SCI INSTITUTE
TECHNICAL REPORT

NOISE, WISE and SAGE: Algorithms for Rapid Isosurface Generation

Yarden Livnat

UUSCI-1997-001

Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA

December, 1999

Abstract:

Exploratory scientific visualization is a valuable paradigm for understanding complex physical phenomena. When these phenomena have associated volumetric scalar fields, isosurface extraction is a critical tool. An isosurface is the set of points in which the scalar field has a particular value, the isovalue. The position of an isosurface, as well as its relation to other neighboring isosurfaces, can provide clues to the underlying structure of the scalar field. In medical imaging applications, isosurfaces help extract particular anatomical structures and tissues. These isosurfaces are static in nature. Many applications require a more dynamic use of isosurfaces. In these applications scientists need to interactively change the isovalue in order to gain better insight into simulation results. This research develops isosurface generation algorithms that enable rapid exploration of large datasets both for local and remote visualization. The work focuses on fast localization of an isosurface within very large datasets, as well as giving scientists prompt feedback during exploratory sessions. This work examines the entire isosurface generation process and the components that make up that process. The goal of this work is to isolate bottlenecks in the isosurface generation process and to develop methods with which to address them.

Two theoretical analyses of the isosurface extraction process are presented, as well as three isosurface extraction methods which are based on these analysis. The first extraction method, termed NOISE, provides an extremely fast way of locating the data cells that intersect the isosurface. The NOISE method is based on a new representation we termed the Span Space, and which is the result of the first theoretical analysis. The second analysis of the extraction process leads to the notion of a view-dependent isosurface extraction. In this approach, only the visible portion of an isosurface is extracted based on both the view parameters and on the actual screen resolution. To methods, WISE and SAGE, which are based on the view-dependent approach are also presented. These methods provide rapid responses to a remote (and local) user and enable one to view and manipulate the visible portions of the isosurface in interactive rates. Results of these isosurface extraction algorithms are presented.

THE UNIVERSITY OF UTAH

NOISE, WISE AND SAGE: ALGORITHMS FOR RAPID ISOSURFACE GENERATION

by

Yarden Livnat

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

The University of Utah

December 1999

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Yarden Livnat

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

	Chair: Christopher R. Johnson
	Charles D. Hansen
	Robert S. MacLeod
	Peter S. Shirley
	Jamie S. Painter

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of Yarden Livnat in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Christopher R. Johnson
Chair, Supervisory Committee

Approved for the Major Department

Robert R. Kessler
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

Exploratory scientific visualization is a valuable paradigm for understanding complex physical phenomena. When these phenomena have associated volumetric scalar fields, isosurface extraction is a critical tool. An isosurface is the set of points in which the scalar field has a particular value, the "isovalue." The position of an isosurface, as well as its relation to other neighboring isosurfaces, can provide clues to the underlying structure of the scalar field. In medical imaging applications, isosurfaces help extract particular anatomical structures and tissues. These isosurfaces are static in nature. Many applications require a more dynamic use of isosurfaces. In these applications scientists need to interactively change the isovalue in order to gain better insight into simulation results.

This research develops isosurface generation algorithms that enable rapid exploration of large datasets both for local and remote visualization. The work focuses on fast localization of an isosurface within very large datasets, as well as giving scientists prompt feedback during exploratory sessions. This work examines the entire isosurface generation process and the components that make up that process. The goal of this work is to isolate bottlenecks in the isosurface generation process and to develop methods with which to address them.

Two theoretical analyses of the isosurface extraction process are presented, as well as three isosurface extraction methods which are based on these analysis. The first extraction method, termed NOISE, provides an extremely fast way of locating the data cells that intersect the isosurface. The NOISE method is based on a new representation we termed the Span Space, and which is the result of the first theoretical analysis. The second analysis of the extraction process leads to the notion of a view-dependent isosurface extraction. In this approach, only the visible portion of an isosurface is extracted based on both the view parameters and on the actual screen resolution. To

methods, WISE and SAGE, which are based on the view-dependent approach are also presented. These methods provide rapid responses to a remote (and local) user and enable one to view and manipulate the visible portions of the isosurface in interactive rates. Results of these isosurface extraction algorithms are presented.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I thank the members of the Scientific Computing and Imaging research group for their support and valuable insights. I also thank the members of the Advanced Computing Laboratory at the Los Alamos National Laboratory for their helpful comments and support during my stay there in the summer of 1995. This research was made possible, in part, by grants from The National Science Foundation,the Department of Energy and the Utah Centers of Excellence. Finally, I wish to thank the members of my committee for their valuable input and support and above all their encouragement.

CHAPTER 1

INTRODUCTION

Definition 1 (Isosurface)

Given a scalar field, an isosurface is the collection *of all the points with a given value (the isovalue).*

Definition 2 (Isosurface Extraction)

Given a set of samples of a scalar field and an isovalue, find a set of surfaces that approximate *the boundary of the isosurface.*

Definition 3 (Isosurface Extraction over a Grid)

Given a set of sample of a scalar field organized in a grid *given a* rule *how to interpolate inside a grid cell, find a set of surfaces that approximate the boundary of the isosurface.*

1.1 Overview

Isosurface extraction is a powerful tool for investigating volumetric scalar fields. The position of an isosurface, as well as its relation to other neighboring isosurfaces, can provide clues to the underlying structure of the scalar field as seen in Figure 1.1. In medical imaging applications (a), isosurfaces permit the extraction of particular anatomical structures and tissues. These isosurfaces are static in nature. A more dynamic use of isosurfaces is called for in many scientific computing applications, such as mantle convection simulation (b) and interpretation of seismic data (c). In these applications scientists need the ability to change the isovalue dynamically in order to gain better insight into simulation results.

