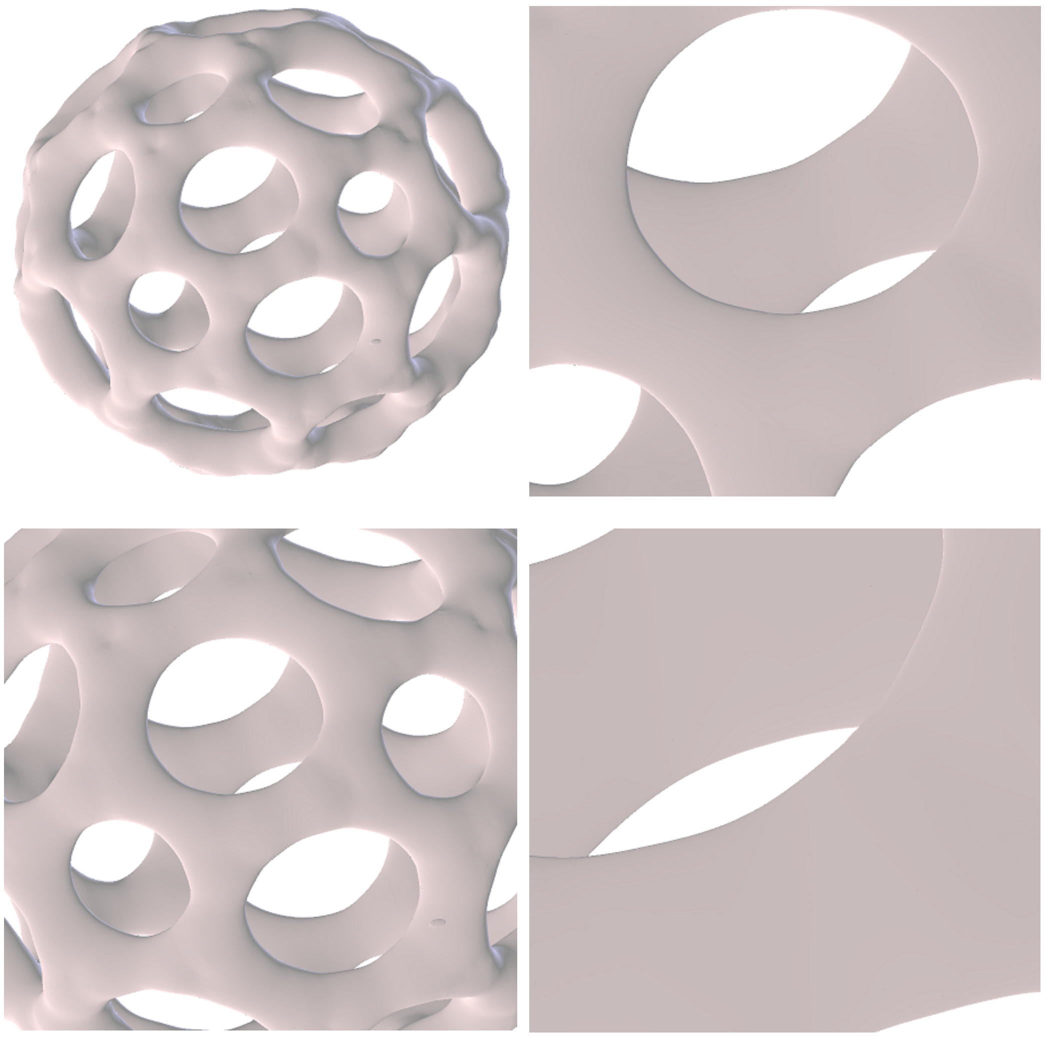

FIG. 4.5. Several shots in a zoom animation of a 32^3 volume reconstruction show the resolution independence of the BT Volume format.

