SCALABLE ISOSURFACE VISUALIZATION

Zhiyan Liu

A DISSERTATION

PRESENTED TO THE FACULTY

OF PRINCETON UNIVERSITY

IN CANDIDACY FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE

BY THE DEPARTMENT OF

COMPUTER SCIENCE

Advisor: Kai Li

June 2008

Abstract

Scientific data visualization assists users in understanding and analyzing large volumes of raw data by converting them into graphic representations. The main challenges in building a visualization system are large datasets, high-resolution displays, and the need to visualize remote datasets over slow networks. This dissertation focuses on visualizing isosurfaces efficiently under these challenges.

First, to visualize large datasets efficiently on high-resolution displays, this dissertation proposes a new isosurface extraction algorithm that generates approximate results much faster than previous algorithms. The algorithm extracts portions of the isosurface in a view-dependent manner by ray casting and propagation. It casts rays through a volume to find visible active cells as seeds and then propagates their polygonal isosurface into the neighboring cells. Small pieces of the isosurface are generated by distance-limited propagation and joined together to form the final surface. Evaluation shows that this progressive algorithm generates an approximate result quickly before the final correct image is reached over time. In addition, the algorithm scales with the resolution of the display and supports adaptive-resolution visualization.

This dissertation further discusses design options that are used for a fast implementation of the base algorithm.

Finally, this dissertation presents a simple and effective protocol for remote isosurface visualization. The protocol breaks up the isosurface visualization pipeline at the 3D primitive stage for the server to send primitives to the client. Several techniques are introduced to reduce communication requirement and to provide interactive visualization. The server runs an efficient isosurface extraction algorithm that generates view-dependent portions of the isosurface. The resulting 3D primitives are organized into groups and sent to the client to be rendered. This approach uses primitive compression, progressive level-of-detail, and primitive caching techniques to improve the interactivity perceived by the user. This dissertation reports and compares the experiment results under different settings to show the effectiveness of these techniques. The results show that, when applied together, the techniques reduce both the amount of data transferred and time spent when the result becomes 90% correct by two orders of magnitude.

Acknowledgements

If you view graduate study as a long journey, the completion of the dissertation is like reaching the destination of your journey.

Last summer I through-hiked the Long Trail, the first long distance trail in the United States. Running north-south along the whole length of the Vermont state, the trail is two hundred and seventy five miles long. It took me nineteen consecutive days to finish it. When I stood at its northern terminus on the Canadian border, I felt immense joy, a huge sense of accomplishment, and a deep gratitude toward the people who made my journey possible. Just like how I feel today.

When I look back at my through hike, I identified three factors for my success: strong body, strong will, and strong support. Although experienced hikers can finish the trail with minimal support, to me strong support is essential. My parents had understood and blessed my mission. My husband and friends had come frequently to provide transportation and hike with me part of the time, carrying many of my gears so that I could recover from a low level pain in the right knee. At one point, a drought dried many water sources and I even thought of stopping there, but a day hiker gave me all her water

and I was able to reach the next major stream. I know I could not have made it without the support of my family, friends, and strangers.

And so it is with my dissertation. I can imagine that for those who are truly gifted and self-driven, the graduate degree may be achieved without too much help from the others. However for someone who is in want of talent and discipline like me, it takes every bit of my mother's loving nagging, my father's tacit urging, my advisor's guidance, my collaborators' help and my friends' encouragement to get here.

I have been extremely fortunate to work with my advisor, Professor Kai Li. Kai's wisdom, insights, and charisma has guided and influenced me throughout the years. He has shown me by example what a great researcher and teacher should be like. Discussions and conversations with Kai never fail to motivate me. Were it not for his patience and persistence, this dissertation would have been impossible.

I would like to express my deep appreciation to my committee members: Professors Adam Finkelstein, Tom Funkhouser, Olga Troyanskaya, and Szymon Rusinkiewicz. Your advice and support is invaluable to me. A special thanks goes to Professor Brian Kernighan, for reminding me to finish my writing every time we met. It had worked.

I would like to thank my collaborators in the Display Wall project: Grant Wallace, Han Chen, Yuqun Chen, Rudro Samanta, and Allison Klein. To my friends Jessica Fong, Limin Jia, Lena Petrovic, Zaijin Guan, Akihiro Nakao, Matthew Webb: my graduate years would have been unbearable without your friendship.

I also want to thank our Graduate Coordinator Melissa Lawson. You saw me through from admission to graduation. Your efficient and calm manner made the paperwork that much less formidable.

This work was supported in part by Intel Corporation, the National Science Foundation (under grants CDA-9624099, EIA-9975011, CNS-0406415, and EIA-0101247) and Department of Energy (under grants ANI-9906704 and DE-FC02-99ER25387). I am also grateful to the management at Google, Inc. for giving me time to work on the dissertation.

I dedicate this dissertation to my parents and my husband, Ruoming Pang, for always being there for me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Isosurface Visualization

Scientific visualization is a process of presenting numerical data in graphical forms, such as images, animations and charts. It is a widely used technique that aids users in data analysis, pattern discovery, and hypothesis verification. In many cases the underlying data are *scalar fields* either acquired or computed in three dimensional (3-D) spaces. For example, medical imaging methods such as computed tomography (CT) and magnetic resonance imaging (MRI) measure density distribution in the human body; weather forecasting uses numerical climate modeling to simulation temperature, humidity, and pressure distributions over time.

In general, a *3-D scalar field* is a continuous R^3→R function with the 3-D space as its domain and scalar values as its range. A scalar field *F* is usually represented by a

volumetric dataset D, which records function values at sample points. The function values at all other points are estimated by interpolation.

The need for volumetric data visualization first arose from medical imaging applications. CT and MRI produce multiple two dimensional cross-sectional tissue radiodensity data. To use this data in diagnosis, in the past radiologists had to interpret imaging results on a 2-D slice-by-slice basis. This non-intuitive method has several limitations. It requires substantial expertise and is time consuming. Later, different 3-D visualization techniques were invented. They provide significantly better support for analysis and diagnosis in medial imaging [9]. Most of these 3D approaches fall into two main categories. One is *volume rendering*, where values are mapped into colors and opacity levels and the whole volume is rendered into a 2-D image. The other category, *isosurface visualization*, extracts isosurfaces and displays them in 3-D.

An *isosurface* is the set of all the points that have the same scalar value, which is called the *isovalue* or *threshold*. With a carefully selected threshold, an isosurface in a medical dataset marks material boundary such as skin or bone structures. Users can interactively control isosurface visualization by adjusting the threshold and changing the view point, and the underlying visualization system extracts and renders the isosurface.

Although both volume rendering and isosurface extraction are widely used and sometimes used in conjunction in volume visualization, in this dissertation we focus on isosurface visualization and investigate the problem of fast isosurface visualization of large and remotely hosted datasets on a scalable display.

1.2 Challenges and Goals

There are three main challenges in designing an isosurface visualization system:

Large datasets

A volumetric dataset can contain hundreds of millions of data points .The National Library of Medicine's Visible Man Computed Tomography (CT) dataset has 512×512×522 data points. Its later companion dataset, the Visible Woman dataset, has a higher resolution of 512×512×1734. This reflects a trend of data explosion in the scientific community. CERN, the European Organization for Nuclear Research, has petabytes of data from particle physics experiments.

As Bell et al [4] pointed out, most scientific disciplines now have computational branches to simulate and solve complex models. Datasets generated from simulations can be arbitrarily large, and they often include a fourth dimension, time. Visualization of these datasets often involves extracting isosurfaces of a certain value from each time step and rendering them in series. This places an even higher requirement on the time complexity of the visualization system.

Moreover, because the typical behavior of a user is to first adjust the isovalue until a significant threshold is found, it is very important for the algorithm to produce a rough approximation of the surface in an interactive time frame.

Scalable display

Historically display resolution increases at a far slower pace than computational power or storage capacity. However, emerging scalable high-resolution display surfaces, such as the Display Wall [22, 40] and the Hyperwall [36] use a cluster of computers to drive an

array of projectors or flat panel LCD screens. This technology increases screen size by two orders of magnitude and resolution by one order of magnitude. The GigaPixel project at Virginia Tech [1] is currently building a 5×10 touch screen LCD wall that has approximately 100 million pixels. This makes showing all the fine details in a large dataset at once a possibility. Studies [3, 32] show that this type of display surface provides a less stressful user experience and a better sense of confidence and makes users more effective at task performance.

To use tiled high-resolution display, however, requires the underlying visualization algorithm to be insensitive to the number of pixels.

Distributed infrastructures

Early data visualizations are performed locally and this greatly limits their accessibility. A remote user has to download the dataset and install the visualization tool before he can start visualization. Large datasets exacerbate the problem because the computation complexity often exceeds an average desktop's capabilities. This creates a high barrier for the users. Given that the datasets are usually hosted at an institution with access to computing clusters, preferably the visualization should be done in a distributed fashion between servers, which reside at data hosting sites, and client systems close to the users. The server extracts isosurfaces while the client renders them. Ideally, if the client-side rendering system can be implemented inside a web-based user interface, remote data visualization can be as simple as browsing a web page from a user's point of view. This easy accessibility will benefit a very large user base.

Thus the goal of this dissertation is to provide interactive isosurface visualization for very large datasets on scalable display systems that are located remotely from datasets and computing resources.

1.3 Background and Related Work

1.3.1 Terminology

Volumetric datasets are classified as *structured* or *unstructured* depending on the distribution of the sample points. If the coordinates of the sample points can be generated by a function, the dataset is *structured*; otherwise it is *unstructured*. CT and MRI datasets are often points on three dimensional Cartesian grids. Other grid shapes such as curvilinear are also used in physical simulations.

Each value at a sample point is called a *volumetric element* (*voxel*). It is the three dimensional counterpart of a pixel. For a cuboid dataset, the size (*resolution*) is expressed by $Nx \times Ny \times Nz$, where Nx, Ny, and Nz are the numbers of voxels in each direction.

The volume is divided into non-overlapping sub-regions called *cells*. Each cell is a convex hull defined by neighboring voxels. Cells do not contain internal voxels. In a structured dataset, cells are (warped) hexahedra. In an unstructured dataset, cells can have an array of forms, including tetrahedra, pyramids, etc. Depending on the definition, two cells are said to be neighbors if they share a face or an edge. For a structured dataset, the coordinates of voxels, the subdivision of the volume into cells, and the neighboring relationship among cells are all implicit. For an unstructured dataset, this information is explicitly given and thus requires more storage and processing time.

In structured datasets, cells can be grouped into *meta-cells*. A meta-cell is often a larger (warped) hexahedra spanning *Kx*, *Ky*, and *Kz* cells in the x, y, and z directions. It is called a *Kx*×*Ky*×*Kz* meta-cell. Meta-cells provide a hierarchical way of organizing a dataset.

This dissertation focuses on structured rectangular datasets. The algorithm can be adapted to work with unstructured datasets.

1.3.2 The Marching Cubes Algorithm

Isosurface extraction is a well-studied problem. Early attempts [7, 11, 20] compute isocontours on two dimensional slices and connect them together with triangles. However, user intervention is sometimes required to disambiguate connections. Independently proposed by Lorensen and Cline [26] and Wyvill et al [43], Marching Cubes is probably the most famous isosurface extraction method and the base for numerous optimizations. Unlike previous attempts to produce the isosurface globally, Marching Cubes uses a divide and conquer approach to attack the problem. It solves the problem of constructing the portion of an isosurface in one cell. The algorithm processes each cell sequentially, in scan-line order. After all the cells are processed, the complete isosurface is produced.

The original Marching Cubes works on structured datasets with cuboid cells. With some adaptation it can work on curvilinear datasets and unstructured datasets.

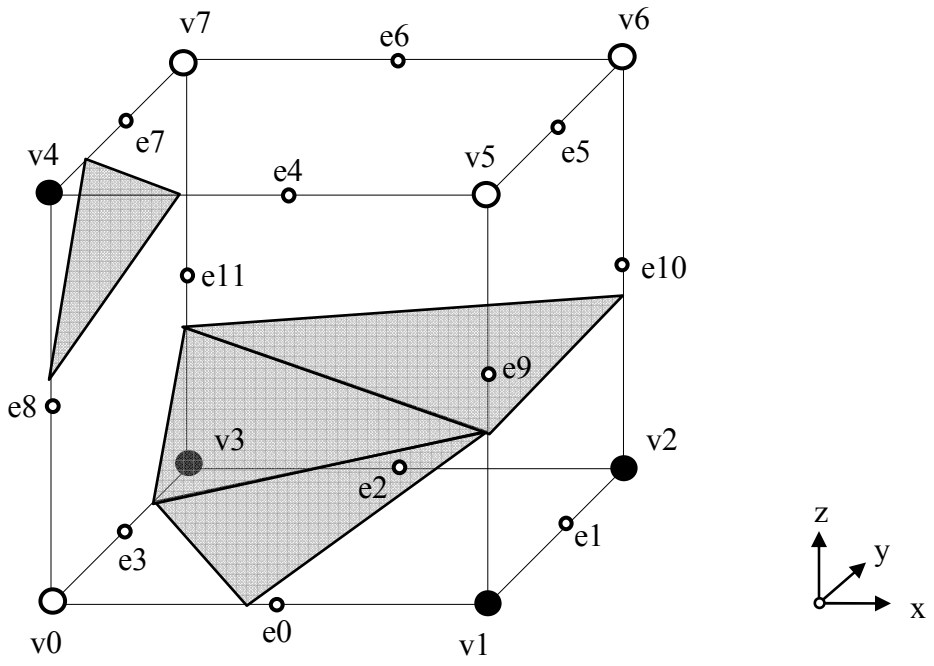


Figure 1.1: Marching Cubes applied to one cell.

A cuboid cell *C* has eight vertices. We number them 0 through 7 as shown in Figure 1.1. Each vertex *Vi* has a value *F(Vi)* associated with it. Compared with the isovalue *v*, let *Mi* = 0 if *F(Vi)* < *v* and = 1 if *F(Vi)* > *v*. We mark the vertices with *Mi* = 1 with a black dot in the graph. $M_7M_6M_5M_4M_3M_2M_1M_0$ is an eight-bit vector that defines the topology of the isosurface inside the cell. If *M* = 00000000 or 11111111, the isosurface does not pass through *C*. Otherwise *C* contains a portion of the isosurface and we say it is an active cell. In our example, *M* = 00011110 and the isosurface inside the cell is approximated by the gray triangles.

Using complementary and rotational symmetry, the 256 possible values of *M* can be reduced to 15 patterns. Marching cubes uses a tessellated triangle mesh to approximate the isosurface. A triangle topology table is pre-computed for each value of *M*. Every

vertex lies on an edge where the voxel pair brackets the threshold. The coordinates of vertices are computed by linear interpolation. Each vertex is also associated with a normal for shading purposes. The normal is an interpolation between the normalized gradients of the two neighboring voxels.

Much work has been done to amend and extend the Marching Cubes idea. [27, 33] address the ambiguity problem in Marching Cubes. Instead of interpolating between voxels, Discretized Marching Cubes [29] uses midpoints of cell edges as the triangle vertices, thus reduces the number of triangles at the price of approximation accuracy. Marching Tetrahedra and Marching Triangles [15] are variants that improve the performance of Marching Cubes. Many papers [2, 30, 31, 39] propose using Marching Cubes and its variants adaptively to generate surfaces where fewer triangles are used at areas with lower curvature.

1.3.3 Optimization for Marching Cubes

We define the *interval* (*span*) of a cell C to be [*minv*, *maxv*], where *minv* and *maxv* are the minimum and maximum of values on C's vertices, respectively. C is said to be *active* if and only if $minv < v < maxv$.