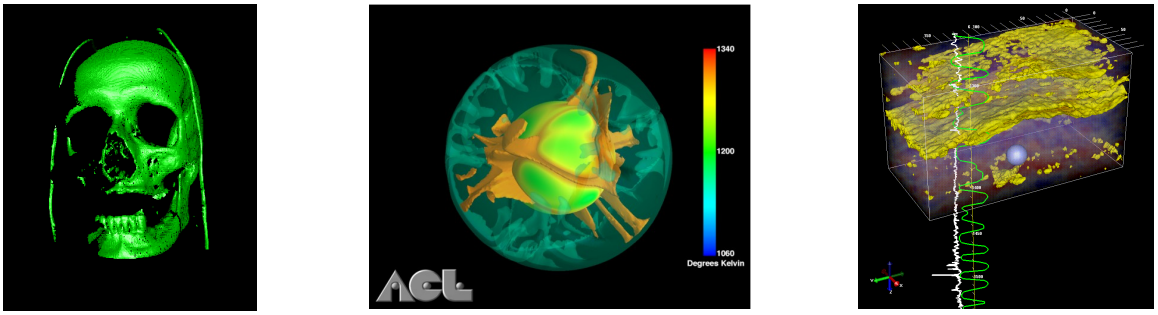


Figure 1.1. Medical and scientific computing applications of isosurfaces.(a) A skull from the Visible Woman project. (b) Mantle convection. Hot mantle (orange) flow upward to the surface while the colder mantle (transparent cyan) descent toward the earth core. (c) Imaging of seismic data. Two isosurfaces of a constant magnitude are shown embedded in a volume visualization of the data. A single trace and an SP-log curves at one of the wells are also shown.

The push for higher degrees of accuracy in scientific computing applications and high resolution medical scanners has caused data sets to grow ever larger. The sheer size of these data sets poses a major obstacle for interactive investigation. An isosurface may be composed of many complex but separated components. Scanning very large datasets and isolating these components can take several minutes even on supercomputers. Once isolated, a geometric representation of the isosurface needs to be generated and rendered onto the user screen. Yet the size (number of polygons) of the extracted isosurface can overwhelm even high-end graphics hardware accelerators.

Another obstacle is posed by the structure of the data. While data in medical imaging are provided at structured grid positions, scientific data sets frequently consist of geometry represented by unstructured finite element grids. Structured grids require less memory per data point as the locations of the data cells are known implicitly from their index. In addition, structured data sets enable the use of simple traversal methods and are well suited for hierarchical representations. Unstructured grids provide the scientist with more freedom in representing complex environments. The visualization of such grids is nevertheless much harder.

The last decade has seen the rise of the desktop computer. Today's desktop computers provide more raw CPU power, memory and better graphics hardware than a

supercomputer of 10 years ago. This in turn enables scientists and medical personnel to conduct more of their research on their own personal computers. Small to moderate data sets can now be processed and visualized right at the scientist's desk. Along with the improvement in desktop computer power, the latest supercomputers are now massive parallel processors with hundreds of processors and tens of gigabytes of memory. The enormous power of these supercomputers comes at a very high cost, which results in the centralization of supercomputers in only a few sites, such as the national laboratories

The availability of relatively cheap yet powerful desktop computers and massive parallel supercomputers in only a few locations has led to the development of a new visualization paradigm, namely remote visualization. The fundamental drive is to enable the scientist to perform very large simulations on a remote supercomputer while visualizing and investigating the results on the local desktop. For the remote visualization paradigm to be successful, several technical obstacles need to be addressed. These obstacles include remote steering of the simulation and visualization, generation of visualization cues (e.g., isosurface extraction) and transmission and visualization over the Internet.

1.2 Aims and Objectives

The goal of this research was to develop isosurface generation algorithms that would enable rapid exploration of large datasets both for local and especially remote visualization. The work focuses on fast localization of an isosurface within very large datasets, as well as providing prompt feedback to the scientist during an exploratory session.

This work examines the entire isosurface generation process and the components that make up that process. One of the tasks in this work was to isolate bottlenecks in the isosurface generation process and to develop methods with which to address them. Further goals were developing fast isosurface localization algorithms and providing efficient and rapid, if only partial, results to the user.

1.3 Organization

The thesis is comprised of seven chapters and two appendices.

Chapter 2 presents a brief history of the evolution of isosurface extraction methods. This chapter also details those methods which are relevant to this work. In chapter 3

we introduce a new taxonomy of isosurface extraction methods and a new view of the underlying domain which we call the *span space*. We then employ the span space as a common backdrop for comparing the pros and cons of some of previous extraction methods which were introduced in Chapter 2 as well as some of the new methods which where developed based on the new span space perspective. While the span space provides a new theoretical perspective of the underlying domain, it does not specify any particular way of extracting isosurfaces. With the introduction of the span space we also proposed a partition scheme of this space and a new method for fast localization of isosurfaces based on this partition scheme. Chapter 4 examine details this binary partition scheme which is based on a kd-tree. The chapter also cover our new *Near Optimal Iso Surface Extraction*(NOISE) method, which is suitable for both structured and unstructured grids. The span space and the NOISE method enable rapid localization of isosurfaces but do not address the visualization of the extracted isosurface. In Chapter 5 we examine isosurface extraction issues in the context of a complete visualization process. Based on this analysis, we then propose a new view-dependent approach to isosurface extraction in large and complex datasets. Chapter 6 lays down the foundations for view dependent algorithms. A general three step approach is presented first, followed by two implementations. The first implementation accelerates the occlusion tests by examining a warped projection of the iso surface during the extraction and is termed WISE *(Warped IsoSurface Extraction)*. The second method improves on the WISE method (and as such was termed SAGE) by providing early termination conditions as well as extracting triangles in parallel. Chapter 7 summarizes this work and discusses future research directions.

Appendix A presents a worst case analysis for two isosurface extractions algorithm. Appendix B demonstrates the cost of point projection in the Warped IsoSurface Extraction (WISE) method.