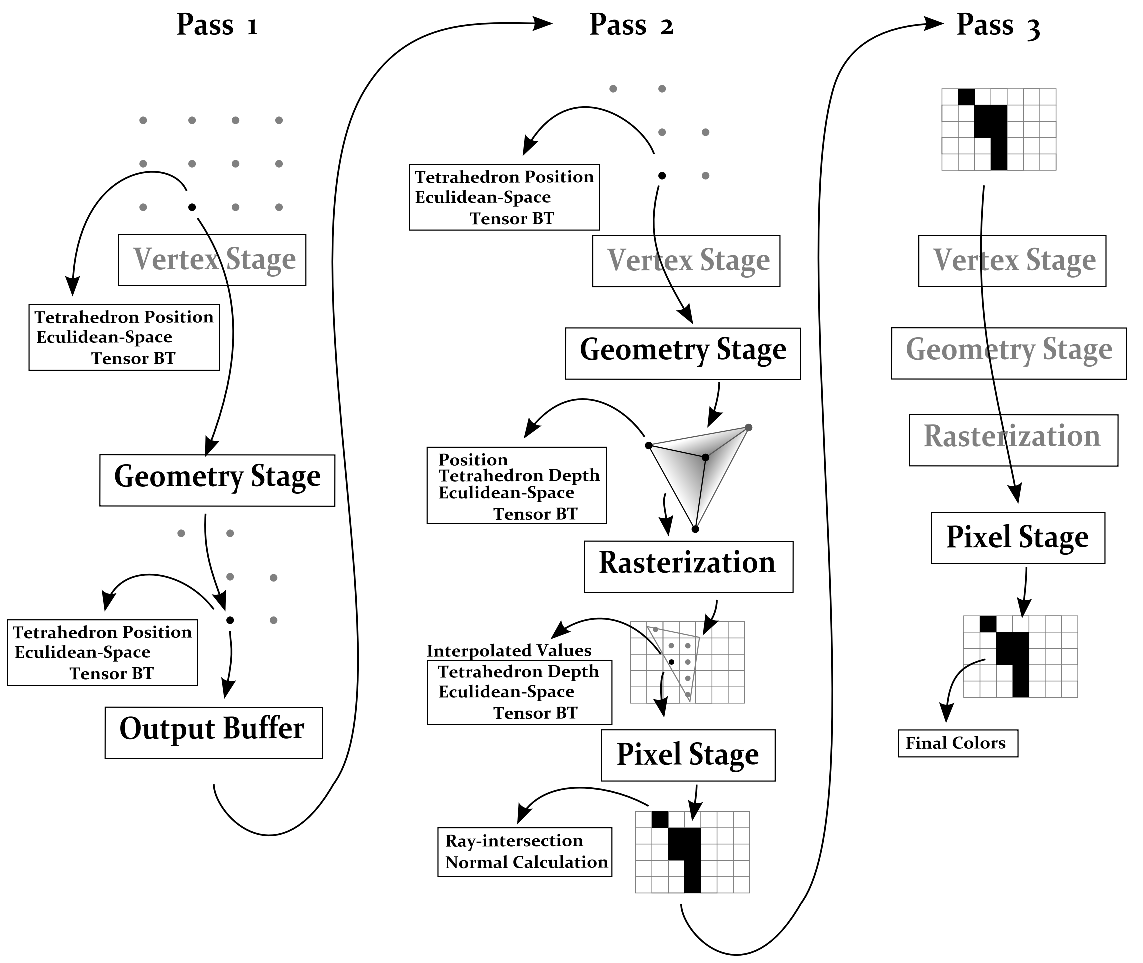

FIG. 4.6. Our BT volume rendering pipeline. Pass 1 removes all tetrahedra, which are stored as single vertices, that do not contribute to the current isosurface, storing the resulting set of contributing tetrahedra in an intermediate buffer. Pass 2 expands each tetrahedron into screen-space triangles, interpolating depth across the triangles to the pixel shader, which calculates ray-intersection and surface normals. Pixels which do not hit an isosurface are discarded. Pass 3 generates the final shading in another pixel shader.

4.4.1 First Pass: Culling

The first accelerated rendering pass uses the *stream-out* capability of geometry shaders. This hardware feature allows the output stream from the geometry shader stage to be stored in a buffer to be passed into later rendering passes. We use this capability to determine the subset of tetrahedra which actually contribute to a given isosurface value by using the bounding property of BT (Section 3.3.1). Our geometry shader for this stage checks the largest and smallest weight values, and passes the vertex to the output stream if and only if the target isosurface is between them. Before outputting the resulting vertex, however, it also adds the offset to the BT so that the zero-isosurface is transformed to the desired isosurface (Equation 3.5.1). Because each BT has already been transformed into Euclidian space, this is a single addition to the constant term. The resulting output stream is stored in a second buffer, which is used for all subsequent steps. Because the contents of this buffer will only change when the isosurface changes, we re-use this buffer until the user decides to change the isosurface level. This pass is depicted in the leftmost column of Figure 4.6.

4.4.2 Second Pass: Rendering

The rendering pass generates the actual pixel coverage and surface normal information in a screen-space buffer. It works from the culled buffer from the previous stage, so only BT which contribute to the current isosurface are processed. The geometry shader is responsible for creating the screen-space triangles to ensure correct pixel coverage, while the pixel shader calculates the actual ray-intersection and normal vectors. This pass is depicted in the center column of Figure 4.6.

Geometry Shader Our system uses the geometry shader to generate screen-space triangles to cover each tetrahedron, interpolating depth information across each triangle for use by the ray-intersection. We accelerate the process of creating screen-space triangles in the geometry shader by precomputing triangle coverage. Consider the problem of generating screen-space triangles for two tetrahedra which differ only by a translation. Although this problem is reasonable to calculate in a shader (Cohen, 2006), we choose to precompute the triangle breakdown in order to accelerate this stage. We compute the screen-space triangle coverage for each tetrahedron in the voxel $V_0, 0, 0$, translating the appropriate triangles into the correct position during rendering. Note that this does not work under perspective projection, since the triangle breakdown is not constant across translated tetrahedra in this case. Since scientific applications frequently use isographic cameras, we feel that this is a reasonable restriction.

Pixel Shader Finally, we use the pixel shader to determine if each pixel which hits the tetrahedron also hits the isosurface we are trying to render. Since our BT are stored in tetrahedral indexed form, we first have to expand them out to full tensor form. We can compute the equation of the viewing ray inside the tetrahedron to be the line between $[x, y, n, 1]$ and $[x, y, f, 1]$, where (x, y) is the screen position of the pixel and n, f are the near and far intersections of the viewing ray with the tetrahedron. We can compute this from the depth information interpolated across each triangle. Before we can compute the intersection, however, we have to translate this line into Euclidian space (since it is currently in screen space), which is a simple matrix multiplication.

Actually computing the intersection points is performed by solving for the roots of the cubic in equation 3.5 and choosing the solution with the smallest positive value which is inside the tetrahedron. If no valid root is found, the pixel shader discards (terminates) the pixel, and no data is output. If an intersection is found, the shader continues to find the surface normal at that intersection (Blinn, 2006) and stores that in the screen-space buffer at that pixel location.

4.4.3 Third Pass: Shading

The last pass of our algorithm generates the final pixel color. Our pixel shader detects if an intersection was found for each pixel, generating a background color for each empty pixel and evaluating our lighting model for each pixel with an intersection. Our lighting model is a simple cool-to-warm model (Gooch *et al.*, 1998). Final rendered images can be seen in Figures 1.2, 4.3, 4.5, and 5.7.

Chapter 5

RESULTS

This section presents the results we have obtained for our system. We present results for the three different κ-partitions from Section 4.1.2 - freudenthal, face-divided, and face-centered. First we present results for least-squares fitting of several reconstruction filters consisting of timing results and error analysis. Next we present analysis of the convolution step in our algorithm and space constraints, followed by rendering performance.

5.1 Least-Squares Approximation

We tested our least-squares approximations for three different filters: Gaussian with standard deviations 0.5, 0.6, and 0.7, B-Spline polynomial, and Catmull-Rom polynomial. We tested both C^0 and C^1 continuity for all three κ-partitions from section 4.1.2, which consist of 6, 12, and 24 tetrahedra per voxel, respectively. We fit all of the filters to a 6x6x6 BT volume. Table 5.1 shows the error from each of these least-squares fitting operations.

As we would expect, more tetrahedra in our κ-partition results in lower error in the fitting. Doubling the number of tetrahedra in the partition decreases the error substantially. The less stringent C^0 continuity constraints also introduce much less error than the more demanding C^1 constraints, although the resulting volume may have high-frequency discontinuities.