For a typical (non-pathological) $N \times N \times N$ dataset, the number of active cells is $O(N^2)$. This means the majority of cells are not active, or uninteresting. However, since the user-specified isovalue can be arbitrary, no cell can be determined to be uninteresting a priori. Marching Cubes scans every cell and spend a large percentage of time on filtering out uninteresting cells. Because the uninteresting cells do not contribute to the output, it is important to reject them as quickly as possible.

Optimizations for Marching Cubes mostly aim to reduce or eliminate the filtering overheads. They can be classified into two categories: space-based and range-based.

Space-Based Approaches

Octree-based isosurface extraction [13] uses spatial decomposition of the volume to decrease filtering overheads. An octree is a tree where each internal node has eight children. In the octree organization of a volumetric dataset, every node represents a (meta-) cell. The root node is the whole volume, and leaf nodes are individual cells. For an internal node, the planes $x=Cx$, $y=Cy$, and $z=Cz$ divides its volume into eight sub-volumes, each of which is represented by a child node.

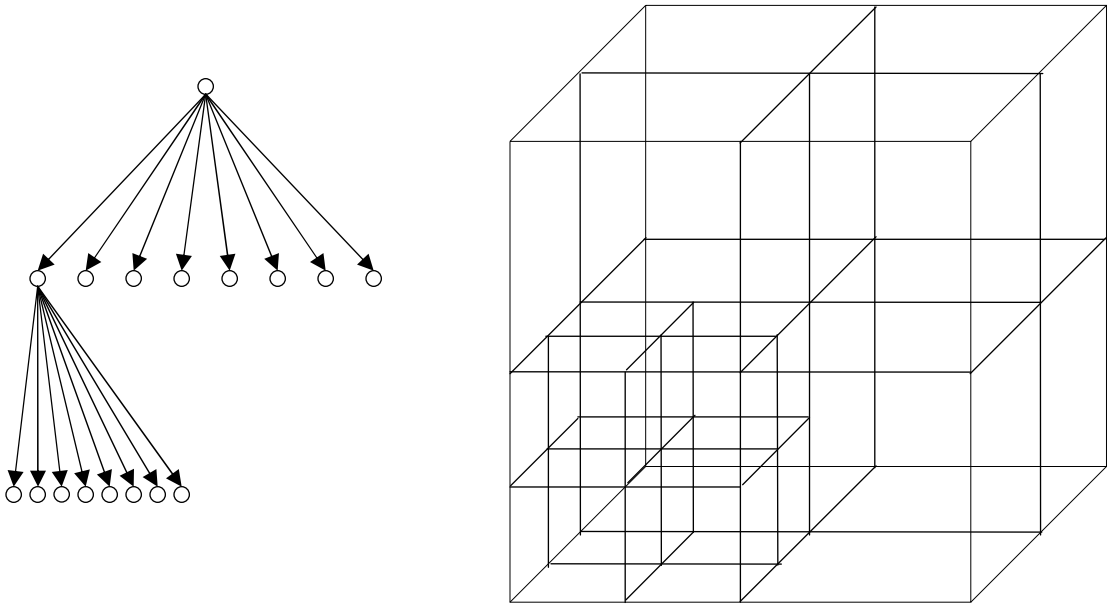


Figure 1.2: An octree.

Octree defines a spatially hierarchical way of organizing cells. Using a bottom-up approach, every internal node's interval can be computed as the union of its children's intervals. During isosurface extraction, the octree is examined from the top down. If an

internal node's interval is uninteresting, the whole corresponding meta-cell can be discarded, thus eliminate the need to examine every cell in there.

Wilhelms and Van Gelder proposed Branch On Need Octree[42] (BONO) to reduce the storage requirements for the additional data structure. For a structured dataset, the storage overhead is about 16% of the original dataset size.

Octrees can also be used on unstructured datasets, but the preprocessing time and storage overhead are both much higher.

Space-based methods work well when the underlying field has small gradients. In the case where the field has substantial local variation, the majority of filtering overheads cannot be avoided.

Range-Based Approaches

Unlike space-based approaches, range-based approaches disregard any spatial proximity information. Instead, they organize cells according to their value proximity.

Using a cell's span (*minv*, *maxv*) as its coordination in the two-dimension space, we can plot all the cells in a dataset as dots in the min-max plane, as shown in Figure 1.3. The active cells are those in the shaded area defined by $min <= v$ and $max >= v$.

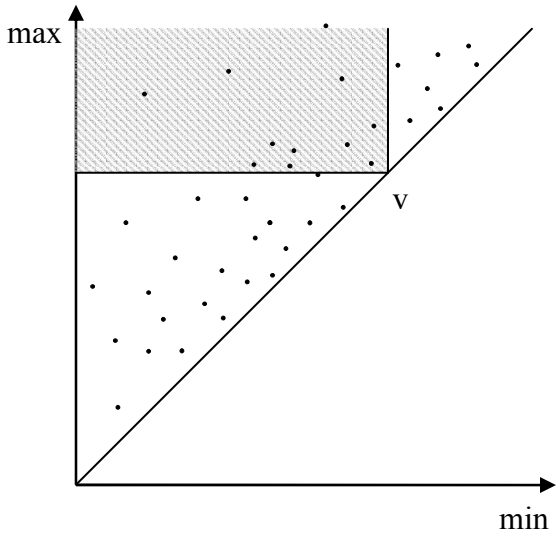


Figure 1.3: Range-based view of active cells.

Range-based approaches aim to find the active cells quickly. Cignoni *et al*. proposed the interval-tree method [8] and Livnat *et al*. proposed to use span space [25]. In preprocessing, both of these algorithms sort all the cells according to minimum and maximum values and construct a search tree; then, for a given threshold, these methods search the tree to find all the active cells.

Range-based approaches only visit the active cells. The fact that they do not rely on spatial information makes them preferable in processing unstructured datasets.

1.3.4 Propagation-Based Algorithms

In an active cell, at least one face has values both above and below the threshold. Therefore the neighboring cell that shares this face is also active. Starting from one active cell (the *seed cell*), a breadth first search (BFS) can find all the connected active cells, and produce a connected component of the isosurface.

Based on this fact, propagation-based algorithms [18, 19] focus on finding at least one seed per connected component for any given isovalue. Kreveld et al. proposed an approximation algorithm [21] for finding a seed set that is utmost twice the size of the minimum seed set.

Propagation-based approaches avoid all filtering overheads, but the processing time can be long. In extreme cases such as a field where neighboring voxels alternate between 0 and 1, the seed set size can be O(n), where n is the number of voxels.

1.3.5 View Dependent Algorithms

All the previously mentioned methods generate the whole isosurface. With rapidly growing dataset size and near constant display resolution, the visible part of the isosurface at any given time is becoming increasingly smaller. View-dependent algorithms only produce the visible portion of the isosurface, thereby making the viewing process more interactive.

Pixel-Based Approaches

Ray tracing is a method often used in realistic rendering. Parker et al. [35] applied this method to isosurface visualization. For each pixel in the display, a ray is cast into the volume and the first active cell that intersects the ray is found. Inside the cell, trilinear interpolation is used to calculate the shading of the pixel. This approach has a low time complexity but a high time constant. By using a shared memory 128-processor SGI Reality Monster machine to parallelize the process, interactive frame rate (10FPS) was achieved on the full-resolution Visible Woman dataset.

Primitive-Based Approaches

Livnat and Hansen proposed an approach [24] that outputs only the visible triangles (primitives) in the isosurface. The algorithm organizes both the volume space and the screen space hierarchically. Traversing the dataset in a front-to-back order, (meta-) cells are projected onto the screen to test for visibility. Invisible (meta-) cells are discarded. Screen masks at different resolution are used to keep track of visibility. Visibility tests are first done against the coarsest mask, and progressed to finer masks till visibility can be determined.

This approach is dependent on the resolution of the display. Also the output surface may have holes, making it hard to cache the results for later use.

1.3.6 Remote Algorithms

Remote data visualization is important for many applications because acquisition machine, storage server, feature extraction engine, rendering machine, and display are often distributed across a network. The challenge is to build an effective end-to-end remote data visualization system for applications with massive datasets. Massive datasets require a high-resolution display to visualize and analyze, which in turn requires a large number of primitives generated, transferred, and rendered across the network during remote data visualization.

A good remote data visualization method should extract the isosurface efficiently; require only modest network bandwidth; and support interactive, progressive, adaptive-resolution volume visualization on a high-resolution display system.

Engel et al. offers an excellent breakdown [10] of the marching-cubes-like isosurface reconstruction pipeline and taxonomy of different remote isosurface visualization scenarios. In general, the process of geometry-based isosurface visualization can be viewed as a pipeline, as shown in Figure 1.4. First, the dataset is taken out of storage and fed into a filter that identifies the active cells. Next, the active cells are processed and the triangulated approximation of the isosurface is generated. These 3D primitives are then projected, rasterized, and displayed on the screen.

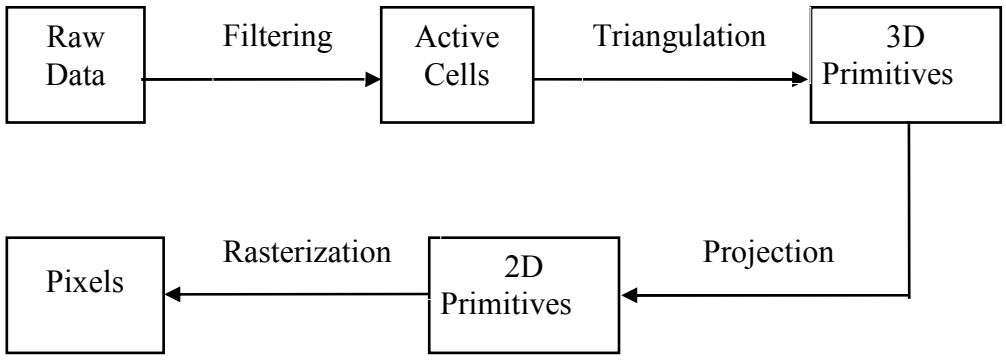


Figure 1.4: The isosurface visualization pipeline.

Engel et al. investigated three different scenarios in remote isosurface visualization. The intermediate representations sent across the network are 3D primitives, partially triangulated active cells, and active cells respectively. The whole isosurface is transferred for all scenarios; this may increase the latency when a new isovalue is used.

Heiland et al. proposed VisBench [14], a flexible component-based system that provides a framework for remote data visualization and analysis. The primary visualization engine is based on the Visualization Toolkit (VTK). The back end server

performs both extraction and rendering and outputs compressed images to the client. This approach does not scale with screen resolution.

Bertel et al. developed Visapult [5, 6], a remote and distributed visualization system for IBR (Image Based Rendering)-assisted volume rendering. In the system, multiple servers perform volume rendering and send the resulting images to a multi-threaded client for use as 2D textures in rendering a scene graph. This system has high data transfer rates and requires a high bandwidth network.

Numerous techniques exist for 3D mesh simplification and compression, e.g. [12, 37]. Hoppe [16, 17] proposed the progressive mesh scheme. A mesh can be progressively simplified or restored by applying a series of edge collapses or vertex splits. This representation is suitable for remote visualization. A base mesh can be sent and rendered first. Subsequently transmitted vertex splits will progressively refine the base mesh. However, the isosurfaces we extract contains hundreds of thousands of triangles, and the simplification process for meshes of that size takes minutes to complete and is impractical for interactive use.

There are also many algorithms that extract adaptive-resolution isosurfaces and offer level of detail control [34, 38]. More recently Westermann et al. [41] proposed a method to adaptively reconstruct isosurface without producing holes. The method uses a focus-point oracle and a curvature-dependent oracle to decide the level of detail at each point.

1.4 Dissertation Contribution

The previous section classifies and reviews related work on isosurface. Although there are many algorithms that aim to optimize various aspects of isosurface extraction, none of them addresses all three challenges---large datasets, scalable display, and distributed infrastructure---that we mention in Chapter One. The contributions of this dissertation include (1) a novel isosurface visualization algorithm that scales with dataset and display; (2) on the basis of the first algorithm, a remote visualization algorithm that minimizes data transmission between server and client. Below we discuss them in more details.

This dissertation proposes a progressive and view-dependent isosurface extraction algorithm that combines ray-casting and propagation. The algorithm's main loop casts one ray for each pixel on the screen. The intersection of the rays and the isosurface is calculated to find visible seeds. From each unvisited visible seed, limited propagation is used to generate a patch of isosurface. When all the rays have been cast, the correctness of the screen is guaranteed.

My experiments show that this approach extracts 99% of the visible isosurface much faster than the best previously-known approach. The algorithm is suitable for interactive visualization of large datasets. Using ray casting, traversal of the invisible sections of the dataset is largely avoided. The propagation enables us to only visit active cells and find the visible isosurface quickly. We also use adaptive-resolution, which means few primitives in the isosurface render to the same pixel and the number of extracted primitives is bounded by the display resolution.

The algorithm is also resolution-insensitive. In my experiment, only less than 1% of the rays (one per pixel) need to be cast for the screen image to reach 99% correctness. This is because propagation is much cheaper than ray-casting at finding visible active cells.

This dissertation further explores design options in the base algorithm. It evaluates the impact of intersection calculation methods---approximate versus exact, three ray-casting ordering choices, propagation termination criteria---, cut-off angle versus maximum count, and various other optimizations. The evaluation identifies the optimal point in the design space for the experiment dataset.

Because the algorithm's main loop is ray casting, it is easily parallelizable by breaking the screen into smaller rectangular regions. For the tiled display system of Display Wall, where each tile is driven by a computer, I implement a parallel isosurface visualization system where each computer extracts and renders for its display surface, a rectangle that only overlaps slightly with its neighboring tiles. The resolution of this rectangle is that of a generic projector, in our case 1024x768. In this simple parallel implementation, each computer works independently, and the overall performance is no worse than the slowest tile.

To visualize isosurface remotely, I design a remote visualization protocol that uses primitive compression, progressive level-of-detail (LOD), and primitive caching. My experiments show that the amount of data transferred can be reduced by two orders of magnitude.

This dissertation shows that my algorithm addresses all three challenges and is well suited for today's isosurface visualization needs.

The following chapters describe my algorithms in detail. Chapter 2 introduces the base algorithm, a scalable progressive view-dependent algorithm that combines ray-casting and propagation. Chapter 3 studies variants of the base algorithm and evaluates different design options to improve the performance. Chapter 4 addresses the problem of remote visualization; it explores division of the visualization pipeline between client and server and presents a protocol that minimizes the data transmission between client and server.

Chapter 2

Progressive View-Dependent Isosurface Propagation

This chapter proposes a new hybrid isosurface extraction algorithm that shares several of the features of existing acceleration methods. The main idea is to use ray casting into an octree as a way to identify visible seed cells (rather than directly computing the isosurface as in the method of Parker *et al*.[35]) and then use propagation (as in Itoh and Koyamada [18]) to extend the isosurface from the seed cells. Unlike previous propagation methods that propagate to the whole isosurface, our method uses distance and viewing criteria to decide where to stop propagation, and thus generates only a small piece of the isosurface connected to each seed cell. These pieces are patched together to form a view-dependent region of the isosurface, which includes all the triangles that are visible as well as a small number of occluded triangles that are near the visible surface.

In addition to largely avoiding the rendering of occluded portions of the isosurface, our method supports two acceleration schemes suitable for interactive visualization. First, we show that we can quickly acquire a very good approximation of the visible isosurface from just a few initial seeds, and then progressively refine the surface as subsequent rays discover the remaining visible active cells. Second, for very high-resolution datasets, we describe a form of adaptive-resolution rendering that relies on the octree organization of the data in order to extract and render triangles at a resolution chosen based on the scale of screen pixels relative to the cell size.