CHAPTER 2

BACKGROUND

This chapter examines recent innovations in the field of modern isosurface extraction methods. Iso-contouring methods have been in use for a long period of time, but the major breakthrough in the field occurred only recently with the works by Wyvill et al. [30] in 1986 and Lorensen et al. [18] in 1987. The success of these methods stems from their divide-and-conquer strategies. These methods convert the complex global isosurface extraction task into many small and simple local triangulations. The visualized data is often generated or acquired from three-dimensional images or as solutions to numerical approximation techniques, such as from finite difference or finite element methods. These acquisition methods represent the data as a set of polyhedral cells where the data points define the vertices. Divide-and-conquer in this context amounts to replacing the global isosurfacing problem with a local contouring of each of the dataset cells.

Since the introduction of these methods, much effort has been focused on both the divide (search for those cells that intersect the isosurface) and the conquer (local triangulation) phases. The discovery of an ambiguity in the original Marching Cubes lookup table led to more research on the local modeling of the isosurface of each of the cells. With the work by Wilhelms and Van Gelder [29] in 1990 the effort has shifted mainly to the search phase. In recent years, with the introduction of very large datasets and efficient search algorithms such as NOISE [17] (described in Chapter 4), emphasis has shifted toward reducing the size of the extracted isosurface. In essence, this effort is directed at the last phase of the divide-and-conquer strategy, namely the gather phase. It is in this last phase that the all the local results are combined into one final global solution.

2.1 Isosurface Extraction Methods

2.1.1 Marching Cubes

The best known isosurface extraction method to achieve high resolution results is *Marching Cubes* [30, 18]. The novelty of the method is the notion of cell independence such that the isosurface can be extracted locally on a per cell basis. The marching cubes method concentrates on the approximation of the isosurface inside the cells rather than on efficient localizations of the involved cells, Figure 2.1. To this end, the marching cube method scans the *entire* cell set, one cell at a time. Each of the cell eight nodes is compared with the given isovalue to form an eight-bit entry into a small pre-computed lookup table that describes how to triangulate that cell. The original marching cube algorithm used a compressed lookup table of 16 entries.

Later research focused on an ambiguity, see Figure 2.2 that was discovered in the original marching cubes lookup table [19, 21, 28]. This ambiguity can lead to a topologically inconsistent isosurface and the introduction of artificial holes. The inconsistency was resolved by using a full lookup table with entries for all the 256 possible cases.

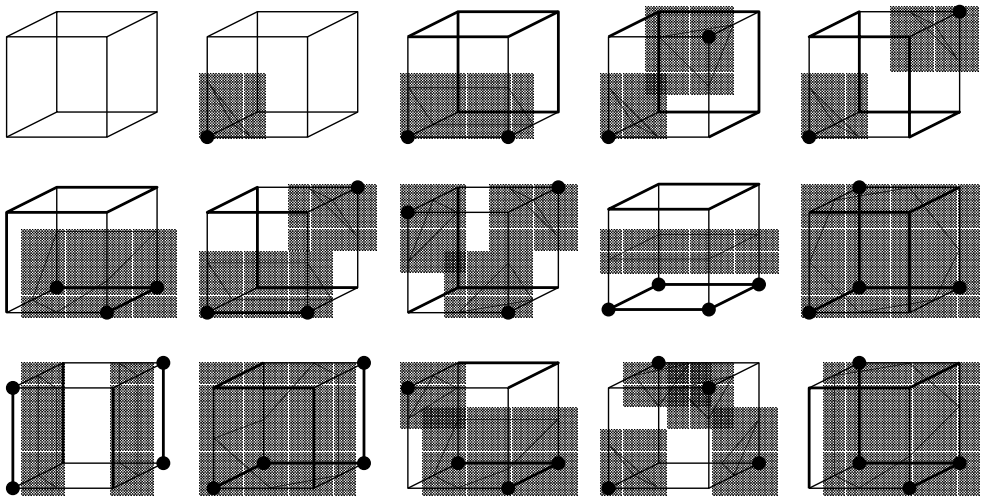


Figure 2.1. The marching cubes triangulation of a cube. The circles mark all the nodes with values which are either larger or smaller than the isovalue. Dark lines represent edges which are not intersected by the isosurface.

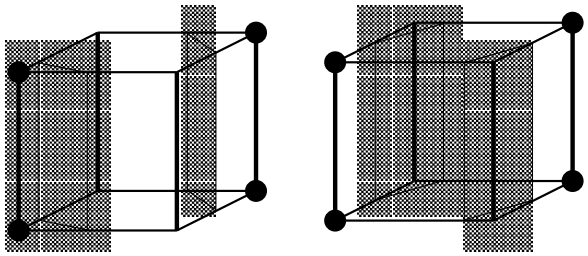


Figure 2.2. Ambiguity: The cube can be divided by more than one way.

2.1.2 Octrees

The marching cubes method did not attempt to optimize the time needed to search for the cells that actually intersect the isosurface. This need was later addressed by Wilhelms and Van Gelder [29], who employed an octree data structure, effectively creating a three-dimensional hierarchical decomposition of the cell set. Each node in the tree was tagged with the minimum and maximum values of the cells it represented. These tags, and the hierarchical nature of the octree, enabled one to trim off sections of the tree during the search and thus restrict the search to only a portion of the original geometric space. Wilhelms and Van Gelder did not analyze the time complexity of the search phase of their algorithm. However, octree decompositions are known to be sensitive to the underlying data. If the underlying data contains some high frequency fluctuations or noise, most of the octree will have to be traversed. Figure 2.3 is an example of such a dataset, which ultimately undermines any geometric decomposition scheme. In Appendix A.1, we present an analysis of the octree algorithm and show that the algorithm has a worst case complexity of $O(k + k \log n/k)$, where n is the size of the data and k is the size of the extracted isosurface. Finally, octrees have primarily been applied to structured grids and are not easily adapted to deal with unstructured grids.