Filter	Freudenthal (6)		Face-Divided (12)		Face-Centered (24)	
	C^0	C^1	C^0	C^1	C^0	C^1
Gauss. 0.5	$1.25\text{x}10^{-6}$	$1.18\text{x}10^{-4}$	$2.24\text{x}10^{-7}$	$4.60\text{x}10^{-5}$	$1.29\text{x}10^{-8}$	$3.04\text{x}10^{-6}$
Gauss. 0.6	$1.74\text{x}10^{-7}$	$3.11\text{x}10^{-5}$	$2.71\text{x}10^{-8}$	$9.01\text{x}10^{-6}$	$2.35\text{x}10^{-9}$	$2.64\text{x}10^{-7}$
Gauss. 0.7	$3.35\text{x}10^{-8}$	$8.42\text{x}10^{-6}$	$5.10\text{x}10^{-9}$	$1.80\text{x}10^{-6}$	$6.19\text{x}10^{-10}$	$3.50\text{x}10^{-8}$
B-Spline	$1.42\text{x}10^{-7}$	$3.21\text{x}10^{-5}$	$2.04\text{x}10^{-8}$	$8.64\text{x}10^{-6}$	$1.94\text{x}10^{-9}$	$2.08\text{x}10^{-7}$
Catmull-Rom	$7.56\text{x}10^{-6}$	$6.53\text{x}10^{-4}$	$1.30\text{x}10^{-6}$	$2.86\text{x}10^{-4}$	$5.94\text{x}10^{-8}$	$2.28\text{x}10^{-5}$

Table 5.1. Squared error for all the filters we performed fitting on. C^0 continuity constraints cause less absolute error than C^1 constraints. In general, moving to a more-expressive κ-partition reduces error in the fitting (although it does increase complexity of the BT volume).

The Bézier spline form also limits how closely higher-frequency functions can be approximated. This can be seen in the gaussian function by examining the error as the standard deviation changes. As it gets larger, producing a wider filter with lower frequency content, the error decreases. Standard deviation 0.7 produces the lowest error of any of our filters, while 0.5 produces almost the highest error.

Anisotropic κ-partitions Figures 5.1 and 5.2 show all of the filter approximations which we used on all three κ-partitions enforcing C^0 and C^1 continuity constraints, respectively. Note that freudenthal, which is not a mirroring tetrahedral partition, produces highly anisotropic artifacts in the resulting volume due to the direction bias of weight points. The face-divided partition with 12 tetrahedra introduces less severe anisotropic bias, while the 24 tetrahedra face-centered partition, which is the only mirroring partition which we examined, does not produce any anisotropic error. The large bias introduced by C^1 continuity fitting for the Freudenthal partition suggests that this combination partition/continuity level should not be used in practice, as the artifacts it will introduce under reconstruction will be severe.

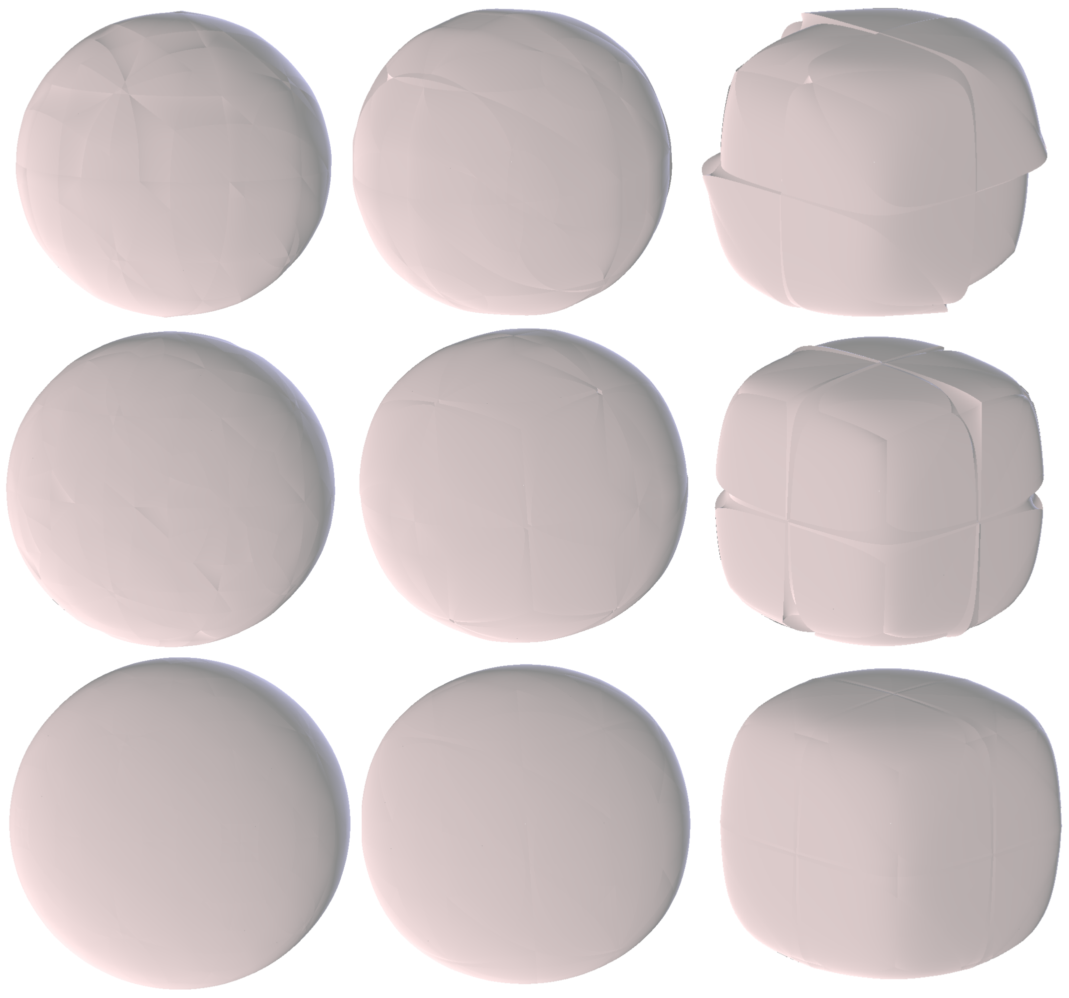

FIG. 5.1. From left to right, each column shows isosurfaces of fitting results for Gaussian with standard deviation 0.7, B-Spline, and Catmull-Rom reconstruction filters, all using C^0 continuity constraints. The sharp discontinuities, which appear obvious in the filter, are not as distracting in a volume reconstructed using the filter. Top row is the freudenthal partition, the middle row face-divided, and the bottom face-centered.