2.1 The Algorithm

The proposed algorithm may be viewed as an extension to the propagation method. Currently it works with structured datasets. The main contribution is to make the propagation algorithm view dependent in a manner that is efficient and incremental while supporting adaptive-resolution visualization.

First let us revisit the definition of the *active cell*. Given an isovalue *v*, we mark all the data points in the dataset with one of the two signs: "+" indicates that the scalar value at that point is above the isovalue, while "-" indicates that the scalar value is below the isovalue *v*. We only consider the non-degenerate case where no one voxel has exactly the value *v*. We can perturb a voxel's value by *epsilon* if it is exactly *v*. If a cell has vertices of different signs, then it is called an *active cell*, and the isosurface of threshold *v* will intersect this cell.

A key observation on which the propagation approach relies was that if the vertices on a face of an active cell do not have the same sign, then the neighboring cell that shares the same face is also active. Therefore, the isosurface can be extended into the neighboring cell. This means that once an active cell is found as the seed, propagating the isosurface from that cell is efficient because one can avoid touching and examining inactive cells. However, neither the Extrema Graph [18] nor the Contour Trees [21] algorithm generates seeds that are guaranteed to be visible. An efficient propagation algorithm should ideally traverse only the active cells that are visible.

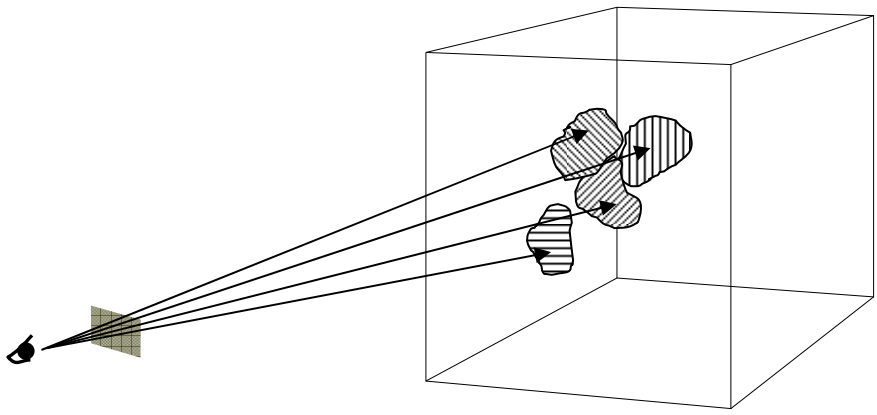


Figure 2.1: Illustration of the algorithm.

Our algorithm executes in three stages, shown in Figure 2.1. For each pixel in the screen space, first a ray is cast from the eye through the pixel into the dataset and the intersection is calculated. Next, the first active cell that contains a portion of the isosurface that intersects with the ray (if it exists) is used as the seed and propagated for a certain distance. Third, all the active cells that have been visited in this pass are examined,

case numbers are generated and the parts of the isosurface in these cells are triangulated using Marching Cubes method.

2.1.1 Ray casting

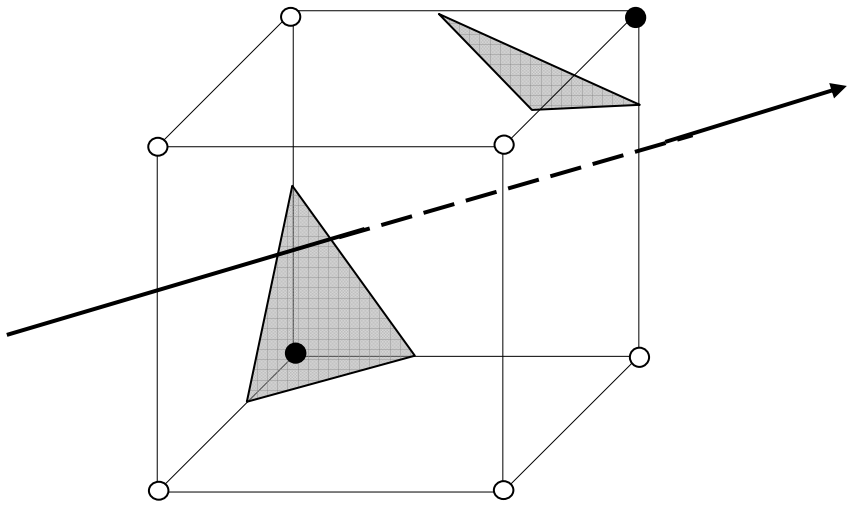


Figure 2.2: An example where the first active cell a ray intersects does not have an isosurface triangulation that intersects with the ray.

Our method uses ray casting to identify active cells as seeds for propagation. The ray-casting step finds the first active cell in which the triangulated isosurface intersects with the ray. This cell is guaranteed to be visible. If it has not been visited, this active cell will be used as the seed for the propagation step. Note that this cell is not necessarily the first active cell a ray intersects, as Figure 2.2 shows. The first active cell a ray intersects may have an isosurface triangulation that does not intersect with the ray, which means the triangulation will not render to the corresponding pixel. By finding the first active cell that actually renders to the corresponding pixel, we guarantee that for each ray cast, the

corresponding pixel has the correct shading. After a ray has been cast for each pixel, the final image will be correct. That proves our algorithm is conservative.

In order to make the ray-casting step efficient, we preprocess the dataset to build a branch-on-need octree (BONO) proposed by Wilhelms and Van Gelder [42] when it is first read into the memory. The octree has a depth of ceiling (*logD)+1*, where *D* is the size of the longest side of the dataset. Comparing with the three level hierarchy used in the ray-tracing method [35], the octree uses more space, but the intersection computation is faster. This is a trade-off between time and space requirements.

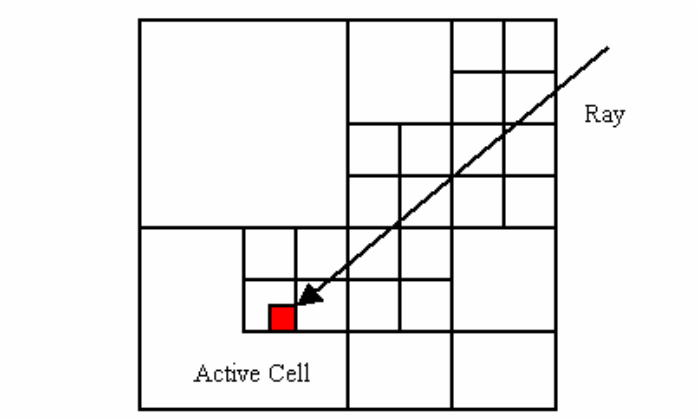


Figure 2.3: Octree optimization of ray-dataset intersection.

For each ray, our algorithm runs recursively to find the first active cell that intersects it. The intersection test can be seen as a depth first search into the octree. The ray is first tested against the whole dataset. If it intersects the dataset and the isovalue is between the overall minimum and maximum of the dataset, the sub-volumes in the dataset that intersect the ray are examined in a front-to-back order. The algorithm performs intersection tests and value comparisons recursively till it finds an active cell that the ray intersects, as shown in Figure 2.3. If the algorithm exits without finding an active cell,

then the ray does not intersect with the isosurface in the dataset. The corresponding pixel will not be shaded.

Once the active cell is found, we proceed to test whether the isosurface triangulation inside it actually intersects with the ray. We use the algorithm proposed by Möller and Trumbore [28] for this calculation. If the ray does not intersect the triangle(s), the depth first search resumes and the next active cell the ray intersects is found and tested. This is done till an active cell whose isosurface triangulation intersects with the ray is found or the ray exits the dataset, which indicates that the isosurface does not cover the corresponding pixel.

We use a hash table to record which cells have already been visited. If the active cell found by the recursive algorithm has not been visited, we will mark it and use this cell as the seed for the propagation step. If the active cell has been visited, then no further action will be taken.

To accelerate the ray-casting step, we perform the intersection calculation in the object coordinate system. The viewing volume is calculated from the projection matrix and mapped back to the object coordinate system using the current model view matrix. The coordinates of the voxels are fixed and implicit: each voxel is on a grid and has integer coordinates. The intersection is significantly faster than using the world coordinate system because every intersection test is with a cube with edges parallel to coordinate axes. This approach works for both perspective and orthographic projections.

A typical way to visualize the data is to begin by casting sparse and evenly distributed rays in the screen space, then add more rays to increase the ray density gradually till a ray has been cast for each pixel or user interrupts the extraction.

For a given screen resolution, the order of rays to be cast can be predetermined: the first ray goes from the center of the screen, the next 4 rays are each from the center of one of the 4 quadrants of the screen, and so on. The granularity of ray casting affects both the extraction speed and the isosurface precision. Fine-grained ray casting takes longer time, but it yields precise isosurface representation. Our design is to let user control the density of ray. When the user changes the isovalue or the viewport, all the calculations for the previous frame are immediately stopped and new ones begin. If the user does not interrupt, a ray will be cast for each pixel and the correct isosurface will be generated.

2.1.2 Propagation

Our algorithm uses a queue for propagation. Initially, the active cell found in the ray-casting step is the only one in the queue. For each cell in the queue, the algorithm dequeues it, sends it to the triangulation step, and checks all its active neighbors. If the active neighbor cells have not been visited and satisfy certain propagation criteria, they will be added to the queue. The propagation is in essence a breadth first search that is analogous to flood fill in the two dimensional case.

The propagation criteria need to be chosen carefully. The further the propagation proceeds, the more triangles are produces without incurring any overhead. On the other hand, even though the seed is visible, the cells that it propagates to are not necessarily visible. More propagation may increase the chance of traversing occluded cells. Also, expanding out of the screen space or to the back-faced side of isosurface is not desirable.

Our algorithm sets a cut-off angle for a ray and names it the propagation distance. At each propagation step, the algorithm calculates the angle between the ray we cast and the

vector from the eye to the current cell and stops adding it to the queue when the cut-off angle is reached.

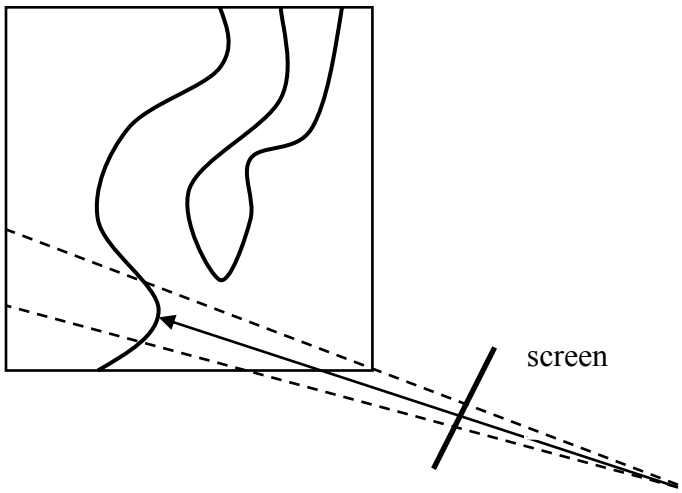


Figure 2.4: Propagation distance controls the projected area of each surface patch.

Figure 2.4 shows using a cut-off angle as the propagation distance. The propagation will not expand out of the region defined by the dashed lines. Suppose the first *m* rays have a combined propagation area that covers the whole screen, then after all the *m* rays are cast, the only possible reason for inaccuracy in the isosurface we computed is that there is an isolated part of the isosurface in front of the isosurface we expanded and it did not intersect with any ray we have cast. To treat this situation we just cast more rays at different locations. If the user does not interrupt, at last one ray will be cast for each pixel and result is guaranteed to be correct. We may generate more triangles than that are actually visible, but it is interactive because the user gets approximate results that refine

with time. Our results show that for several datasets, only a relatively small number of rays are needed.

In order to avoid generating occluded triangles, we calculate the angle between the current ray and the vector from the eye to the cell being propagated to detect whether the isosurface folds back. We name this angle the distance angle. If the isosurface folds back in a cell, that cell will not be added to the queue and the propagation from the cell will stop. The detection works as follows. If a cell C_1 is examined and its neighbor C_2 is added to the queue, then we say C_1 is C_2's predecessor in propagation, and C_2 is C_1's successor. Normally, the distance angle between the ray and the vector increases as the propagation proceeds. When the distance angle for a cell decreases comparing with its predecessor cell, then the isosurface is curving back and the cell should be discarded. However, to deal with bumpy surfaces, our algorithm has a small tolerance value. Only when the decrement exceeds the tolerance value do we stop.

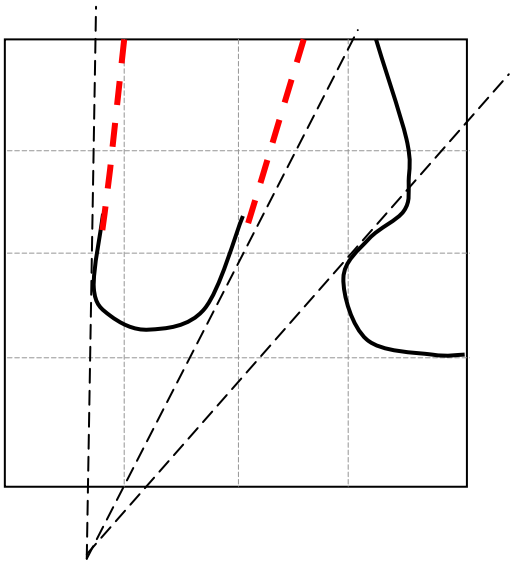


Figure 2.5: Surface shown in red dashed lines are not extracted.

Propagation is an efficient way of extracting triangles; we are willing to pay the small price of rendering a few more triangles to keep the propagation going. The decrement is calculated from the largest distance angle among all the (indirect) predecessors of a cell, so even if in each propagation step the decrement is very small, they can accumulate and stop the propagation. This solves the problem where the isosurface curves back and propagates in a direction that is almost parallel to the ray, as shown on the left side in Figure 2.5. The dashed part of isosurface will not be propagated to, whereas a slight depression in the surface is tolerated and propagation continues on the right side.

2.1.3 Adaptive-resolution isosurface

For a large dataset or a distant viewpoint, it is possible that a cell is of sub-pixel size when projected to the screen. If it is an active cell, then more than one triangle in the isosurface will render onto the same pixel. That is a waste of computing resources and does not increase the image quality. Our algorithm detects such cases and reduces the data resolution to 2×2×2 (treating a meta-cell that consists of 8 cells as a single cell and ignoring all the inside values) or even lower. This is feasible because our ray-dataset intersection traverses down the octree hierarchy. We can stop at any resolution if the (meta-) cell projects to less that one pixel on the screen. Given the eye position, screen position, and screen resolution, we can compute an array D, such that for a meta-cell of size $2^i \times 2^i \times 2^i$, if its distance from the eye is larger than $D[i]$, then it should be treat as one single cell. When the propagation crosses the resolution boundary defined by D, our algorithm stops at the boundary. At the resolution boundary, there will be cracks in the

actual representation of the isosurface, i.e. the triangles from different resolutions may not connect to each other, but the cracks will not be visible because they are of sub-pixel size. For every shaded pixel, the color is correctly rendered by a triangle that intersects with the corresponding ray. This is similar to [24], where the set of triangles that are rendered is only a subset of all the visible triangles, and where a single point is used to represent a faraway meta-cell. The view-dependent methods generate results that user perceives as identical with the complete representation from his current viewpoint.