2.1.3 Extrema Graphs

Recently, Itoh and Koyamada [12] presented a new method for generating isosurfaces over unstructured grids using *extrema graphs*. The search starts at a *seed* cell known to intersect the isosurface, and propagates recursively to its neighbor cells. Knowing how the isosurface intersects the current cell enables the algorithm to move only to those neighbor cells that are guaranteed to intersect the isosurface.

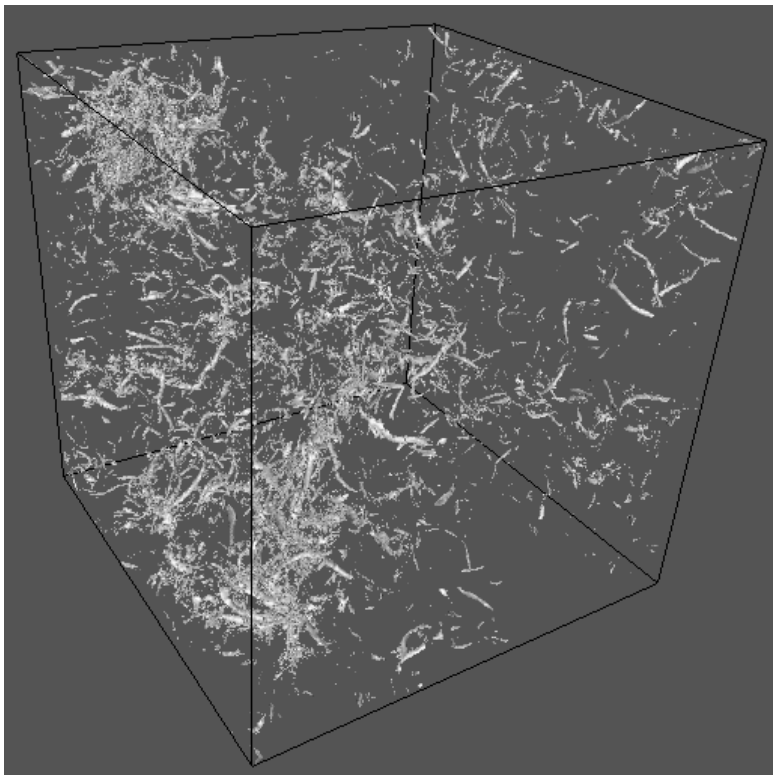


Figure 2.3. Turbulent flow in a fluid dynamic simulation representing the magnitude of fluid velocity and shows the onset of turbulence. The elongated structures are vortex tubes.

In order to find such a seed cell, Itoh and Koyamada employed extrema graphs. The nodes of these graphs are those cells that include local extrema vertices. Each arc in the graphs has a list of the cells connecting its two end nodes.

Given an isovalue, the extrema graph is first scanned to located arcs that span across the isovalue. The cells in each such arc list are then scanned sequentially until a seed cell is found. Boundary cells must also be traversed; hence the complexity of the algorithm is at best the size of the boundary list, which Itoh and Koyamada estimate as $O(n^{2/3})$.

The following argument shows that the number of arcs might be $O(n)$ in the worst case. Such cases occur when the data exhibit small perturbations such that most of the nodes are local extrema. In such a case, the numbers of arcs in the extrema graph can be equal to the number of cells, though each arc will contain only a single cell.

Storage requirements for the extrema graph method can be high, since the propagation search requires four links from each cell to its neighbors in addition to the maximum

and minimum values of its vertices. In addition, the algorithm uses a queue during the propagating search, yet the maximum required size of the queue is unknown in advance.

2.1.4 Volume Thinning

One of the disadvantages of the extrema graphs was the need to scan all the boundary cells in addition to the arcs of the graphs during a search. Itoh et al. [13] improved on the extrema graphs method by replacing the extrema graphs with a volumetric skeleton by applying a thinning algorithm commonly used in image processing. The volumetric skelaton preserved the topological features of the volume and connectivity of the extrema points.

The volumetric skeleton method removed the dependency of the extrema graphs method on the boundary cells and has better overall performance compared to the extrema graphs.

2.1.5 Value Space Decomposition

Decomposing the value space, rather than the geometric space, has two advantages. First, the underlying geometric structure is not not used in the search phase and thus this decomposition works well with unstructured grids. Second, for a scalar field in three-dimensions, the dimensionality of the search is reduced from three to only two. Figure 2.4 depicts a typical value based approach to isosurface extraction. Each cell is represented by a segment based on the minimum and maximum values of the cell vertices, similar to the octree approach presented in Section 2.1.2. An isosurface extraction in this perspective amount to locating all the segments that are intersected by the vertical line at the given value v.

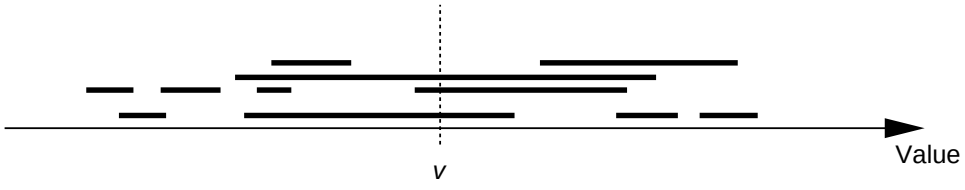


Figure 2.4. Value space decomposition. The cells are represented by line segments based on the minimum and maximum values at the cell vertices. Isosurface extraction is achieved by locating all the segments that intersect the vertical line at v.