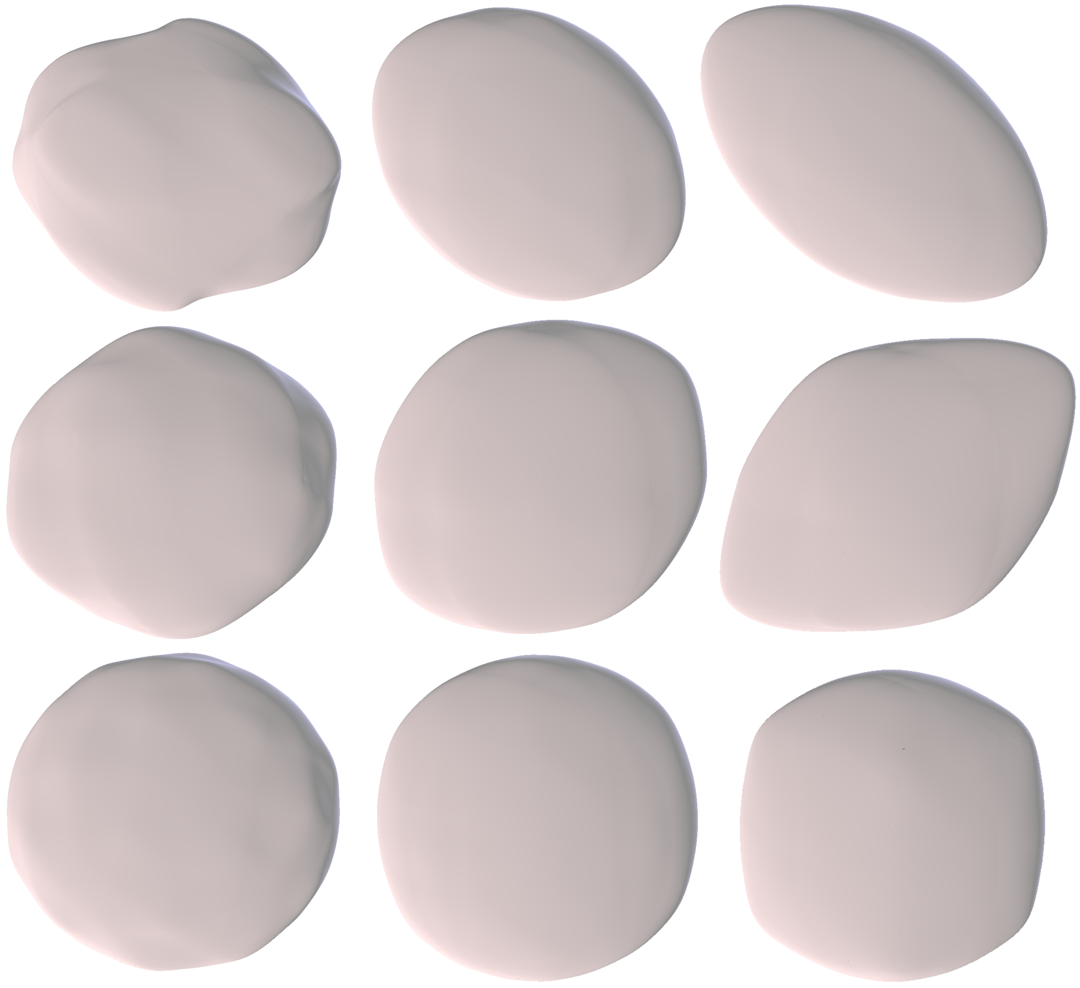

FIG. 5.2. From left to right, each column shows isosurfaces of fitting results for Gaussian with standard deviation 0.7, B-Spline, and Catmull-Rom reconstruction filters, all using C^1 continuity constraints. The directional bias of some isosurfaces is due to the anisotropic distribution of spline weights in those partitions. Top row is the freudenthal partition, the middle row face-divided, and the bottom face-centered.

Artifacts from Continuity Constraints κ-partitions composed of few tetrahedra also suffer from low-frequency artifacts which are not present in the original data set. This is because the C^1 fitting is over constraining the system, resulting in much higher error than the equivalent C^0 fitting. Figure 5.3 shows a comparison of the artifacts present when our fitting is constrained to obey C^0 vs. C^1 constraints with the B-Spline filter. Since our goal in this case is to approximate the target B-Spline filter so that reconstructions with the approximation are visually as close as possible to using the B-Spline, the raw RMS error data has limited usefulness. However, this image clearly shows the difference between C^0 and C^1 continuity constraint fitting in terms of how they affect the final reconstruction: C^0 fitting produces high-frequency discontinuities, while fitting the additional C^1 constraints trades this for smooth lower-frequency constraints. Given the fact that many applications have been using linear reconstruction, which has similar although much more pronounced artifacts than C^0, we believe that the higher-frequency artifacts are less distracting and thus C^0 fitting is adequate for most applications. This is a preliminary result, and would have to be verified by a formal user study.

5.2 BT Volume Convolution

We performed convolution on several common test volume data sets, listed in Table 5.2. The limiting factor of our method was found to be the amount of graphics memory available, which is due to the fact that each Bézier tetrahedron must be stored independently. In addition, the first rendering pass to cull out tetrahedra which do not participate in a given isosurface requires a second buffer which, in the worst case, contains the entire volume, and thus effectively doubles the memory required by our system. Finally, the two 1600x1200 backbuffer/textures that we use to actually display on the screen take up memory, along with various smaller buffers and loss due to alignment issues.