2.2 Results

We have implemented the algorithm above on a PC that runs Windows 2000, and conducted experiments with the implementation. The PC hardware includes a 933Mhz Pentium III CPU, an NVIDIA GeForce 256 graphics card, and 512 MB of main memory. We applied our algorithm to the head section of the Visible Woman CT data, using a 512×512×209 dataset, which is at its original data resolution. A cut-off angle of 1.81 degrees was used for all the experiments. The visualization is done in full screen mode with a screen resolution of 1600x1200. In the absence of user interrupt, 1,920,000 rays will be cast, one from each pixel.

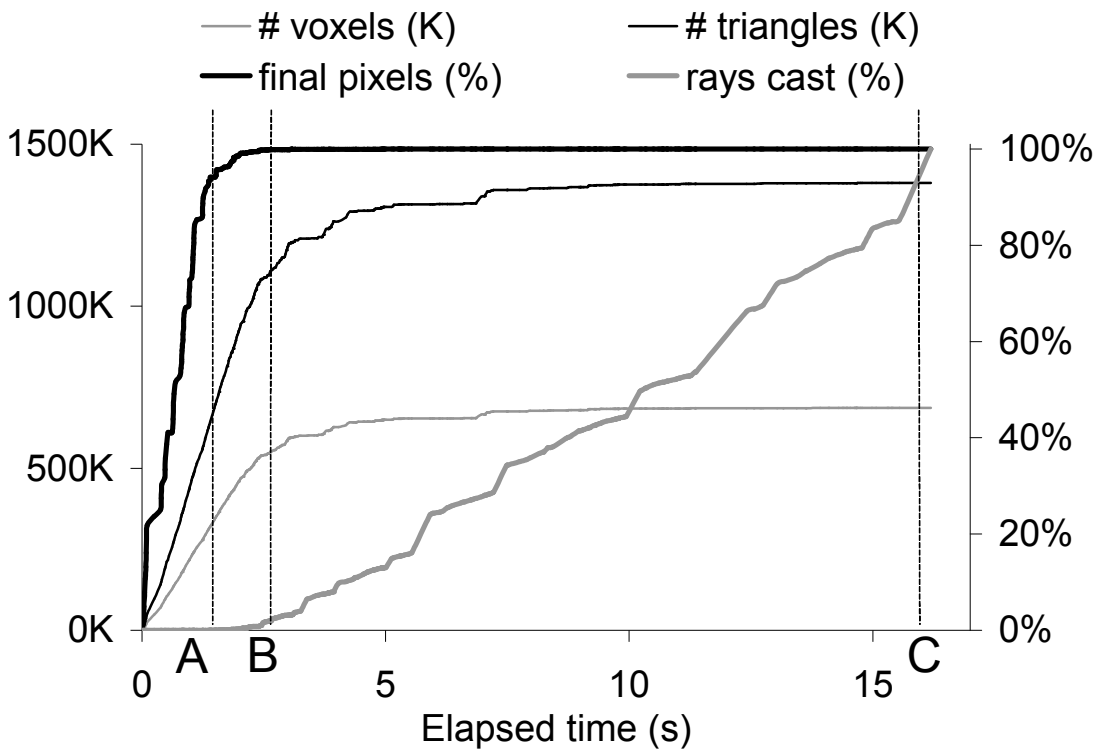


Figure 2.6: The front full view of the skin. V=600.5, left column in color plate.

Point	Time (s)	Triangles (K)	Final Pixels (%)	Rays (K)
A	1.1	514	85.3	0.2
B	2.4	1,068	99.5	15.6
C	16.2	1,380	100.0	1,920

Table 2.1: The three points in Figure 2.6.

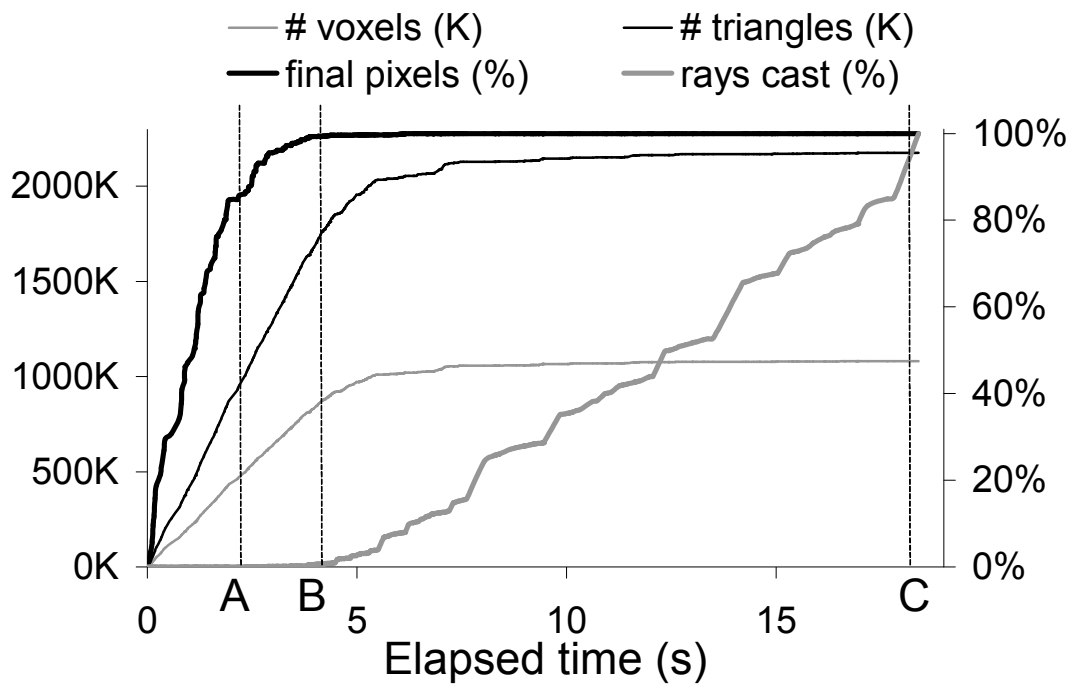


Figure 2.7: The side full view of the skeleton. V=1220.5, right column in color plate.

Point	Time (s)	Triangles (K)	Final Pixels (%)	Rays (K)
A	2.2	953	85.7	0.6
B	4.3	1,796	99.5	15.1
C	18.4	2,175	100	1,920

Table 2.2: The three points in Figure 2.7

To quantitatively measure how close the an intermediate representation of the isosurface is to the correct and final representation, for each pixel that has been rendered to in the final representation, we check whether it has the same color in the intermediate image, and if so name it a final pixel. The percentage of the final pixels among all the rendered pixels indicates the correctness of the intermediate image. Figure 2.6 shows in an experiment of extracting the Visible Woman's skin, how the percentage of final pixels,

the number of active cells visited, the number of triangles generated, and the percentage of rays cast change as the computation proceeds. The statistics for points A, B, and C are shown in Table 2.1. The corresponding screen images are shown in the color plates.

Figure 2.7 and Table 2.2 show the result from another experiment that extracts the bone structure from Visible Woman's head. Despite the different characteristics between the skin and skeleton surface, the similarity between the graphs shows that our algorithm behaves consistently. Both cases show that the proposed algorithm works progressively and efficiently. After casting a few rays, our algorithm generates most of the isosurface. In the skin extraction case, over 85% of the isosurface is extracted in 1.1 seconds with only 240 rays cast (point *A*), whereas 99.5% of the isosurface is extracted in about 2.4 seconds with about 0.8% rays cast (point *B*). To obtain 100% pixels, it took 16.2 seconds. The bone extraction case has similar curves but it took 2.2 seconds to obtain 85% of the isosurface and 4.3 seconds to extract 99.5% of the isosurface. The total extraction took 18.4 seconds. This is because the isosurface of the bone has about 60% more triangles to render than the skin.

The last 0.5% of pixels took much longer time to extract in both cases, but they make very little difference on the screen.

Figure 2.8 shows the breakdown of the computation time. The definitions of the stages are as follows:

Intersection time is the time spent on ray-dataset intersection and finding visible seeds.

Extraction time is the time spent on using marching cubes algorithm to compute the isosurface triangulation inside active cells.

Propagation time is the time spent on finding the active neighbors of a cell in the propagation queue and testing whether they satisfy the propagation criteria.

Rendering time is the time spent on rendering all the triangles generated.

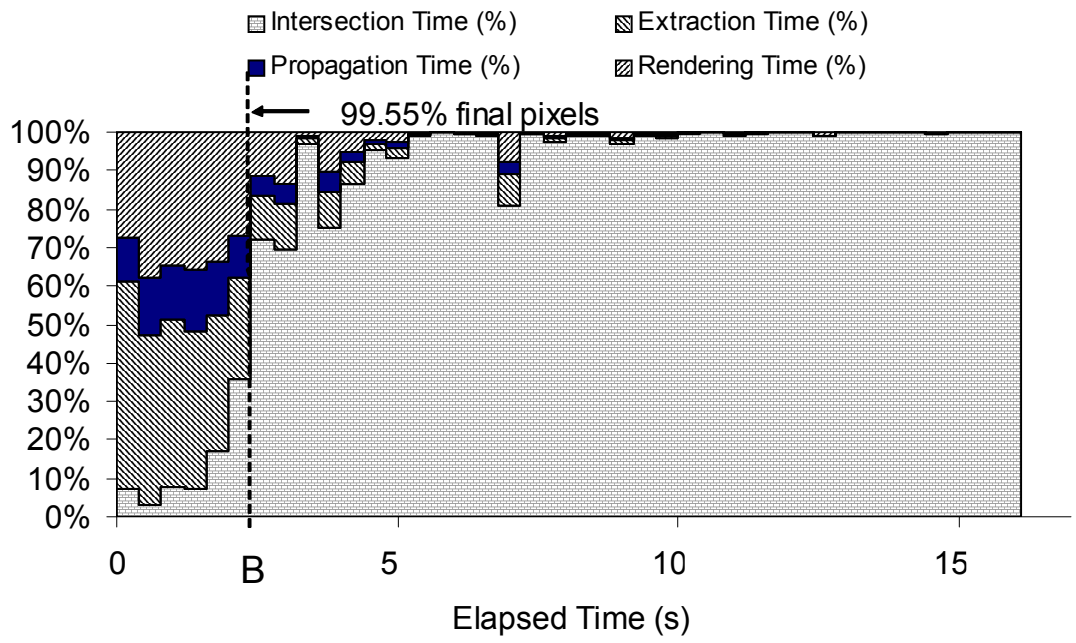


Figure 2.8: Running time decomposition among stages.

Note that among these four time components, only intersection time is proportional to the number of rays cast. The other three are proportional to the number of triangles generated. When the full computation ends, the intersection calculation is the most time-consuming stage. However, most of the isosurface has been generated by the time the intersection stage becomes significant. The dashed line shows point *B* in Figure 2.6. Before point *B*, the time spent in intersection is negligible. At point *B*, the representation is 99.5% correct. After point *B*, the intersection calculation becomes dominant. In this experiment, because of the high resolution of the display (1600×1200), all the triangles are generated at the finest data resolution. This implies that our algorithm is suitable for

large-scale displays. For a fixed dataset, when the resolution of the display increases, only the intersection time will increase, which has very little influence on the position of point *B* on the time axis.

2.3 Comparisons with Previous Algorithms

It is difficult to compare our approach quantitatively with other approaches, without implementing them all on the same hardware platform. However, we can make some qualitative comparisons. Here we focus on view-dependent work.

Our approach allows viewers to see the shapes of the isosurface after casting only a few rays, whereas in the naïve implementation of ray tracing the visual quality is linear in both the number of rays cast and the elapsed time. Also, our approach leverages the cost-effective rendering performance of PC graphics cards. Similar to the ray-tracing approach, our algorithm is image-space based and can be parallelized.

To perform a crude comparison with the WISE method of Livnat and Hansen [24], we ran our algorithm using the same size display (512×512 pixels) as they used in recent experiments [23] with the same Visible Woman dataset. The results of their experiments indicate that running on a SGI Onyx 2 they extract 344,628 triangles in 35.8 seconds, and then render this surface in 0.6 seconds. In contrast, our method, running on a Pentium III 933MHz PC with GeForce graphics card, extracts and renders 1,289,904 triangles in 4.3 seconds. Based on the relative clock rates on the platforms, we expect that our performance would be better on the SGI than that of the WISE algorithm. Very recent work by Livnat and Hansen introduces SAGE [23] a view dependent algorithm that

improves on the performance of WISE (and given the relative hardware difference probably exceeds our performance as well), with a reported extraction time of 4.4 seconds and rendering time of 0.3 seconds for the same dataset. Because our visibility test is more conservative, our method extracts and renders many more triangles than the WISE and SAGE algorithms. However, our experiments indicate that triangle rendering is not a bottleneck, and inexact visibility allows us to quickly find large portions of the surface. Our algorithm very quickly provides a good approximation of the surface: 85% and 99.5% correctness were achieved at 0.9s and 2.2s respectively on the 512×512 display. Finally, comparing these numbers with those in Table 2.1 (1.1s, 2.4s), we show that while the number of the pixels increases by more than a factor of 7 (262,144 to 1,920,000), the points *A* and *B* were only delayed 20% and 10% respectively. This suggests that the progressive aspect of our algorithm scales well for very high-resolution displays.

2.4 Summary

This chapter describes a new isosurface extraction algorithm based on ray casting and propagation. We have shown that the new approach is progressive and efficient.

Our algorithm is suitable for high-resolution displays. The results reported in this chapter were generated using a PC at full screen (1600×1200) resolution. We would like to adapt the algorithms presented here for use with tiled displays, as part of the Princeton Display Wall Project [22]. As an initial step, we ran the program on a large-scale (18-foot) display surface covered by 24 tiled projectors arranged on a 6×4 grid, yielding more than

20 million pixels. A server PC drives each projector, and each PC runs a copy of the isosurface extraction algorithm. When the isosurface is spread over several projectors, we find a corresponding performance improvement because each PC has a partial view of the surface and has fewer triangles to extract and render. However, when the isosurface falls entirely within one projector, the performance drops to that of a single PC. To address this problem, a load-balanced parallel version of the algorithm is needed.

Our algorithm is suitable for large datasets. Currently the entire dataset resides in memory, which limits the size of dataset we can visualize. Because surface propagation has strong data locality, we believe that it will be possible to adapt an out-of-core version of our algorithm.

Remote data visualization has become an important area of research because massive amounts of data are generated and distributed over the network. Since our algorithm aims to reduce the number of triangles generated as well as maintain a fast extraction speed, we believe it is suitable for remote data visualization. Moreover, surface propagation yields triangle patches that should perform well under geometry compression. Finally, we intend to exploit data-locality due to frame-to-frame coherence in interactive data exploration when adapting our algorithm for remote visualization.

Chapter 3

Algorithm Improvements

This chapter explores different design options in order to further improve the algorithm presented in the previous chapter.

3.1 Design Options

3.1.1 Intersection Calculation Options

Our method uses ray casting and intersection calculation to find seed cells. To guarantee the correctness of the final result, the intersection calculation should not stop when an active cell that intersects the ray is found. It is necessary to verify that the isosurface inside the cell intersects the ray as well. We shall call active cell intersection *approximate intersection* and isosurface intersection *exact intersection*. In the previous chapter, Figure 2.2 shows a case where using approximate intersection only causes incomplete isosurface extraction.

However, exact intersection entails substantial floating number calculation, making it an expensive operation. An alternative to this is to first use approximate intersection, where the first active cell intersecting the ray is accepted as the seed. Exact intersection is used in the second pass to ensure correctness.

Section 3.3.1 reports our experimental results to compare the two approaches. Based on the results, we decided to use exact intersection.