2.1.6 The Span Filter

A key issue in isosurface extraction is the size of the dataset. Gallagher [9] addressed this issue by scanning the dataset and generating a compressed representation suitable for isosurface extraction, see Figure 2.5. The range of data values was divided into subranges, termed *buckets*. Each cell was then classified based on the bucket in which its minimum value resides and on how many buckets the cell range spans, i.e., the *span* of the cell. Cells were then grouped according to their span, and within each such group the cells are further grouped according to their starting bucket. Rather than requiring a span list for every possible span length, the method uses one span list to catch all the cells that span more than a predefined number of buckets (more than three span in the example in Figure 2.5). When an iso-value, v, is given the spans are searched one at a time. The search inside each span starts at the bucket which contains the iso-value and continues backward to previous buckets. The number of buckets searched in each span is equal to the span number. In the last span, which contains all the rest of the segments, the search continues backward until the first bucket.

2.1.7 The Active List

A different approach was taken by Giles and Haimes [10]: to find the cells that intersect an isosurface incrementally. Once an isosurface is found, a neighboring isosurface, with an isovalue close to the first one, can be found with minimal effort.

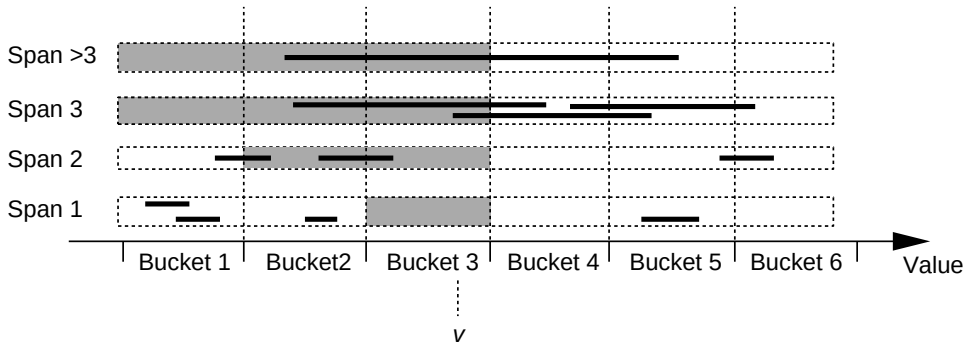


Figure 2.5. The Span Filter. The value space is divided into buckets and the segments are organized based on how many buckets they span. The search is done in each span starting at the bucket which contains the isovalue v and proceeding to previous buckets as shown in the gray areas.

The algorithm (Figure 2.6) is based on two cell lists ordered by the cell minimum and maximum values and on Δ, the global maximum range of any of the cells. When an isovalue is first given, or if the change from the previous value is greater than Δ, then an active cell list is formed. The active list is first initialized with all the cells with a minimum value between the given isovalue, v, and $v - \Delta$, by consulting the minimum list (all the segments which start in the shaded area in top on Figure 2.6). The active list is then purged of the cells with a maximum value less then v. In the given example, eight segments are put into the active list, but five of them are subsequently purged. If the isovalue is changed by less then Δ, then the active list is augmented with the cells that lie between the previous isovalue, v and the new one, nv. The new cells are found by using one of the two ordered lists, based upon whether the change was positive or negative. The active list is then purged again for the cells that do not intersect the isosurface. In the given example, the new isovalue, nv is less than the current isovalue v, and thus the list which is ordered by maximum value is consulted (bottom graph). All the segments which have their maximum value inside the shaded area, i.e. in the range (nv, v) are added to the active list (only one segment in this case). The active list (now contained four segments) is then scanned to remove the active segments (again once in this example) with a minimum value larger than nv.

The example in Figure 2.6 demonstrates one of the pitfalls of the Active List approach. In this example, one of the segments is very large and thus the global Δ is much too large. This caused the algorithm, in the first case, to add many segments into the active list which were subsequently removed. Since the algorithm has no control over the global variable Δ, one bad cell can reduce the effectiveness of the entire method.

2.1.8 Sweeping Simplices

Shen and Johnson [25], developed the Sweeping Simplices method for extracting isosurfaces from unstructured three-dimensional meshes. Their algorithm utilizes both coherence between adjacent isosurfaces and explicit space decomposition.

Sweeping Simplices uses two ordered cell lists, a *sweep list* and a *min list*, see Figure 2.7. Each element in the sweep list contains a pointer to a cell, the cell maximum value, and a flag. The sweep list is then sorted according the cell maximum value. The

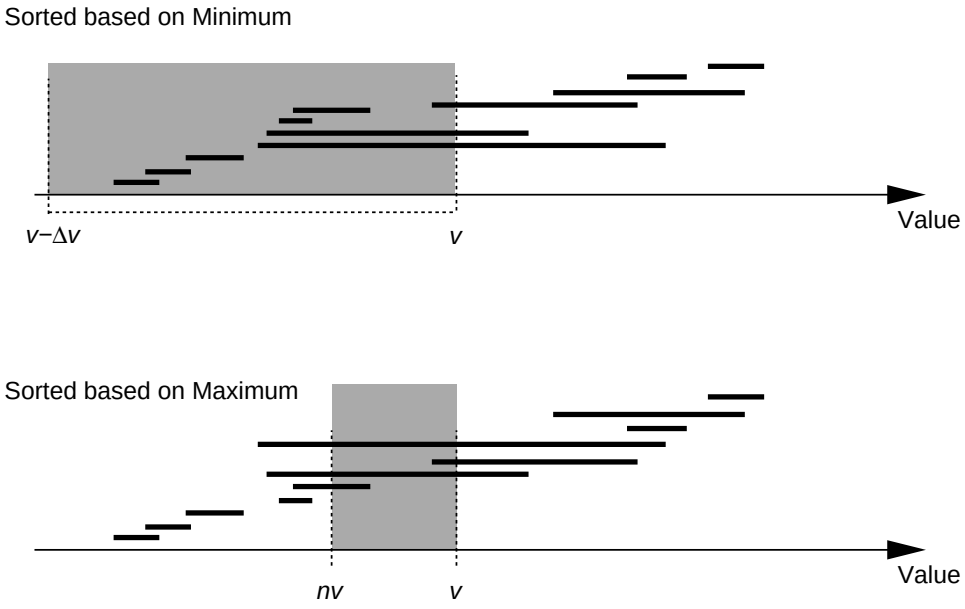


Figure 2.6. The Active List. The segments are sorted in two lists based on their minimum and maximum values. During the initialization or when the isovalue is changed by more than Δ, all the segments with a minimum value in the range $(v - \Delta, v)$ (the top shaded area) are added to the active list. The active list is than purged from all the segments for which the maximum value is less than the isovalue v. When the isovalue is changed to a new value, $nv < v$, the active list is augmented by all the segments with a maximum value in the range (nv, v) (the bottom shaded area). The active list is again purged from all the segments with a minimum value greater than nv. If $nv > v$ than the minimum list is consulted rather than the maximum list.