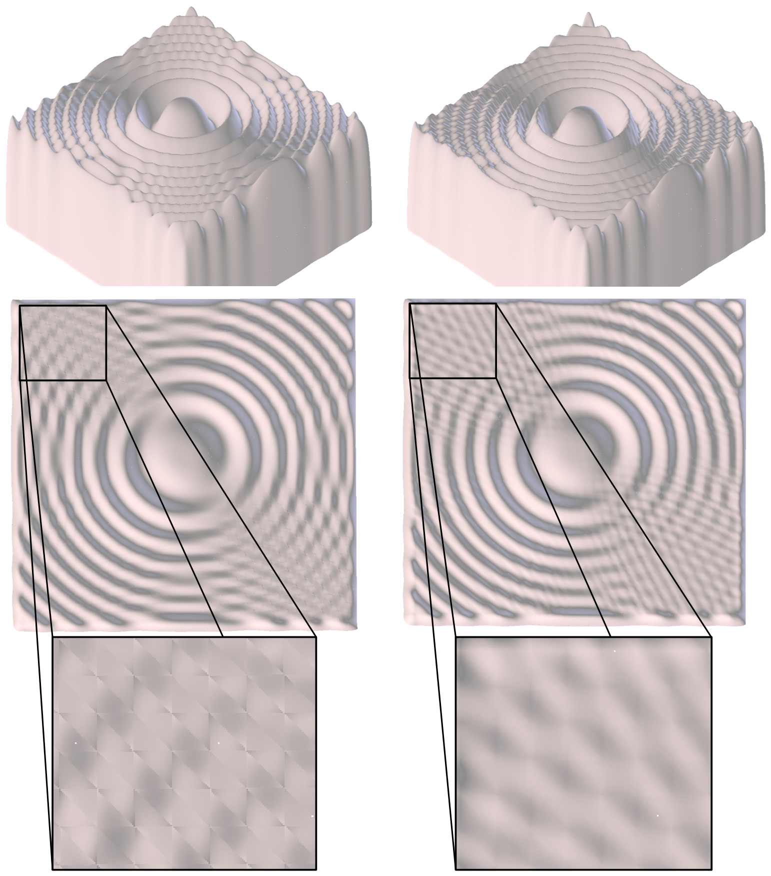

FIG. 5.3. On the left is a reconstruction using the six tetrahedra κ-partition with C^0 constraints, while the left enforces C^1 continuity. In this case it seems that C^0 would be preferable to the smoother C^1 version because the small sharp discontinuities are less distracting in the large image than the lower-frequency artifacts introduced by enforcing the smoother filter.

This leaves us with around 300MB of space for the actual volume data, given the graphics memory space in 32-bit Vista. In order to fit as large volumes as possible into that space, we perform several space optimizations for large volumes. First, we remove all tetrahedra which only contribute to isosurfaces very close to zero, which saves from %5 to almost %50 depending on the particular volume. Second, we store large volumes with 16-bit floating point weights instead of 32-bit. This gives us another %50 storage savings, but does introduce *crack* artifacts in the rendering where the precision loss affects the ray-intersection at the boundaries between tetrahedra. Figure 5.4 shows this artifact.

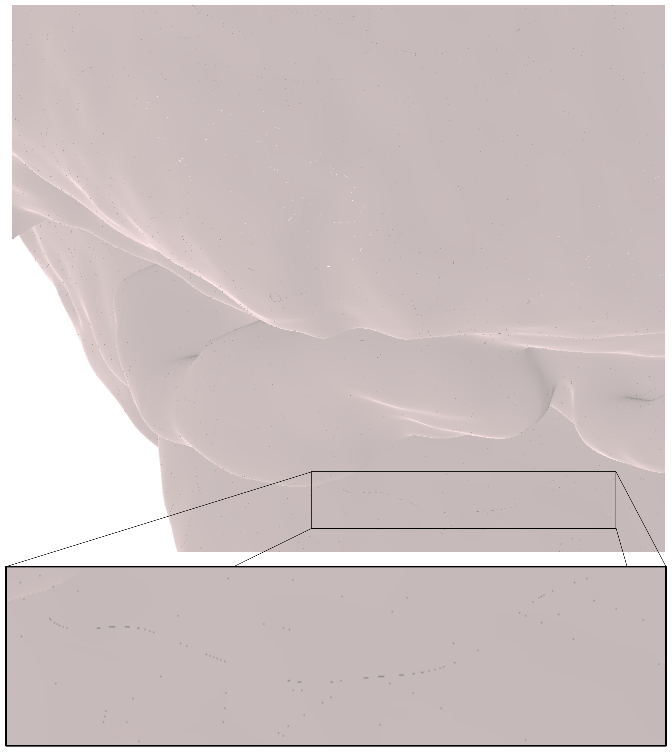

FIG. 5.4. Cracks caused by the 16-bit floating point weight values in our compressed volume format.

Figure 5.5 shows several reconstructions of the Marschner and Lobb data in our system using the 24 tetrahedra κ-function as compared to their ground-truth results using B-Spline and Catmull-Rom splines approximated with C^0 continuity. Although our system is only generating approximations which are not even guaranteed to be continuous, the results are visually almost identical.

Figure 4.5 shows four reconstructions of a Buckyball data set with Gaussian (st. dev. 0.7), B-Spline, and Catmull-Rom BT volume reconstructions and marching cubes. The marching cubes reconstruction shows severe linear artifacts which are not present in the

BT volume reconstructions.

Data Set	foot†	engine†	teddy†	bucky	M&L Test Signal	neghip
Size	128^3	128^2x64	128^2x62	32^3	41^3	64^3
MB	230	157	330	21	37	165
FPS	10	10	8	80	44	14

Table 5.2. Rendering performance and data sizes for our test volumes. Volumes marked with a † remove tetrahedra which only contributed to very small isosurfaces. All size and performance measurements are for the 6-tetrahedron κ-partition rendered on a 1600x1200 resolution desktop.

5.3 Rendering

We ran our system on a quad-core Intel processor with 2 gigabytes of RAM and an ATI 4870x2 graphics card with 2 gigabytes of RAM. All tests were run at 1600x1200 resolution on 32 bit Windows Vista. Due to the memory limitations of the 32-bit OS, we only had one gigabyte of graphics RAM available to us. Table 5.2 shows a summary of performance and size requirements for several volumes at 1600x1200 screen resolution. Volumes marked with a † used 16-bit floating point weight values to reduce size. Figure 5.7 shows several volumetric data sets rendered in our system.