3.1.2 Ray-Casting Order Options

The granularity of ray casting determines the speed and the precision of isosurface extraction. Fine-grained ray casting takes time, but it yields precise isosurface representation. Our design lets user control the density of the rays. When the user changes the threshold or the viewpoint, all the calculations for the previous setup are immediately stopped and new ones begin. If the user does not interrupt, a ray will be cast for each pixel and the correct isosurface will be generated.

There are several ways to cast rays for the algorithm.

Fixed refining grids: A straightforward way is to cast sparse and evenly distributed rays in the screen space, then cast rays on progressively finer grids to increase the ray density gradually till a ray has been cast for each pixel or user interrupts the extraction. This method uses a predetermined hierarchy of grids to determine the ray casting order. The rays are organized in a quad-tree fashion. The first level consists of one ray that goes from the center of the screen. The second level has 4 rays that are each from the center of one of the 4 quadrants. In general each level of rays form a uniform grid that doubles the

resolution and contains 4 times as many rays as the last level. At the deepest level, rays simply fill in pixels that have not been accounted for.

Screen read-back: The second way to cast rays is to dynamically decide the ray casting order based on the pixels rendered so far. After casting all rays of a particular level, the algorithm will read back all pixels of the screen. This method first casts rays whose corresponding pixels have not been rendered and then cast rays whose corresponding pixels have been rendered.

Dynamic grid: Another method is to dynamically decide the grid resolution based on the first piece of isosurface generated. When the first piece of new isosurface is generated, the algorithm will test nearby rays to decide the rough size of the surface and dynamically generate a grid in the hope of covering the screen space quickly without leaving any isosurface gaps. The rest of the rays are then cast in the fixed refining grids order.

It is not obvious which of these methods will work well. To evaluate the ray casting order, we have implemented all three ray-casting schemes and compared them in Section 3.3.2. The dynamic grid performs the best in our experiments.

3.2 Propagation Options

Our algorithm uses a queue for propagation. Initially, the active cell found in the ray-casting step is the only one in the queue. For each cell in the queue, the algorithm dequeues it, sends it to the triangulation step, and checks all its active neighbors. If the

active neighbor cells have not been visited and satisfy certain propagation criteria, they will be added to the queue.

An important issue is propagation distance, which determines how far the propagation for each seed cell should go. The further the propagation proceeds, the fewer inactive cells the algorithm has to examine. On the other hand, although the seed is visible, the cells that it propagates to are not necessarily visible. More propagation may increase the chance of traversing occluded cells. Also, expanding out of the screen space is not desirable.

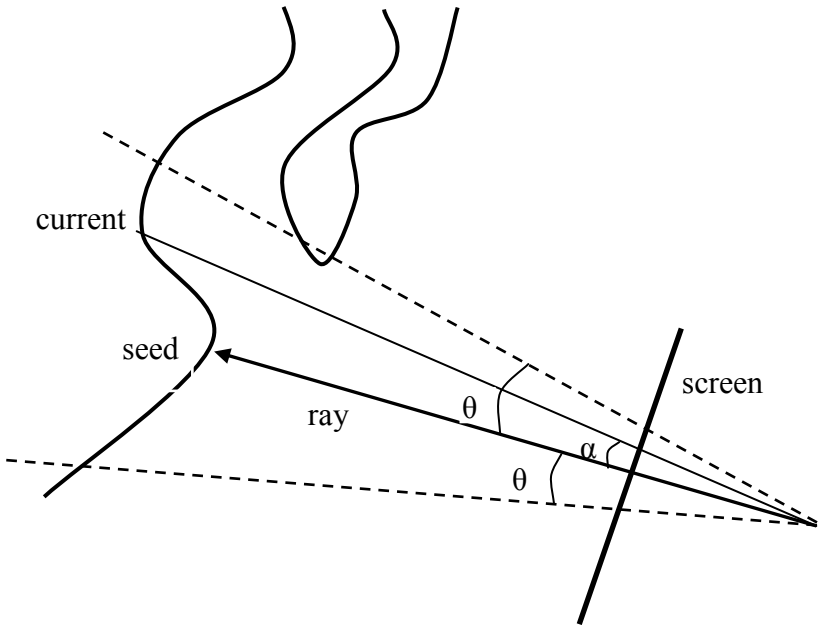


Figure 3.1: Limited propagation.

We consider two design choices. As shown by the figure on the left, the first ray is to calculate the angle α between the current ray and the vector from the eye to the cell being propagated. We set a cut-off angle θ and use it as the propagation distance. At each

propagation step, the algorithm calculates α for the current cell and stops adding it to the queue if $\alpha > \theta$. This method is fairly conservative in terms of generating visible triangles, but it requires vector calculations for each cell visited, which introduces overhead to each propagation step.

Another method is to simply use a maximum count *m* as the propagation distance. In this case, *m* is the maximum number of cells the propagation from a seed cell can visit. It is often possible that each pass of propagation will visit fewer than *m* cells. The ray can hit a gap in the current isosurface and the propagation will stop once the gap is filled and all the neighboring cells had been already visited. Or the ray can hit an isolated isosurface component that contains less than *m* cells. The advantage of this approach is its simplicity.

We have evaluated these two approaches in Section 3.3.3. We decide to use maximum count because of the significant improvement in performance. We also investigated how to choose the value of *m* in Section 3.3.6.

3.3 Performance Evaluation

We have implemented the algorithm on a PC running Windows 2000, and conducted experiments to evaluate several design choices described in the previous section. The PC hardware includes a 933Mhz Pentium III CPU, an NVIDIA GeForce 3 graphics card, and 512 MB of main memory.

We applied our algorithm to the head section of the Visible Woman CT data, using a 512×512×209 dataset, which is at its original data resolution. The visualization is done in

full screen mode with a screen resolution of 1600 × 1200. In the absence of user interrupt, 1,920,000 rays will be cast, one from each pixel.

To quantitatively measure how close an intermediate representation of the isosurface is to the correct and final representation, for each pixel that has been rendered to in the final representation, we check whether it has the same color in the intermediate image, and if so name it a final pixel. The percentage of the final pixels among all the rendered pixels C indicates the correctness of the intermediate image.

In the following comparisons, we records points where C = 90, 99, 99.9 and the end of calculation, when every ray has been cast. Note that 100% correctness can happen before the end of the calculation. For each point, the elapsed time, number of triangles generated, and number of rays cast are measured as a way to compare different design choices.

The basic implementation is the one reported in last chapter. Our original implementation performs exact ray-isosurface intersection for each ray, uses a cut-off angle to control the propagation, and casts rays in fixed refining grids.

3.3.1 Results for Intersection Calculation Options

Exact				Approximate			
C	Time (s)	Δs (K)	Rays	C	Time (s)	Δs (K)	Rays
90	1.37	654	833	90	1.44	683	833
99	1.72	784	4,456	99	1.73	774	4,508
99.9	1.78	786	8,711	99.9	1.87	776	4,855
100	13.8	809	1.92M	100	18.7	798	1.92M

Table 3.1: Exact intersection vs. approximate intersection.

In Section 3.1.1, we discussed two approaches to intersection calculation for ray casting: exact intersection and approximate intersection. Table 3.1 shows the performance of both approaches. Both tests use a cut-off angle θ of 2 degrees.

Although approximate intersection appears to require less computation, some rays have to be checked twice. As a result, the total running time is actually longer than the exact intersection approach. The difference in performance is fairly small with the exception of total running time.

3.3.2 Results for Ray-Casting Order Options

Three ray casting schemes are considered: fixed refining grids, dynamic approach based screen read-back, and dynamic grid based on the first piece of isosurface. Table 3.2 shows the results. In all three tests a maximum count of 8000 are used.

Fixed Refining Grids				Screen Read-back				Dynamic Grid			
C	Time (s)	Δs (K)	Rays	C	Time (s)	Δs K)	Rays	C	Time (s)	Δs (K)	Rays
90	0.80	380	209	90	0.81	380	209	90	0.73	340	118
99	0.98	463	879	99	0.98	463	879	99	1.03	486	1,016
99.9	1.38	591	18,013	99.9	1.34	575	18,013	99.9	1.23	552	8,636
100	13.2	641	1.92M	100	13.5	641	1.92M	100	13.3	650	1.92M

Table 3.2: Experiment results of three ray-casting orders.

The results suggest that the three ray-casting schemes give similar performance. The dynamic grid scheme has the best running time to reach $C = 99.9\%$. Because the improvement is not significant, we still use the Fixed Refining Grids for its simplicity. We intend to study performance improvements for the Dynamic Grid scheme in the future.

3.3.3 Results for Propagation Options

We have considered two design choices of propagation distance to control how far the propagation for each seed cell should go: cut-off angle and maximum count. Table 3.3 shows the results of the two approaches with best parameters of m and θ. The left column of Table 3.3 gives the performance of using a maximum count of 8000. The right column is the performance of the cut-off angle approach with $\theta = 2$.

Maximum Count

C	Time (s)	Δs (K)	Rays
90	0.80	380	209
99	0.98	463	879
99.9	1.38	591	18,013
100	13.2	640	1.92M

Cut-off Angle

C	Time (s)	Δs (K)	Rays
90	1.37	654	833
99	1.72	784	4,456
99.9	1.78	786	8,711
100	13.8	809	1.92M

Table 3.3: Comparing two propagation options: maximum count vs. cut-off angle.

It is clear that the maximum count approach is superior to the cut-off angle approach. The running time is substantially less than the cut-off angle approach. This is because the cut-off angle approach requires more complex calculations than the simple maximum count and may generate large numbers of invisible triangles when propagating to a piece of occluded isosurface that runs almost parallel to the ray.

3.3.4 Caching Interpolants and Triangles

Isosurface triangulation is expensive because interpolation involves floating point divisions. An internal edge is shared by 4 cells so it will be visited 4 times. We use a hash table to record the interpolants.

Caching On				Caching Off			
C	Time (s)	Δs (K)	Rays	C	Time (s)	Δs (K)	Rays
90	0.64	380	209	90	0.80	380	209
99	0.78	463	879	99	0.98	463	879
99.9	1.13	591	18,013	99.9	1.38	591	18,013
100	11.7	641	1.92M	100	12.9	641	1.92M

Table 3.4: Caching on vs. caching off.

Each vertex in the isosurface is shared by 6 triangles on average. Instead of copying the coordinates and normal repeatedly we use indexed face sets as the representation for a patch of isosurface generated from one pass of propagation. The interpolant cache is reset at the beginning of each pass of propagation. When an item in the hash table is hit only its index in the vertex array is returned. In our experiments indexed face sets save 60-70% storage space and render more quickly.

All the triangles generated are cached to accelerate ray-triangle intersections. All the face sets are stored in a scratch space so that is no overhead for triangle caching.

The performance improvements by using interpolant and triangle caching are shown in Table 3.4.

3.3.5 Screen Bounding Box

When the isosurface covers only a small part of the screen, many ray-isosurface intersection tests in the empty region of the display will return nothing. In some of the tests, the ray does not intersect the dataset at all. CPU cycles are wasted without

contribution to the results. We partially address this problem by projecting the dataset onto the screen and finding the screen bounding box for the dataset. Rays outside the bounding box will not be cast. In our experiments, this optimization shortens the running time for remote viewpoints. When the bounding box covers the full screen, this optimization introduces negligible overhead by doing one more comparison per ray. We scale down the size of dataset on the screen by different percentages and show the respective reductions in running time by using screen bounding box in Table 3.5. Since in the normal viewpoint we use the dataset covers almost the whole screen, there is only minimal improvement.

Size (% of normal)	100	50	25	10
Reduction in running time (%)	3	24	39	49

Table 3.5: The effect of using screen bounding box.

3.3.6 Impact of m

The choice of *m* (the maximum number of cells visited per propagation) can affect the performance of the algorithm. Table 3.6 below shows how the position of point *C* = 99.9% and total number of triangles rendered change with *m*.

m (K)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
C=99.9 (s)	3.73	1.52	1.31	1.33	1.20	1.07	1.10	1.13	1.07	1.17	1.18	1.15	1.18	1.12	1.26
#T (K)	506	529	558	562	609	620	639	641	661	669	636	659	685	738	768

Table 3.6: The effect of various maximum count values.

Generally speaking, too small an *m* causes more rays to be cast to generate the approximate result, while too big an *m* propagates to more invisible triangles. These both

affect the performance of the program. The best choice of *m* also varies with the threshold. Certain profiling of dataset should be done to decide *m* offline.

3.4 Results

In this section we present test results from our optimized implementation. Figure 3.2 shows in an experiment of extracting the Visible Woman's skin, how the correctness, the number of active cells visited, the number of triangles generated, and the percentage of rays cast change as the computation proceeds. The three vertical lines show the positions of points where $C = 90, 99, 99.9$. The corresponding statistics are shown in Table 3.7. The corresponding screen images are shown in the color plates.

Figure 3.3 and Table 3.8 show the result from another experiment that extracts the bone structure from Visible Woman's head. The similarity between the graphs shows that our algorithm behaves consistently. Both cases show that the proposed algorithm works progressively and efficiently. After casting a few rays, our algorithm generates most of the isosurface.

In the skin extraction case, over 90% of the isosurface is extracted in 0.52 seconds with only 79 rays cast, whereas 99.9% of the isosurface is extracted in about 1.07 seconds with about 0.7% rays cast. To obtain 100% pixels, it took 11.7 seconds. The bone extraction case has similar curves but it took 0.99 seconds to obtain 90% of the isosurface and 2.65 seconds to extract 99.9% of the isosurface. The total extraction took 12.7 seconds. This is because the isosurface of the bone has about 70% more triangles to render than the skin.

The last 0.1% of pixels took much longer time to extract in both cases, but they make very little difference on the screen.

SKIN (v = 600.5)

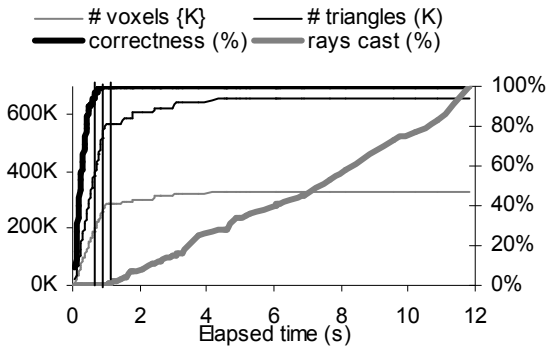


Figure 3.2: Front full view of skin.

C	Time (s)	Δs (K)	Rays
90	0.52	306	73
99	0.79	461	947
99.9	1.07	569	13,361
100	11.7	661	1.92M

Table 3.7: Points in Figure 3.2, *m*=9000.

Bone (v=1224.5)

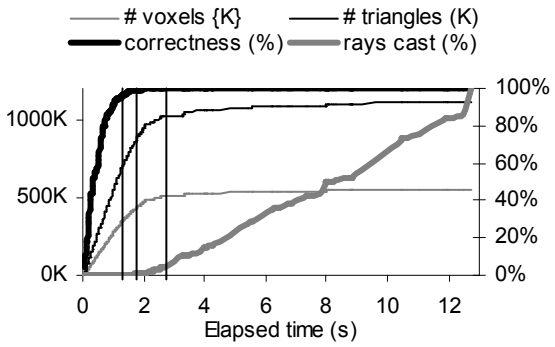


Figure 3.3: front full view of bone.

C	Time (s)	Δs (K)	Rays
90	0.99	523	801
99	1.65	830	4,738
99.9	2.65	1,070	73,099
100	12.7	1,112	1.92M

Table 3.8: Points in Figure 3.3, *m*=5000.