min list contains the minimum value for each cell as well as a pointer to the corresponding element in the sweep list and is ordered by the minimum values. The initialization step requires a time of $O(n \log n)$.

Given an isovalue, the Sweeping Simplices algorithm *marks* all the cells that have a minimum value less than the given isovalue using the min list by setting the corresponding flag in the sweep list. If an isovalue was previously given, then the min list is traversed between the previous isovalue and the new one. The corresponding flags in the sweep list are then set or reset based on whether the new isovalue is greater or smaller than the previous isovalue.

Once the flags are changed, the sweep list is traversed starting at the first cell with a maximum value greater than the new isovalue. The cells that intersect the isosurface are those cells for which their corresponding flag is set. The complexity of the algorithm is

$O(n)$ in both time and space.

The Sweeping Simplices algorithm uses a hierarchical data decomposition. At the lowest level, the range of data values is subdivided into several subgroups. Other levels are created recursively by grouping consecutive pairs from the previous level. At the top level there exists a single subgroup with the range as the entire dataset. The cells are then associated with the smallest subgroup that contains the cell. Each subgroup is then associated with a min and sweep list as previously described. Isosurface extraction is accomplished by selecting for each level the subgroup that contains the given isovalue and performing the search using its min and sweep lists.

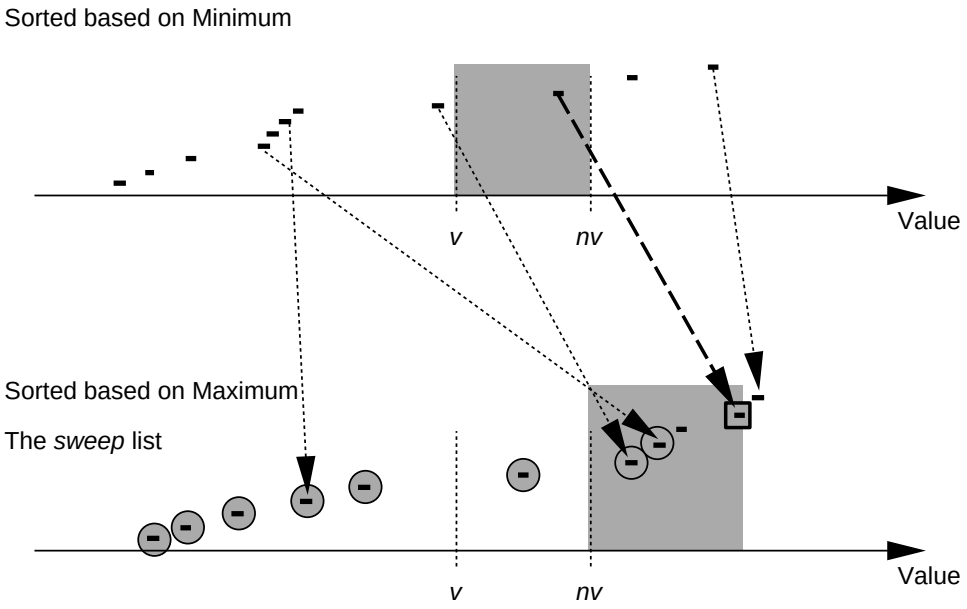


Figure 2.7. Sweeping Simplicies. Each entry in the *min* list point to the corresponding entry in the *sweep* list. After the initialization for isovalue v all the cells with a minimum value less than v are marked (a circle in the bottom graph). When the isovalue is changed to nv, the *min* list is scan between v and nv (the top shaded area) and for each entry the corresponding entry in the *sweep* list is marked (a square in bottom graph). The *sweep* list is than scan between v and nv (the shaded area in the bottom graph) and only those segments that are marked (three in this example) are those that intersect the new isovalue nv. Note that if $nv < v$ than the corresponding entries in the *sweep* will be unmarked rather than mark.

2.1.9 Other Isosurface Extraction Methods

Chapter 3 presents the *span space*, a new approach to the problem of isosurface extraction over the value space. After the introduction of the span space [17], two new methods were developed which are based on the span space [24, 7]. The descriptions of these methods, therefore, are presented after the introduction of the span space, in section 3.2.2.2 and section 3.2.2.3 respectively.

CHAPTER 3

THE SPAN SPACE

Originally, isosurface extraction methods were restricted to structured grid geometry and, as such, early efforts focused on extracting a single isosurface [18] from the volumetric data set. Recently, in an effort to speed up isosurface extraction, several methods were developed that could be adapted to the extraction of multiple isosurfaces from structured [29, 12] as well as from unstructured geometry [9, 10]. Nevertheless, for large data sets, these methods do not allow for interactive investigation of the data set, especially for unstructured grids.

Defining n as the number of data cells and k as the number of cells intersecting a given isosurface, most of the existing algorithms have time complexity of $O(n)$. While [29] has an improved time complexity of $O(k \log(\frac{n}{k}))$, the algorithm is suitable only for structured hexahedral grids.