5.3.1 Scalability and Bottleneck Identification

In order to determine if the system is pixel limited or memory bandwidth limited, we performed several performance tests at different resolutions. Changing the resolution of the renderer leaves the memory bandwidth requirements unchanged (because the volume data is not changing) but increases the number of pixels which have to be filled, testing whether the pixel shader is the bottleneck in the rendering algorithm. On all of the volumes which we tested, there was a maximum of one frame-per-second difference between the

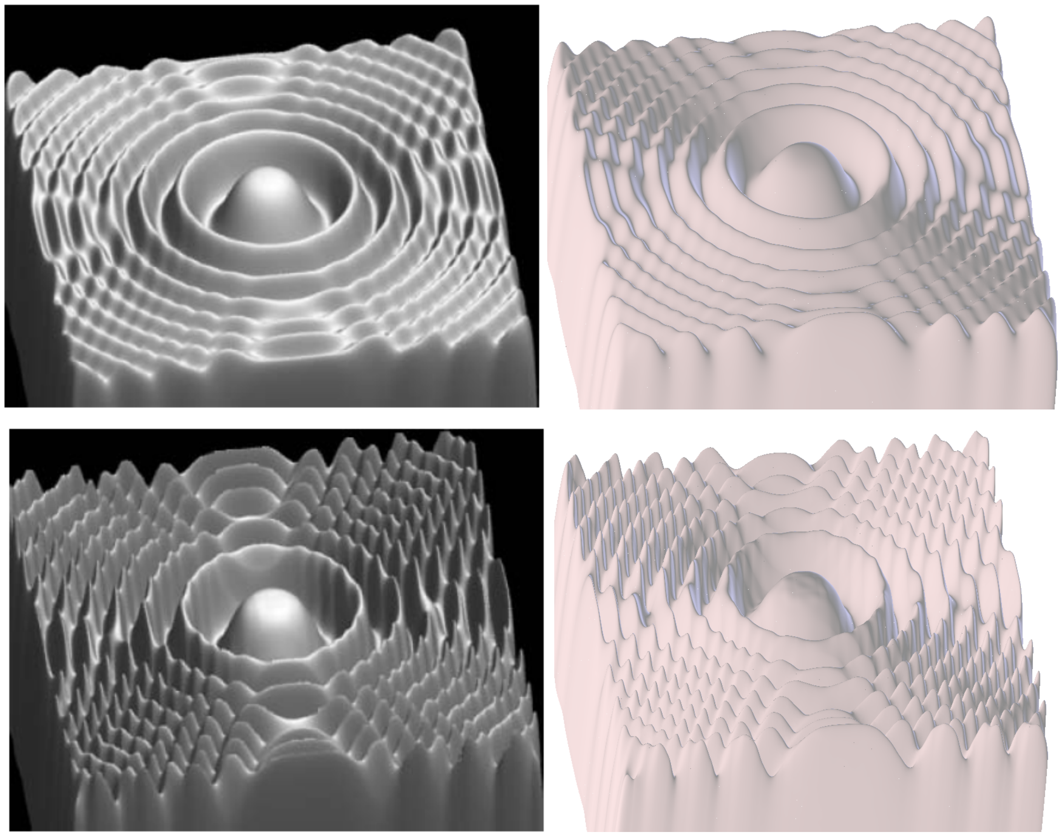

FIG. 5.5. Various reconstructions of the Marschner and Lobb test data set. On the left is the ground truth result from Marschner & Lobb (1994), with our result using the 24 tetrahedron κ-partition on the right. The top row is for B-Spline reconstruction, while the bottom row shows Catmull-Rom reconstruction. For these results, we used C^0 continuity.

640x480 resolution, which was the smallest that we tested, and the full-screen 1600x1200 resolution. This means that, even though we are performing expensive root finding in the pixel shader, the bottleneck of our rendering is internal data bandwidth on the GPU, so reducing the amount of data passed around would be beneficial.