3.5 Summary

This chapter studies how to improve our base algorithm described in the last chapter. We have learned the following from our evaluation of the design choices of the algorithm:

1. Exact intersection calculation is slightly better than approximate intersection calculation.

2. Ray casting order does matter slightly. The dynamic grid performs the best. There might be other improvements to this method.
3. Using a maximum count instead of a cut-off angle to control propagation can substantially reduce the running time to extract most of the isosurface. The maximum count approach produces a smaller number of triangles than the cut-off angle approach.
4. Caching interpolants and triangles can improve the performance.
5. Using a bounding box for the dataset can effectively reduce the number of rays required.
6. The maximum count value can impact the performance slightly.

By carefully making design decisions, we can improve the algorithm significantly. The resulting algorithm improves the progressive aspect substantially. In extracting the same skin isosurface, our old implementation (exact intersection, fixed refining grids, using cut-off angle, no other optimizations) reaches C = 90% and 99.9% at 1.26 and 3.03 seconds respectively. To extract 90% of the isosurface, the new approach that combines the best options improves over the original one by a factor of 2.4. To extract 99.9%, it improves by a factor of 2.8 and requires casting only a small number of rays, less than 0.7% of the pixels of the display screen.

Chapter 4

Remote Visualization

An important issue in remote data visualization is to decide what work in the visualization pipeline should be done on the server and what should be done on the client. In one extreme case, the server sends the whole dataset over the network to the client and the client performs local geometry extraction and visualization. This approach requires too much communication bandwidth. A naïve way to reduce the communication bandwidth requirement is to let the server send all the active cells to the client for processing. This approach still requires quite a bit of communication bandwidth. Furthermore, both methods demand too much computation power on the client machine.

In another extreme case, the server performs both geometry extraction and rendering [14]. The resulting frames can be compressed and sent to the client. This approach frees the client from doing processing, but it requires the server to compress the resulting images aggressively. Otherwise, the method may also require high communication bandwidth. This approach is not scalable with screen resolution.

We are in favor of using 3D primitives as the intermediate representation of the isosurface to transfer from the server to the client. This approach requires the server to have enough computing power to perform feature extraction and the client to have enough rendering power to render the primitives for its display. Today's PC with an inexpensive commodity graphics card is adequate for a single display. A cluster of PCs is adequate for a display wall. This approach has the potential to minimize the communication requirement for remote data visualization.

This chapter describes three techniques applied incrementally to our isosurface extraction algorithm, which is a hybrid algorithm of ray-casting and isosurface propagation. The first technique is to conveniently organize 3D primitives into groups and to efficiently compress the groups into indexed face sets. The second technique is to extract high-level isosurface primitives first so that the client can see the isosurface shape quickly with minimal communication requirement. As the time goes, the server will extract isosurface primitives at progressively finer level-of-detail so that the client can see the details of the isosurface. The third technique is to cache the primitives on the client side to exploit the coherence between frames so that no primitives need to be resent when the client changes the viewpoint with the same isovalue. After a user rotates an isosurface around, subsequent frames could be rendered with minimal communication. This is a major advantage over sending images over the network.

We have implemented our proposed algorithm in a simple remote data visualization environment. Our evaluation shows that these techniques are quite effective. Applied to the base algorithm incrementally, each technique more than doubles the user-perceived

speed of visualization. Combined together, the three techniques reduce the time needed to obtain a 90% or 99% correct result by 98% and 92% respectively.

4.1 The Algorithm

We have described an isosurface extraction algorithm that extracts portions of the isosurface in a view-dependent manner based on the combination of ray casting and propagation. The extraction engine organizes the resulting partial isosurface into groups of primitives and sends them in a compressed form over the network to the client. To enhance the interactivity and quality perceived by the user, our algorithm extracts the isosurface at different levels of detail and displays a coarse representation of the isosurface first. The result refines with time if the user does not interrupt. When the isovalue is coherent across frames, only new primitives are extracted and sent. Each primitive in the isosurface is processed at most once.

4.1.1 Base algorithm

Our algorithm described in the previous chapters suits a remote visualization system well as the base algorithm for several reasons. First, it generates view dependent isosurfaces and reduces both computational cost and bandwidth requirement. Second, the isosurface pieces can be used as the units for primitive compression because the triangles in each piece are connected together. Third, these pieces can be used as the units to exploit frame coherence of the isovalue and avoid extensive communication between the server and the client. Finally, the dataset is organized into an octree to speed up ray-isosurface

intersection; this hierarchical representation of data can be used to generate isosurface from a coarse level to a detailed level progressively.

4.1.2 Primitive Compression

Our extraction algorithm casts rays into the data volume and computes the intersection points with the isosurface and then performs bounded propagation from those points. The output of each pass of propagation is a number of triangles that form a continuous piece of surface. We call this triangle set a *group*. A group represents a part of isosurface extended from a seed cell. The isosurface can be viewed as made up of these groups. Since an isosurface easily contains hundreds of thousands of triangles, it is hard to manipulate them individually. Grouping is a natural way to organize the 3D primitives into manageable sets. We use the group as the indivisible unit for transmission, rendering, and later manipulations.

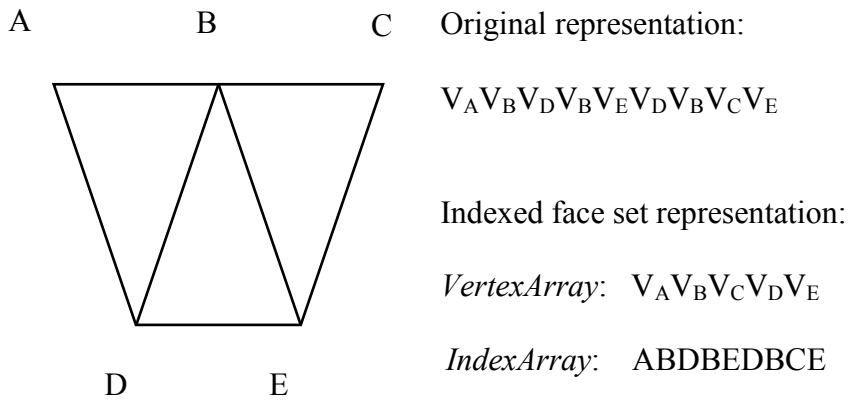


Figure 4.1: An example of indexed face set representation.

Whenever the extraction engine generates a new group, it is sent to the client for rendering and caching. The groups (data) account for 99% of the total network traffic in

our system. Since a group represents a piece of surface with a lot of spatial coherence, compressing it before transmission is both natural and desirable. There exist many generic data compression schemes as well as those specifically designed for 3D primitives. However, most of these compression schemes will produce results that cannot be directly rendered by OpenGL. The client machine will be forced to decompress the data before rendering it. This poses additional demands on the computational power of the client machine, which is contradictory to our design philosophy. We choose to compress a group of triangles in the form of an indexed face set, as shown in Figure 4.1. The compression is free of cost in our implementation. In the representation of isosurface, each vertex is shared by 6 triangles on average. The coordinates and normal calculation of each vertex involves interpolation between two voxels and their gradients. To avoid performing the same costly calculations multiple times, we use a hash table to record vertices that have been calculated in the current pass of propagation. Instead of storing its coordinates and normal in the hash table, we simply store the index of the vertex in the *VertexArray*. The *VertexArray* only contains distinct vertices. When a vertex is used again in a newly extracted triangle, we look it up in the hash table and copy its index into the *IndexArray*. After the propagation terminates, both arrays are sent to the client.

In our experiments this compression results in 60%-70% reduction of data size. Moreover, this compact representation can be directly and efficiently rendered by an OpenGL routine at the display client side. Indexed face sets render faster than separate triangles because there is less data to send to the graphics card and each distinct vertex only needs to be projected and rasterized once. Our approach lowers the network bandwidth requirement as well as speeds up both the extraction engine and the display

client. In our experiments, the total running time is reduced by 47% using primitive compression.

4.1.3 Progressive LOD

LOD, short for *level of detail*, describes how coarse the representation of the isosurface is. In algorithms that organize the dataset into a hierarchical representation such as an octree, coarser LODs are easily achieved by applying the marching cubes algorithm on $2^k \times 2^k \times 2^k$ meta-cells. When the isosurface is far away or display resolution is low, triangles in the isosurface could be smaller than the area of a pixel on the screen. Extracting the isosurface at a coarser LOD saves computation time without compromising the image quality. Coarser LODs are also frequently used for parts of the isosurface that are not the focus of visualization.

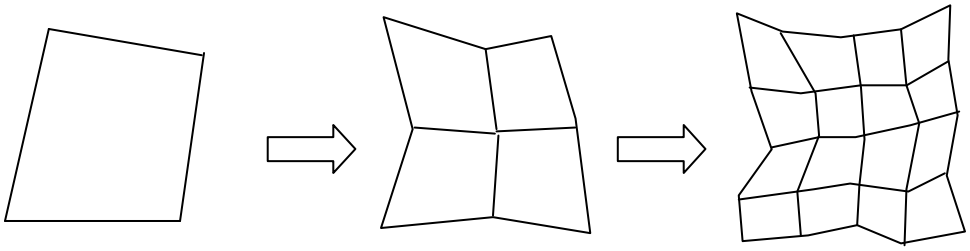


Figure 4.2: A coarse representation of surface refines into a detailed representation.

In our original implementation of the extraction algorithm (described in Section 2.1), coarser LODs are only used for the groups in which the triangles project to smaller than a pixel. However, when adapting the extraction algorithm for remote visualization, we realize that even only the view-dependent part of the isosurface at the finest LOD can be

quite large. When the network bandwidth is limited, the user may have to wait for some time before seeing the shape of the isosurface.

To enable the user to see the approximate shape of the isosurface quickly, we decide to first extract the isosurface at some coarse LOD, and then refine one level at a time till the final LOD is used, as shown in Figure 4.2. Usually the number of active cells in the dataset is $O(n^2)$ where n is the size of the dataset in one dimension. Therefore when the isosurface becomes one level coarser, the size of its representation decreases by a factor of 4 on average. This means even if we start from the coarsest LOD possible, the amount of data to be extracted and transmitted will increase by 33% at most. In practice, we generally start from 2 levels coarser than the final LOD.

Our base algorithm gives 99% correct results very quickly, but the full computation takes longer. This is because we need to cast one ray per pixel to guarantee the correctness of the final results. At the early stage of the computation, when only 1% of the rays are cast, most visible parts of the isosurface are discovered and extracted. For the rest of the computation, most of the rays are cast, only to find that they either do not intersect the isosurface or intersect a part of the isosurface that has already been extracted. Only very few triangles are extracted to make the final result correct.

In our new extraction engine for remote visualization, we need not be concerned with correctness for all the LODs except the final one. We only cast 0.1% - 1% of the total rays for each LOD except for the final one. In the final pass, the original algorithm is used and one ray per pixel is cast.

In our experiments, on top of the base algorithm plus primitive compression, applying the progressive LOD technique increases the total running time by about 21%

because of the extra primitives extracted, but reaches the 90% correctness point in 40% of the original time.

4.1.4 Primitive caching

A common scenario in isosurface visualization is that the user keeps the isovalue unchanged across a number of frames and only changes the viewpoint. If the client caches all the primitives it has received for this isovalue, no primitive needs to be extracted and transmitted for more than once. However, rendering all the cached primitives is not desirable because of the potentially large size of the isosurface. We want to render only a view-dependent subset of the cached primitives for each frame. To manage each primitive and compute exact visibility individually is computationally very expensive.

Our approach is to use the group as the basic unit for rendering decision and only compute approximate visibility. For each cached group, the extraction engine records the seed cell. The server also keeps track of the ID of the group each visited cell belongs to. When the client sends a request with the same isovalue, the server first casts one ray from the eye towards each seed, records the groups that intersect those rays, and sends their IDs to the client for rendering. Then the server starts regular computation. If a ray hits a cached group that has not been rendered by the client yet, that ID is sent to the client. If a ray hits a part of the isosurface that has not been extracted, a new group is generated and sent to the client to be rendered and cached. In our approach, each group that is rendered is at least partially visible.

When the extraction engine extracts several different representations of the isosurface, the client receives groups of LODs in the order from coarse to fine. If there are no cached groups, the client simply renders each group and clears the depth buffer when the LOD of the group changes. This is to ensure the finer representation overwrites the coarser representation. When there are cached groups, the situation is more complicated. After the client sends out the new viewpoint, the server performs a fast visibility test and sends the client a list of all the visible groups to be rendered. This includes groups of different LODs. The client renders them in the order of refining LOD and clears the depth buffer at every LOD change. However, the server is still performing the extraction starting from a coarse LOD. A new group of that LOD may be generated and sent to the client. The client cannot render this group without rendering all the groups of finer LOD that overlap with it. To solve this problem, our solution is to let the client cache all the new groups it receives, but only render those of the final LOD. The newly extracted coarse groups will not be rendered until next frame.

In our experiments, for a new isovalue, the user gets a 90% correct result after 0.54s, and 99% after 2.61s. When the isovalue stays the same and the dataset is rotates by an angle, the user the user gets a 90% correct result after 0.08s, and 99% after 0.47s.

4.1.5 Communication Protocol

We adapted the original local visualization algorithm into an extraction engine . Given a viewpoint and an isovalue, the server extracts groups of primitives and sends them to the client. We implemented a simple client that takes user input and renders the primitives as

specified by the server. The communication protocol between the server and client are shown in Figure 4.3.

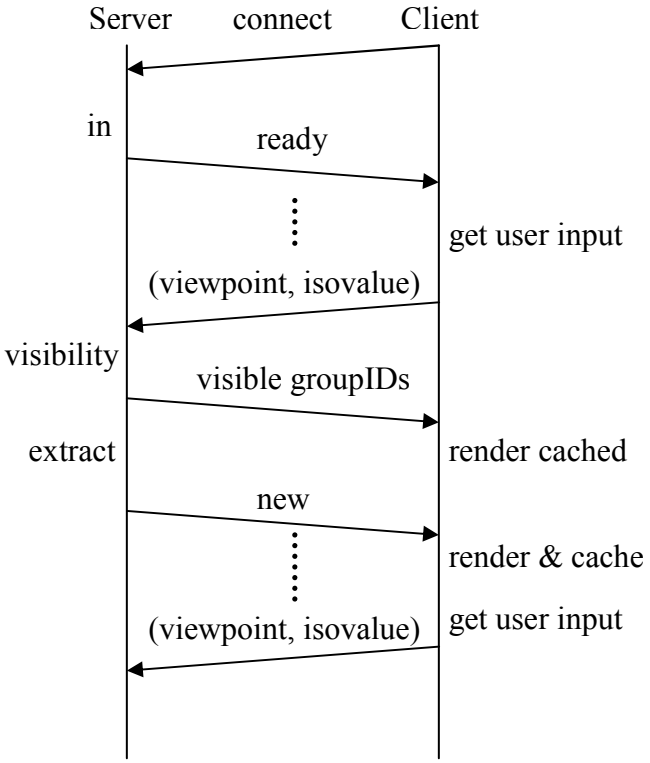


Figure 4.3: The communication protocol.

The client first connects to the server and specifies which dataset to visualize. The server reads the data in, builds auxiliary data structures, and responds with a ready message. The client keeps sending the user input viewpoint and isovalue to the server. If the isovalue is the same as last frame, the server quickly performs a visibility test and notifies the client which cached groups should be rendered. While the client renders, the server performs extraction and sends any new groups over. Whenever the user changes the view configuration, the client sends a new *(viewpoint, isovalue)* pair and interrupts the server's computation.

4.2 Results and Analysis

We would like to answer three questions:

How well does each technique work in terms of reducing the communication requirement to achieve certain level of quality of an isosurface?

What is the performance contribution of each technique for certain network configuration?