This chapter investigates the underlying domain for structured and unstructured problems and a new decomposition of this domain, which we have called the *span space*, is proposed. The span space is then used in Section 3.2 as a common backdrop for comparing previous methods of isosurface extraction. Chapter 4 presents a new fast and efficient method which is based on this span space perspective. This near optimal isosurface extraction (NOISE) method has a worst case complexity of $O(\sqrt{n} + k)$ and is suitable for both structured and unstructured grids.

3.1 The Span Space

3.1.1 Definition

Let $\varphi : \mathbb{G} \rightarrow \mathbb{V}$ be a given field and let $\mathcal{D}$ be a sample set over φ, such that,

$$\mathcal{D} = \{d_i\} \qquad d_i \in \mathbb{D} = \mathbb{G} \times \mathbb{V}$$

where, for some $p, q \in \mathbb{Z}$, $\mathbb{G} \subseteq \mathbb{R}^p$ is a geometric space and $\mathbb{V} \subseteq \mathbb{R}^q$ is the associated value space. Also, let $d = |\mathcal{D}|$ be the size of the data set.

Definition 4 (Isosurface Extraction) *Given a set of samples $\mathcal{D}$ over a field $\varphi : \mathbb{G} \to \mathbb{V}$ and an interpolation rule, find for a given single value $v \in \mathbb{V}$,*

$$S = \{g_i\} \quad g_i \in \mathbb{G} \quad \textit{such that,} \quad \varphi(g_i) = v. \tag{3.1}$$

Note that S, the isosurface, *need not be topologically simple.*

Approximating an isosurface, S, as a global solution to Equation 3.1 can be a difficult task due to the sheer size, d, of a typical science or engineering data set.

The data samples, $\mathcal{D}$, are often generated from three-dimensional images or as solutions to numerical approximation techniques, such as from finite difference or finite element methods. These methods naturally decompose the geometric space, $\mathbb{G}$, into a set of polyhedral cells, C, where the data points define the vertices. While $n = |C|$, the number of cells, is typically an order of magnitude larger than d, the approximation of the isosurface over C becomes a manageable task. Rather than finding a global solution one can seek a local approximation within each cell. Hence, isosurface extraction becomes a two-stage process: locating the cells that intersect the isosurface and then, locally, approximating the isosurface inside each such cell. We focus our attention on the problem of finding those cells that intersect an isosurface of a specified isovalue.

On structured grids, the position of a cell can be represented in the geometric space $\mathbb{G}$. Because this representation does not require explicit adjacency information between cells, isosurface extraction methods on structured grids conduct searches over the geometric space, $\mathbb{G}$. The problem as stated by these methods is defined as follows:

Approach 1 (Geometric Search) *Given a point $v \in \mathbb{V}$ and given a set C of cells in $\mathbb{G}$ space where each cell is associated with a set of values $\{v_j\} \in \mathbb{V}$ space, find the subset of C which an isosurface, of value v, intersects.*

Efficient isosurface extraction for unstructured grids is more difficult, as no explicit order, i.e., position and shape, is imposed on the cells, only an implicit one that is difficult

to utilize. Methods designed to work in an unstructured domain have to use additional explicit information or revert to a search over the value space, $\mathbb{V}$. The advantage of the latter approach is that one needs only to examine the minimum and maximum values of a cell to determine if an isosurface intersects that cell. Hence, the dimensionality of the problem reduces to two for scalar fields.

Current methods for isosurface extraction over unstructured grids, as well as some for structured grids, view the isosurface extraction problem in the following way:

Approach 2 (Interval Search) *Given a point $v \in \mathbb{V}$ and given a set of cells represented as intervals,*

$$I = \{[a_i, b_i]\} \qquad \textit{such that,} \quad a_i, b_i \in \mathbb{V}$$

find the subset I_s such that,

$$I_s \subseteq I \qquad \textit{and,} \quad a_i \leq v \leq b_i \quad \forall (a_i, b_i) \in I_s,$$

where a norm should be used when the dimensionality of $\mathbb{V}$ is greater than one.

Posing the search problem over intervals introduces some difficulties. If the intervals are of the same length or are mutually exclusive they can be organized in an efficient way suitable for quick queries. However, it is much less obvious how to organize an arbitrary set of intervals. Indeed, what distinguishes these methods from one another is the way they *organize* the intervals rather than how they perform searches.

Definition 5 (The Span Space) *Let C be a given set of cells, define a set of points $P = \{p_i\}$ over $\mathbb{V}^2$ such that,*

$$\forall c_i \in C \qquad \textit{associate,} \quad p_i = (a_i, b_i)$$

where,

$$a_i = \min_j \{v_j\}_i \quad \textit{and} \quad b_i = \max_j \{v_j\}_i$$

and $\{v_j\}_i$ are the values of the vertices of cell i.

A key point is that *points* in two-dimension exhibit no explicit relations between themselves, whereas *intervals* tend to be viewed as stacked on top of each other, so that overlapping intervals exhibit merely coincidental links. Points do not exhibit such arbitrary ties and in this respect lend themselves to many different organizations. However, as shown later, previous methods grouped these points in very similar ways, because they looked at them from an interval perspective.

Using our augmented definition, the isosurface extraction problem can be stated as,

Approach 3 (The Span Search) *Given a set of cells, C, and its associated set of points, P, in the span space, and given a value $v \in \mathbb{V}$, find the subset $P_s \subseteq P$, such that*

$$\forall (x_i, y_i) \in P_s \quad x_i < v \quad y_i > v.$$

Note that $\forall (x_i, y_i) \in P_s, \quad x_i < y_i$ as the maximum value attained in a cell can not be smaller the the minimum value. Therefore, the associated points will lie above the line $y_i = x_i$. A geometric perspective of the span search is given in Figure 3.1.