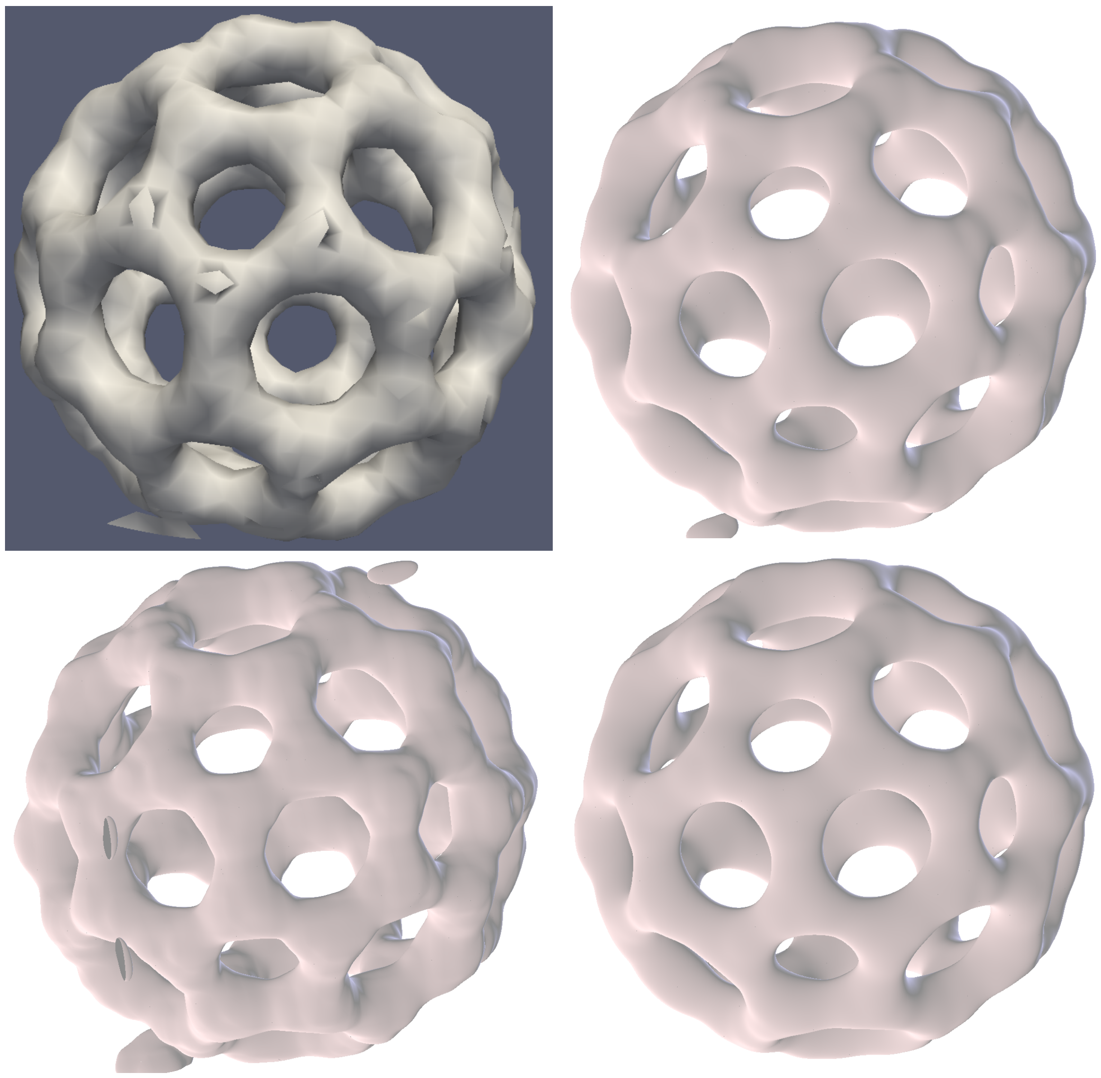

FIG. 5.6. Four isosurface reconstructions of a Buckyball data set. From upper left clockwise, marching cubes, BT volume B-Spline, BT volume Catmull-Rom, BT volume gaussian with st. dev. of 0.7. All BT volume reconstructions used 24 tetrahedra κ-partition and C^0 continuity.

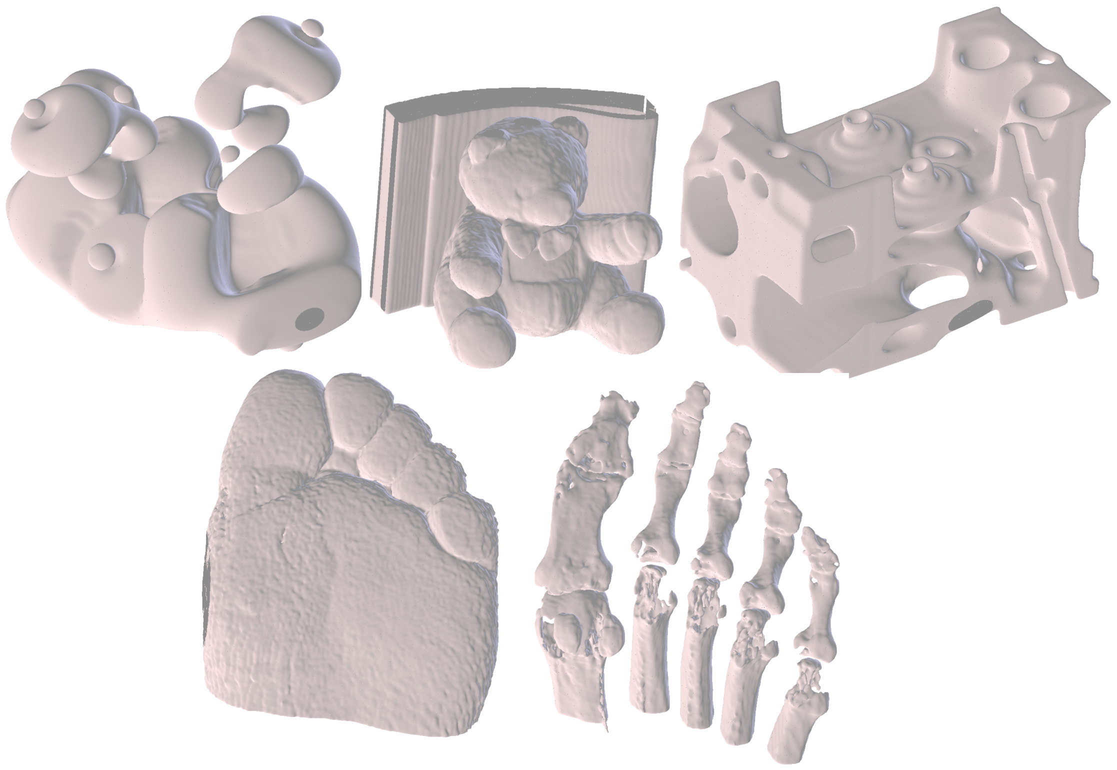

FIG. 5.7. Test data sets rendered using our method. From top to bottom, left to right: simulation of electron density of a protein molecule, scan of a teddy bear, scan of an engine block, and a scan of a foot rendered for density of skin and bone.

Chapter 6

CONCLUSION

We have presented the BT volume as a volume representation which provides a solution to the difficult problem of rendering high-quality isosurfaces. The BT volume is defined as a 3D super spline on a special type of tetrahedral partition which we call a κ-partition. As a proof of concept, we have implemented three different κ-partitions in our work, consisting of 6, 12, and 24 tetrahedra, respectively. Because they are piecewise-continuous polynomials, BT volumes can approximate arbitrary 3D functions, and we have demonstrated how to generate such approximations using a least-squares technique.

We also demonstrated how to produce a BT volume as the result of convolution of a BT volume-format reconstruction filter and an arbitrary regular scalar-valued volume. We compute BT volume-format filters using the least squares technique, showing that the convolution result is exactly another BT volume. We have demonstrated this technique for several example volumes and filters, including Gaussian, Catmull-Rom, and B-Spline reconstruction filters. We showed the advantages of this system as compared to many other volume rendering systems, including the facts that BT volumes are continuous, smooth representations and that they have the ability to represent many different reconstructions, including large footprint reconstruction filters, with little additional cost.