How well does the isosurface extraction algorithm suit these techniques for remote data visualization?

To evaluate the proposed algorithm, we have implemented the algorithm and tested it across three network configurations. Our experiments are conducted with the 256×256×209 Visible Woman head dataset and an 800×600 display window. The server that runs the isosurface extraction engine is a 933Mhz Pentium PC with 512 MB of main memory. The display client is a 1.5Ghz Pentium PC with an NVIDIA GeForce3 graphics card. We have experimented with three network connections between the server and the client: 100Mbps, 10Mbps Ethernet, and an emulated 1Mbps connection.

To show how each feature in our algorithm works, we will start with the base algorithm and apply the three techniques incrementally to show their effects. For each network bandwidth, we have 4 test cases:

1) Base algorithm

2) Base algorithm + primitive compression

3) Base algorithm + primitive compression + progressive LOD. The isovalue is new so there are no cached groups at the client

4) Base algorithm + primitive compression + progressive LOD. The isovalue is used once before. There are some cached groups on the client side. The new view is a rotation around the dataset from the old view. Note that only the visible groups in the last view are cached. If the isovalue has been used more times the results will become better since more of the isosurface will be cached.

To compare the performance of these test cases, we measure on the client side how the quality of the image changes with time. We are interested to know how soon the image becomes 90%, 99%, or 99.9% correct, and how much data has been transferred by then.

To quantitatively measure the correctness of the intermediate result at each sampling point in timeline, we compare the current screen capture image with the final and correct image and compute the correctness of the intermediate image as follows:

We call a pixel that is rendered in the final image a *significant pixel*. All other pixels have the background color and do not contain any information. For each significant pixel P, we check its color in the intermediate image. If it is not rendered yet, the error $e(P)$ is defined as 1 because the pixel does not provide any information yet and is totally different from that in the final image. If it is rendered, the error $e(P)$ is simply defined as the average of the L2 (Euclidean) distances between the two colors:

$e(P)=(((R_f(P)-R_i(P))^2+(G_f(P)-G_i(P))^2+(B_f(P)-B_i(P))^2)/3)^{1/2}$, where R_f G_f B_f and R_i G_i B_i are the RGB components of pixel colors in the final image and the intermediate image respectively. The RGB components are normalized to $[0,1)$.

The total error E of the intermediate image is defined as the RMS (root-mean-square, or quadratic mean) of all the pixel-wise errors: $E = (\Sigma e(P)^2/N_{sp})^{1/2}$, where N_{sp} is the number of significant pixels.

The correctness C of the intermediate image is defined as $1-E$. C is in the $[0,1]$ range, with 0 meaning nothing has be rendered, and 1 meaning the image is completely correct.

4.2.1 Communication Requirements

Regardless of the network bandwidth used in the experiments, for a certain test case, the amount of data that has to be transferred when reaching a certain correctness is fixed. Table 4.1 compares the 4 test cases for the points where C = 90%, 99%, 99.9% and 100% respectively.

C (%)	Data Transferred (Bytes)			
	Base algorithm	With primitive compression	With progressive LOD	With cached primitives
90	16.7M	4.0M	1.0M	73
99	24.2M	6.4M	9.7M	0.3M
99.9	24.3M	6.4M	10.9M	1.2M
100	24.3M	6.4M	10.9M	1.2M

Table 4.1: Comparison of amounts of data transferred for the 4 test cases.

The table shows that the primitive compression technique reduces the data size by more than 70% without adding computational overhead for either server or client.

When the progressive LOD technique is used, because we extract several representations of the isosurface at different levels of detail, the total amount of data transferred at the end of the computation increased. However, the coarser representations are smaller. Only 1MB data is needed to make the result 90% correct, compared to 4MB in test case 2. This suggests the progressive LOD technique provides a more progressive result and is suitable for visualization with limited bandwidth.

When there are some cached groups on the client side, the amount of data that needs to be transferred is further reduced. Right after rendering all the cached groups that are visible in this view (as told by the server), the result is more than 90% correct. In the end, a moderate amount of new primitives is extracted and transferred across the network. This shows the primitive caching technique increases the interactivity and progressiveness of the visualization further.

4.2.2 100Mbps Ethernet Connection

Table 4.2 shows the time needed for each test case to reach certain correctness.

C (%)	Time Elapsed (seconds)			
	Base algorithm	With primitive compression	With progressive LOD	With cached primitives
90	4.09	1.3	0.54	0.08
99	5.9	2.26	2.61	0.47
99.9	7.41	3.71	4.78	1.54
100	7.46	3.98	4.85	2.64

Table 4.2: Time elapsed when reaching certain correctness points. 100Mbps connection.

Figure 4.4 shows how the correctness changes with time for the 4 test cases.

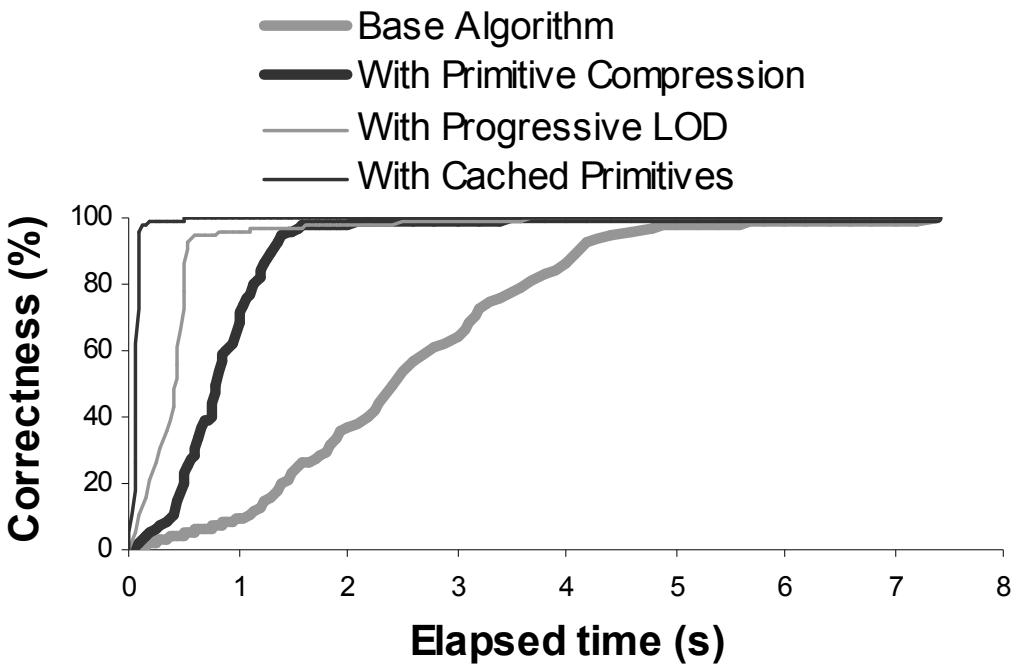


Figure 4.4: Correctness vs. time curves.100Mbps connection.

The data shows that when the primitive compression technique is added to the base algorithm, the time needed for the result to become 90% and 99.9% correct is reduced by 66% and 50% respectively.

When the progressive LOD technique is used, because more primitives need to be extracted, transferred, and rendered, the total running time increases. However, because the coarser representations are faster to transfer and render, the time needed for the result to become 90% correct is reduced by an additional 60%. The correctness curves for test case 2 and 3 intersect at point (1.6s, 97.5%). Test case 3 outperforms test case 2 in the period up to this point and provides the user with a more progressive result.

When the client has some cached primitives, the visible ones among them get rendered almost immediately and provide an approximate result that is refined by new primitives sent by the server. This explains why the correctness curve for test case 4 shifts extremely to the left. The time need to reach 90% correctness is reduced by another 85%.

4.2.3 10Mbps Ethernet Connection

We repeated the experiments for the 4 test cases with a 10Mbps Ethernet connection to investigate how network bandwidth affects the quality of visualization.

Table 4.3 shows the time needed for each test case to reach certain correctness.

C (%)	Time Elapsed (seconds)			
	Base algorithm	With primitive compression	With progressive LOD	With cached primitives
90	16.89	4.35	1.39	0.23
99	24.56	7.11	11.05	0.93
99.9	26.11	8.62	14.32	2.89
100	26.38	8.68	14.6	3.98

Table 4.3: Time elapsed when reaching certain correctness points. 10Mbps connection.

Figure 4.5 shows how the correctness changes with time for the 4 test cases. Figure 4.5 is very similar to Figure 4.4 in shape, only that the improvements from each applied technique are larger in percentage.

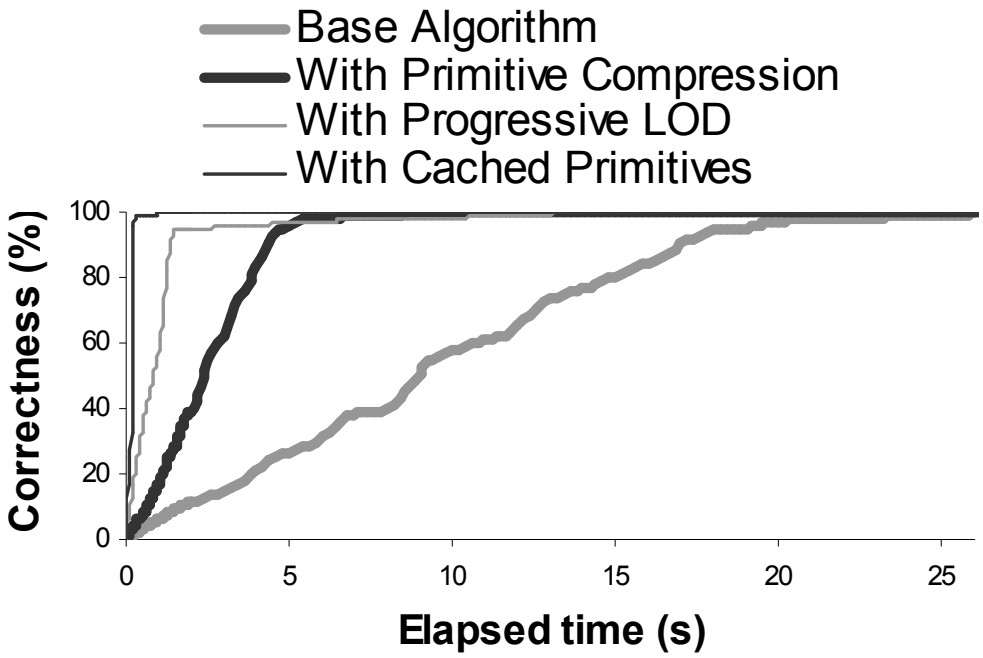


Figure 4.5: Correctness vs. time curves. 10Mbps connection.

4.2.4 Emulated 1Mbps Connection

C (%)	Time Elapsed (seconds)			
	Base algorithm	With primitive compression	With progressive LOD	With cached primitives
90	153.87	37.38	9.13	0.23
99	224.63	60.22	90.4	3.31
99.9	226.7	61.89	103.51	12.41
100	226.98	61.95	103.79	13.49

Table 4.4: Time elapsed when reaching certain correctness points. 1Mbps connection.

We emulated a 1Mbps connection by adding appropriate delays after sending each packet on the server side.

Table 4.4 shows the time needed for each test case to reach certain correctness.

Figure 4.6 shows how the correctness changes with time for the 4 test cases. Its shape is similar to both Figure 4.4 and Figure 4.5, with each incrementally applied technique bringing an even larger speedup ratio. This suggests that the three techniques we propose offer more benefits in a lower-bandwidth configuration.

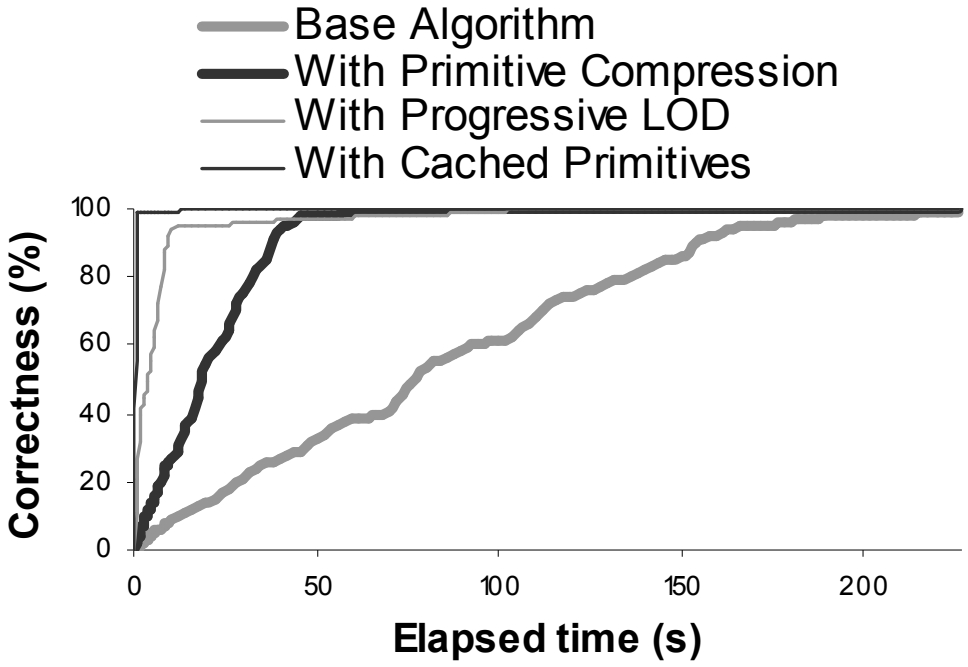


Figure 4.6: Correctness vs. time curves. 1Mbps connection.

4.2.5 Discussion

Comparing Figure 4.4 through Figure 4.6, we can see that they are all of similar shape but stretched on the time axis. When the bandwidth decreases 10-fold from 100Mbps, the

effect is that the total running time increases approximately 3-fold. This implies that the network is becoming the bottleneck link, and the time spent on network communication and computation/rendering are roughly the same. When the bandwidth decreases another 10-fold, the visualization time increases about 9-fold. This implies that in the 1Mbps-10Mbps bandwidth range, network is the dominating bottleneck and most of the running time is spent on network communication. Our future work on remote visualization should be focused on reducing network traffic.

We are interested in knowing whether our base algorithm is suitable for the three techniques we propose. Our base algorithm generates the view-dependent portion of the isosurface in the form of pieces of connected mesh, whose compression and simplification are well studied and efficient. Our algorithm organizes the dataset into an octree and facilitates isosurface extraction at different LODs. Our proposed concept of group is a collection of connected and spatially proximate triangles. It is an excellent unit for extracting, rendering, caching, manipulating the primitives. Other view-dependent approaches [23, 24] generate 3D primitives that are not necessarily connected or homogeneous, and therefore are not as suitable for these techniques.

4.3 Summary

This chapter designs and implements a remote isosurface visualization system. We proposed three acceleration techniques for the base algorithm and evaluated their contributions to the performance. We tested our system under three different network bandwidths.

Our algorithm is simple and effective. The three proposed techniques greatly improve the performance of the system, providing the user with highly progressive and interactive results. The improvement effects are more eminent with lower bandwidth connections.

The primitive compression technique reduces the amount of data transferred over the network by 70%. Depending on the network bandwidth, the amount of time needed to reach 90% correct point is reduced by 65%-75%.

The progressive LOD technique provides the user with a coarser and smaller representation quickly, reducing the amount of data transferred and time spent at the 90% correct point by 75% and 60%-70% respectively.

The primitive caching technique exploits the frame coherence of the isovalue. When an isovalue is used in the last frame, the server performs a fast visibility test and notifies the client which cached groups to render first. This technique reduces time needed to reach 90% correctness by more than 85%.