BT volumes can also be rendered efficiently on current graphics hardware. We presented a complete rendering algorithm, along with performance results from our proof-of-concept implementation. First, all spline primitives in the super spline which actually intersect the desired isosurface are extracted. Each extracted spline is represented as a single vertex, which is expanded into screen-space triangles which cover all possible contribution from this spline. Finally, all pixels covered by the primitive are filled by performing exact ray-intersection and outputting the normal if there was an intersection for that pixel. Shading is performed in a deferred manner in a separate pass.

The BT volume format achieves a balance between rendering speed and rendering quality which has not been achieved before for isosurfaces. Our results are flexible because we support multiple reconstruction filters and fast because we can map efficiently onto graphics hardware. Finally, because we separate the rendering and convolution steps of volume rendering, we can achieve very high quality results.

Bibliography

Anderson, J. C.; Bennett, J.; and Joy, K. I. 2005. Marching diamonds for unstructured meshes. In *IEEE Visualization 2005*, 423–429.

Blinn, J. 2003. *Jim Blinn's Corner*. Morgan Kaufmann.

Blinn, J. F. 2006. How to solve a cubic equation, part 1: The shape of the discriminant. *IEEE Comput. Graph. Appl.* 26(3):84–93.

Blythe, D. 2006. The direct3d 10 system. *ACM Trans. Graph.* 25(3):724–734.

Carr, H.; Moller, T.; and Snoeyink, J. 2006. Artifacts caused by simplicial subdivision. *IEEE Transactions on Visualization and Computer Graphics* 12(2):231–242.

Cohen, J. D. 2006. Projected tetrahedra revisited: A barycentric formulation applied to digital radiograph reconstruction using higher-order attenuation functions. *IEEE Transactions on Visualization and Computer Graphics* 12(4):461–473. Student Member-Ofri Sadowsky and Fellow-Russell H. Taylor.

Danskin, J., and Hanrahan, P. 1992. Fast algorithms for volume ray tracing. In *VVS '92: Proceedings of the 1992 workshop on Volume visualization*, 91–98. New York, NY, USA: ACM.

DeMarle, D. E.; Parker, S.; Hartner, M.; Gribble, C.; and Hansen, C. 2003. Distributed interactive ray tracing for large volume visualization. In *PVG '03: Proceedings of the*

2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics, 12. Washington, DC, USA: IEEE Computer Society.

Gerstner, T., and Rumpf, M. 1999. Multiresolutional parallel isosurface extraction based on tetrahedral bisection. In *in Proc. VolVis99*, 267–278. Press.

Goetz, F.; Junklewitz, T.; and Domik, G. 2005. Real-time marching cubes on the vertex shader. In *Eurographics 2005 short presentations*. Eurographics Association.

Gooch, A.; Gooch, B.; Shirley, P.; and Cohen, E. 1998. A non-photorealistic lighting model for automatic technical illustration. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, 447–452. New York, NY, USA: ACM.

Johansson, G., and Carr, H. 2006. Accelerating marching cubes with graphics hardware. In *CASCON '06: Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*, 39. New York, NY, USA: ACM.

Klein, T.; Stegmaier, S.; and Ertl, T. 2004. Hardware-accelerated reconstruction of polygonal isosurface representations on unstructured grids. In *PG '04: Proceedings of the Computer Graphics and Applications, 12th Pacific Conference*, 186–195. Washington, DC, USA: IEEE Computer Society.

Kloetzli, J.; Olano, M.; and Rheingans, P. 2008. Interactive volume isosurface rendering using bt volumes. In *SI3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, 45–52. New York, NY, USA: ACM.

Levoy, M. 1988. Display of surfaces from volume data. *IEEE Comput. Graph. Appl.* 8(3):29–37.

Loop, C., and Blinn, J. 2006. Real-time gpu rendering of piecewise algebraic surfaces. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, 664–670. New York, NY, USA: ACM Press.

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

Marschner, S. R., and Lobb, R. J. 1994. An evaluation of reconstruction filters for volume rendering. In *VIS '94: Proceedings of the conference on Visualization '94*, 100–107. Los Alamitos, CA, USA: IEEE Computer Society Press.

Parker, S.; Martin, W.; pike J. Sloan, P.; Shirley, P.; Smits, B.; and Hansen, C. 1999a. Interactive ray tracing. In *In Symposium on interactive 3D graphics*, 119–126.

Parker, S.; Parker, M.; Livnat, Y.; Sloan, P.-P.; Hansen, C.; and Shirley, P. 1999b. Interactive ray tracing for volume visualization. *IEEE Transactions on Visualization and Computer Graphics* 5(3):238–250.

Pascucci, V. 2004. Isosurface computation made simple: hardware acceleration, adaptive refinement and tetrahedral stripping. In *In Joint Eurographics - IEEE TVCG Symposium on Visualization (VisSym*, 293–300.

Reck, F.; Dachsbacher, C.; Stamminger, M.; Greiner, G.; and Grosso, R. 2004. Realtime isosurface extraction with graphics hardware. In *Eurographics 2004 - short presentations and interactive demos*. Eurographics Association.

Rossl, C.; Zeilfelder, F.; Nurnberger, G.; and Seidel, H.-P. 2003. Visualization of volume data with quadratic super splines. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, 52–60. Washington, DC, USA: IEEE Computer Society.

Sadowsky, O.; Cohen, J. D.; and Taylor, R. H. 2005. Rendering tetrahedral meshes with higher-order attenuation functions for digital radiograph reconstruction. In *In Proc. of IEEE Visualization*, 303–310.

Shirley, P., and Tuchman, A. 1990. A polygonal approximation to direct scalar volume rendering. *SIGGRAPH Comput. Graph.* 24(5):63–70.

Sramek, M., and Kaufman, A. 2000. Fast ray-tracing of rectilinear volume data using distance transforms. *IEEE Transactions on Visualization and Computer Graphics* 6(3):236–252.

Tatarchuk, N.; Shopf, J.; and DeCoro, C. 2007. Real-time isosurface extraction using the gpu programmable geometry pipeline. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, 122–137. New York, NY, USA: ACM.

Theisel, H. 2002. Exact isosurfaces for marching cubes. In *Computer Graphics Forum*, 19–31. Blackwell Publishers for Eurographics Association.

Treece, G. M.; Prager, R. W.; and Gee, A. H. 1999. Regularised marching tetrahedra: improved iso-surface extraction. *Computers and Graphics* 23(4):583–598.