Combined together, these techniques reduce both the amount of data transferred and time spent when the result becomes 90% correct by two orders of magnitude.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

Isosurface extraction is a widely used method in visualizing volumetric datasets of 3D scalar fields. An ideal isosurface visualization system should be able to provide interactive visualization for very large datasets on scalable display systems that are located remotely. This dissertation explores algorithms and system design issues for building such a scalable isosurface visualization system.

To achieve interactive visualization of large datasets, this dissertation proposes a view-dependent, progressive, and resolution-insensitive isosurface visualization algorithm. Like other view-dependent isosurface extraction algorithms, this algorithm does not compute the complete isosurface before rendering the image and thus avoids the significant cost of full isosurface extraction. On the other hand, unlike any previous algorithms, this algorithm couples limited propagation with ray casting and this allows a

significant portion of visible isosurface to be extracted with relatively few number of rays. Evaluation in this dissertation shows that this approach extracts 99% of the visible isosurface much faster than the best-known approach. Rendering a large portion of isosurface in very short time is critical for interactive visualization because it gives users quick feedback on how to adjust the isovalue and view point. Limited propagation also makes the algorithm resolution-insensitive because propagation computes the underlying geometry and is largely independent of number of pixels each propagation covers.

Modern scalable display systems such as the Display Wall system can scale up to tens of millions of pixels. With such high resolution we can visualize very large datasets with both wide coverage and great details. To scale the visualization system to ever larger datasets, however, the visualization algorithm must be parallelized. Towards this end, this dissertation proposes the design and implementation of a parallel version of the aforementioned isosurface extraction algorithm working with an active distributed file system.

The third and last contribution of this dissertation is a remote isosurface visualization protocol. Three key characteristics of this protocol are primitive compression, primitive caching, and progressive level-of-details. Primitive compression combines multiple triangles into one primitive, thus avoiding redundant transmission of shared vertices and edges between adjacent triangles. Primitive caching allows the client to re-use previously computed triangles when the viewpoint changes but a subset of triangles remain visible. Experiments show that the amount of data transferred is reduced by two orders of magnitude by these two techniques.

Finally, while progressive level-of-detail itself does not reduce the amount of data transfer, it shows user a sketch of final isosurface quickly before details are computed and transmitted over network and also plays an important role in improving interactivity of remote isosurface visualization.

5.2 Future work

While algorithms and system designs presented in this dissertation significantly improve state of the art in isosurface visualization, there still remain plenty of open issues for future exploration.

5.2.1 Out-of-core Algorithm

Our algorithm is suitable for large datasets. However, currently the whole dataset is held in memory and this limits the size of the dataset we can process. The whole Visible Woman dataset is about one gigabyte in size and about the largest dataset we can process. Because surface propagation has strong data locality, we believe that it will be possible to adapt an out-of-core version of our algorithm. The auxiliary data structures such as the octree will be kept in memory. The dataset will be divided into blocks, where each block is a cube of voxels. A block will be the unit of disk read and only swapped in-core when it is hit by intersection or propagation.

5.2.2 Load-balanced Parallelization

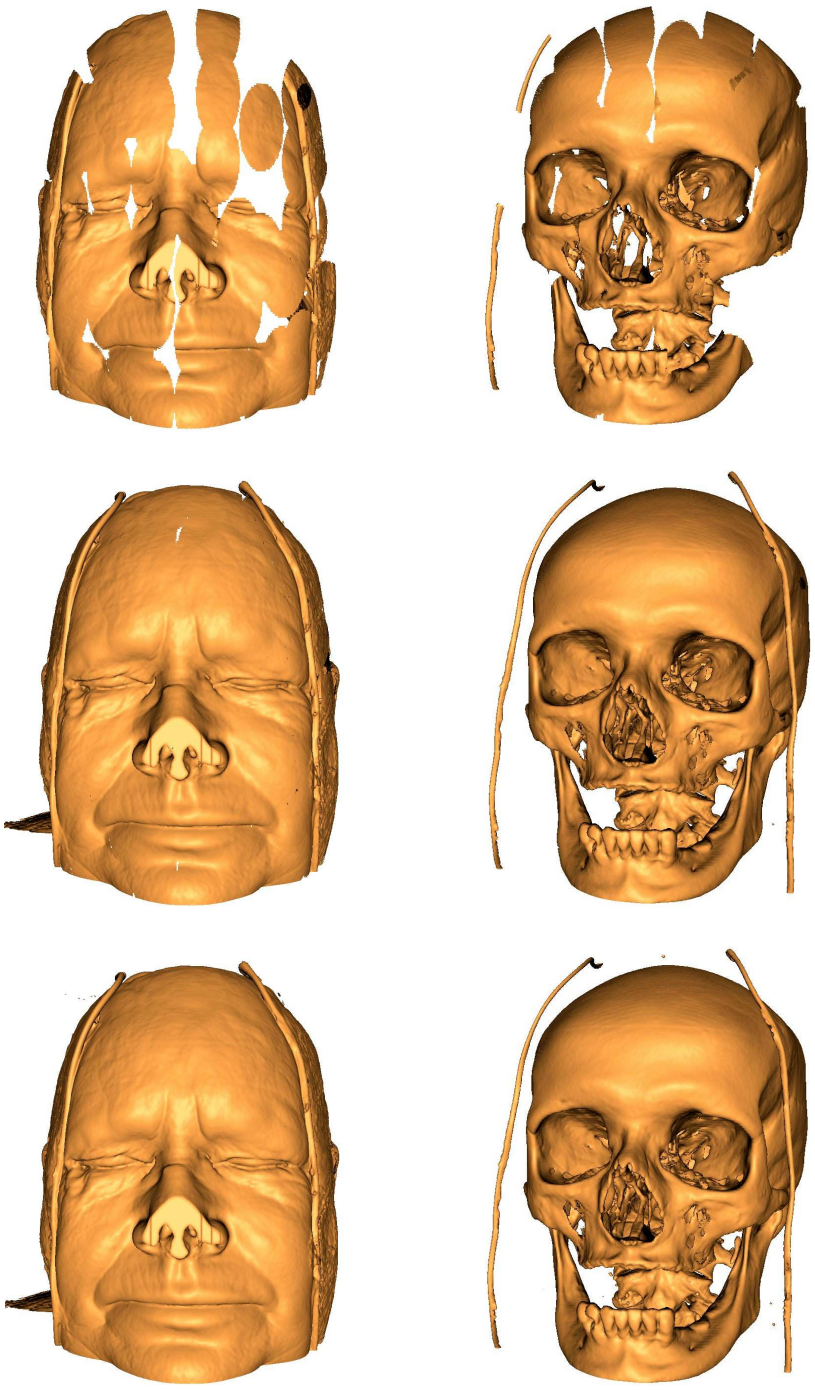
I have implemented a simple parallelization of the algorithm for tiled display driven by a cluster. This is done in master-slave fashion. A master program runs on the control

console of the Display Wall. It takes user inputs and computes the view matrix of the next frame. The master program has knowledge of the configuration of the wall and uses it to divide the screen space and compute the individual view matrices for each of the tiles. It sends the matrices, together with the isovalue, to the slave programs that runs on each of the computers that drive the tiles. The slave program uses this parameter to extract and render for its own tile. Because our approach is screen-based, the change to the algorithm is minimal.

However, limitations exist because this parallelization is not load-balanced. When the isosurface falls into only one tile, the performance degrades to that of a single computer. A load-balanced version of the algorithm will address the problem. The screen can be divided into many regions, each a unit of job. The master will be in charge of job assignment and migration.

5.2.3 Improvement in Remote Protocol

In remote visualization, integration of mesh simplification and compression techniques into the isosurface extraction process can further reduce the network traffic. Since the load of server, client, and network can be tuned by the amount of mesh simplification and compression, it is conceivable to parameterize and adjust it by the bottleneck of the system pipeline. This will provide an optimal remote visualization experience.



Color plate 1: The left column shows three progressively refined images of the skin surface generated at points A, B and C in Figure 2.6. The right column shows images of the bone surface generated at points A, B and C in Figure 2.7. In the middle row, after less than 1% of the rays have been cast, the resulting images are almost indistinguishable from the final images in the bottom row.

References

1. *LIVE: GigaPixel Project*.
2. Akkouche, S. and E. Galin, *Adaptive Implicit Surface Polygonization Using Marching Triangles.* Computer Graphics Forum, 2001. 20(2): p. 67-80.
3. Ball, R. and C. North, *Effects of tiled high-resolution display on basic visualization and navigation tasks.* Conference on Human Factors in Computing Systems, 2005: p. 1196-1199.
4. Bell, G., J. Gray, and A. Szalay, *Petascale Computational Systems.* Computer, 2006. 39(1): p. 110-112.
5. Bethel, W., *Visualization dot com.* Computer Graphics and Applications, IEEE, 2000. 20(3): p. 17-20.
6. Bethel, W., et al., *Using High-Speed WANs and Network Data Caches to Enable Remote and Distributed Visualization.* Proceeding of the IEEE Supercomputing 2000 Conference, Nov, 2000.
7. Christiansen, H. and T. Sederberg, *Conversion of complex contour line definitions into polygonal element mosaics.* ACM SIGGRAPH Computer Graphics, 1978. 12(3): p. 187-192.
8. Cignoni, P., et al., *Speeding Up Isosurface Extraction Using Interval Trees.* IEEE Transactions on Visualization and Computer Graphics, 1997. 3(2): p. 158-170.

9. Cronin, P., *2D or Not 2D That is the Question, But 3D is the Answer.* Academic Radiology, 2007. 14(7): p. 769-771.

10. Engel, K., R. Westermann, and T. Ertl, *Isosurface extraction techniques for Web-based volume visualization.* Proceedings of the conference on Visualization'99: celebrating ten years, 1999: p. 139-146.

11. Fuchs, H., Z. Kedem, and S. Uselton, *Optimal surface reconstruction from planar contours.* Communications of the ACM, 1977. 20(10): p. 693-702.

12. Garland, M. and P. Heckbert, *Surface simplification using quadric error metrics.* Proceedings of the 24th annual conference on Computer graphics and interactive techniques, 1997: p. 209-216.

13. Globus, A., *Octree Optimization.* Symposium on Electronic Imaging Science and Technology, 1991.

14. Heiland, R.W., M.P. Baker, and D.K. Tafti, *VisBench: A Framework for Remote Data Visualization and Analysis* in *Proceedings of the International Conference on Computational Science-Part II* 2001 Springer-Verlag. p. 718-727

15. Hilton, A., et al., *Marching triangles: range image fusion for complex object modelling.* Image Processing, 1996. Proceedings., International Conference on, 1996. 1.

16. Hoppe, H., *Progressive meshes.* Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, 1996: p. 99-108.

17. Hoppe, H., *View-dependent refinement of progressive meshes.* Proceedings of the 24th annual conference on Computer graphics and interactive techniques, 1997: p. 189-198.

18. Itoh, T. and K. Koyamada, *Automatic Isosurface Propagation Using an Extrema Graph and Sorted Boundary Cell Lists.* IEEE Transactions on Visualization and Computer Graphics, 1995. 1(4): p. 319-327.

19. Itoh, T., Y. Yamaguchi, and K. Koyamada, *Volume thinning for automatic isosurface propagation.* Proceedings of the 7th conference on Visualization, 1996: p. 303-310.

20. Keppel, E., *Approximating Complex Surfaces by Triangulation of Contour Lines.* IBM Journal of Research and Development, 1975. 19(1): p. 2-11.

21. Kreveld, M.v., et al., *Contour trees and small seed sets for isosurface traversal.* Proceedings of the thirteenth annual symposium on Computational geometry, 1997: p. 212-220.

22. Li, K., et al., *Building and Using A Scalable Display Wall System.* IEEE Comput. Graph. Appl., 2000. 20(4): p. 29-37.

23. Livnat, Y., *Noise, wise and sage: algorithms for rapid isosurface generation.* Ph.D Dissertation, University of Utah, 1999.

24. Livnat, Y. and C. Hansen, *View dependent isosurface extraction* in *Proceedings of the conference on Visualization '98* 1998 IEEE Computer Society Press: Research Triangle Park, North Carolina, United States p. 175-180

25. Livnat, Y., H.-W. Shen, and C.R. Johnson, *A Near Optimal Isosurface Extraction Algorithm Using the Span Space.* IEEE Transactions on Visualization and Computer Graphics, 1996. 2(1): p. 73-84.

26. Lorensen, W.E. and H.E. Cline, *Marching cubes: A high resolution 3D surface construction algorithm*, in *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*. 1987, ACM Press. p. 163-169.

27. Matveyev, S., *Approximation of isosurface in the Marching Cube: ambiguity problem.* Visualization, 1994., Visualization'94, Proceedings., IEEE Conference on, 1994: p. 288-292.

28. Möller, T. and B. Trumbore, *Fast, minimum storage ray-triangle intersection.* Journal of Graphics Tools, 1997. 2(1): p. 21-28.

29. Montani, C., et al., *Discretized Marching Cubes.* Visualization, 1994., Visualization'94, Proceedings., IEEE Conference on, 1994: p. 281-287.

30. Müller, H. and M. Stark, *Adaptive generation of surfaces in volume data.* The Visual Computer, 1993. 9(4): p. 182-199.

31. Müller, H. and M. Wehle. *Visualization of Implicit Surfaces Using Adaptive Tetrahedrizations*. in *dagstuhl*. 1997.

32. Ni, T., D. Bowman, and J. Chen, *Increased display size and resolution improve task performance in Information-Rich Virtual Environments.* Proceedings of the 2006 conference on Graphics interface, 2006: p. 139-146.

33. Nielson, G. and B. Hamann, *The asymptotic decider: resolving the ambiguity in marching cubes.* Visualization, 1991. Visualization'91, Proceedings., IEEE Conference on, 1991: p. 83-91.

34. Ohlberger, M. and M. Rumpf, *Hierarchical and adaptive visualization on nested grids.* Computing, 1997. 59(4): p. 365-385.

35. Parker, S., et al., *Interactive ray tracing for isosurface rendering* in *Proceedings of the conference on Visualization '98* 1998 IEEE Computer Society Press: Research Triangle Park, North Carolina, United States p. 233-238

36. Sandstrom, T., C. Henze, and C. Levit, *The hyperwall.* Coordinated and Multiple Views in Exploratory Visualization, 2003. Proceedings. International Conference on, 2003: p. 124-133.

37. Schroeder, W., J. Zarge, and W. Lorensen, *Decimation of triangle meshes.* ACM SIGGRAPH Computer Graphics, 1992. 26(2): p. 65-70.

38. Shekhar, R., et al., *Octree-based decimation of marching cubes surfaces.* Visualization, 1996. 96: p. 335-342.

39. Shu, R., C. Zhou, and M. Kankanhalli, *Adaptive marching cubes.* The Visual Computer, 1995. 11(4): p. 202-217.

40. Wallace, G., et al., *Tools and applications for large-scale display walls.* IEEE Computer Graphics and Applications, 2005. 25(4): p. 24-33.

41. Westermann, R., L. Kobbelt, and T. Ertl, *Real-time exploration of regular volume data by adaptive reconstruction of isosurfaces.* The Visual Computer, 1999. 15(2): p. 100-111.

42. Wilhelms, J. and A.V. Gelder, *Octrees for faster isosurface generation.* ACM Transactions on Graphics, 1992. 11(3): p. 201-227.

43. Wyvill, G., C. McPheeters, and B. Wyvill, *Data structure for soft objects.* The Visual Computer, 1986. 2(4): p. 227-234.