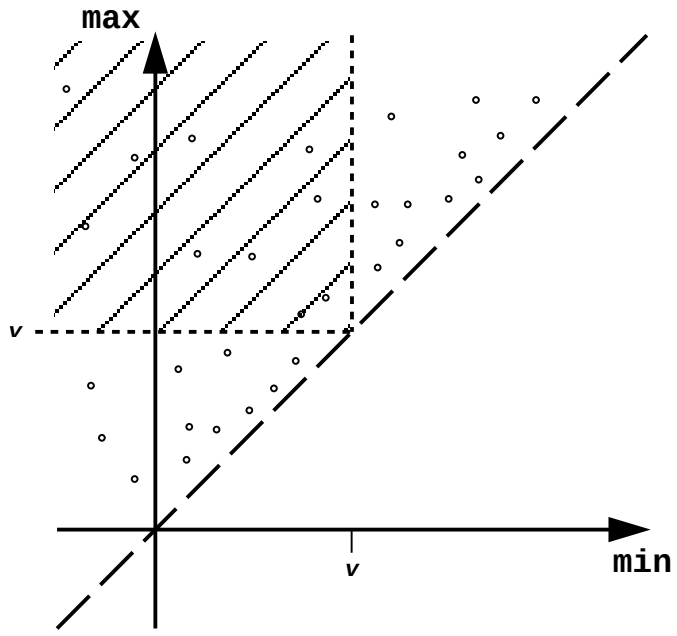


Figure 3.1. Search over the span space. Each point represents one of the dataset cells and its location is based on the minimum and maximum values attained in that cell. The semi infinite shaded area is defined by the given isovalue, v. The points that lie in this shaded area represent the data cells through which the isosurface passes

In general, the span (difference between the minimum and maximum values) of most of the cells is relatively small compare to the full range of values in the dataset. The small single cell span leads to the clustering of the points in the span space close to the line $min = max$. The clustering means that the points are not evenly distributed in the span space and areas further from the $min = max$ line contains very few points.

3.1.2 Neighborhood Search

The Sweeping Simplices and the Active List algorithms were designed to take advantage of coherence between isosurfaces with close isovalues. The interpretation over the span space of this approach is depicted in Figure 3.2. When an isovalue pv is changed to v, then the set of cells that intersect the new isosurface can be generated by adjusting the current set of cells. In essence, if $v > pv$ then we need to remove the cells that lie in the bottom rectangle and add those that lie in the right rectangle. If $v < pv$ the add and remove roles of these rectangles are reversed.

3.2 Value Space Methods over the Span Space

The span space interpretation can serve as a common ground for comparison of previous value-based search methods. This perspective can help to identify some of the pitfalls in those methods and lead to new methods which aim to avoid them. In the following section we analyze previous value space extraction methods which were published before we introduced the span space in 1996 [17]. Section 3.2.2 presents new extraction methods which were developed based on the span space. The first method (NOISE) was published in conjunction with the span space and is the subject of Chapter 4.

3.2.1 Evaluation of Previous Extraction Methods Based on Their Span Space Projection

3.2.1.1 The Span Filter

Figure 3.3 depicts the span filter organization over the *span space*. Note that the compression over the cells' i.d. is not shown. For a given isovalue, v, the cells that intersect the isosurface are those that lie above and to the left of the dashed line. As described in Section 2.1.6, the user first divides the range of values attained in the given

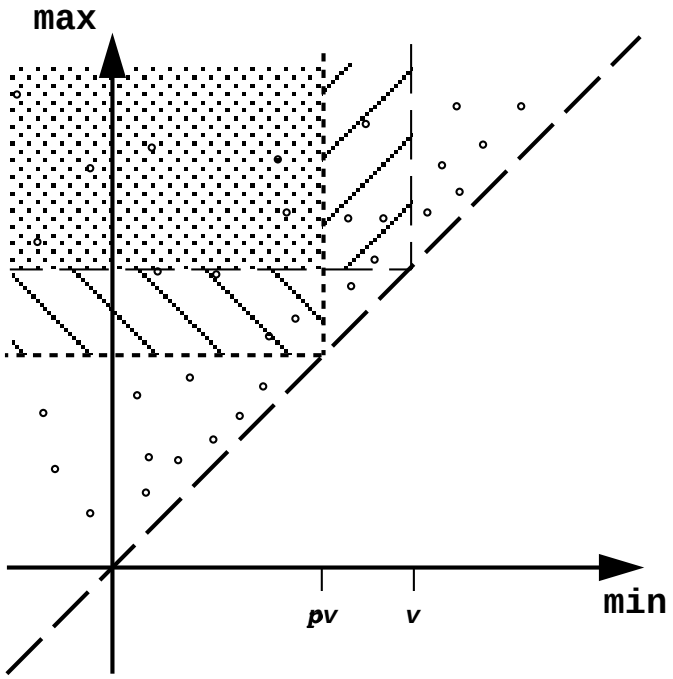


Figure 3.2. Neighborhood Search. The isovalue was changed from pv to v. Points in the central shaded area represent cells that intersect both isosurface. The bottom shared area mark the cells that are part of old isosurface while the right shared area marks cells that intersect only the new isosurface. A neighborhood search method need to isolate only those cells that are in the stripped areas to keep an updated list of cells which intersect the isosuface.

dataset into buckets (shown in the figure as buckets 1 to 7). The division is arbitrary and depends heavily on the particular dataset. The shaded areas marked as span 1 contain points which falls into a single bucket, i.e., both the minimum and maximum values falls into the same bucket. Points in shaded area marked span 2 span two adjacent buckets, i.e., the maximum is one bucket away for the bucket which contain the minimum value. Span 3 areas contain points that span three buckets whereas the single span n area contains all the points that span more than three buckets.

The use of this perspective stresses the importance of the first division into buckets. The entire organization of the domain is controlled by only one set of parameters, the position of the original buckets. This may help to ensure even distribution in the first span but it does not provide control over the distribution of the cells in the other spans. Furthermore, this division is not automated and has to be crafted by trial and error for each new data set. Finally, the search algorithm has a complexity of $O(n)$ in time.

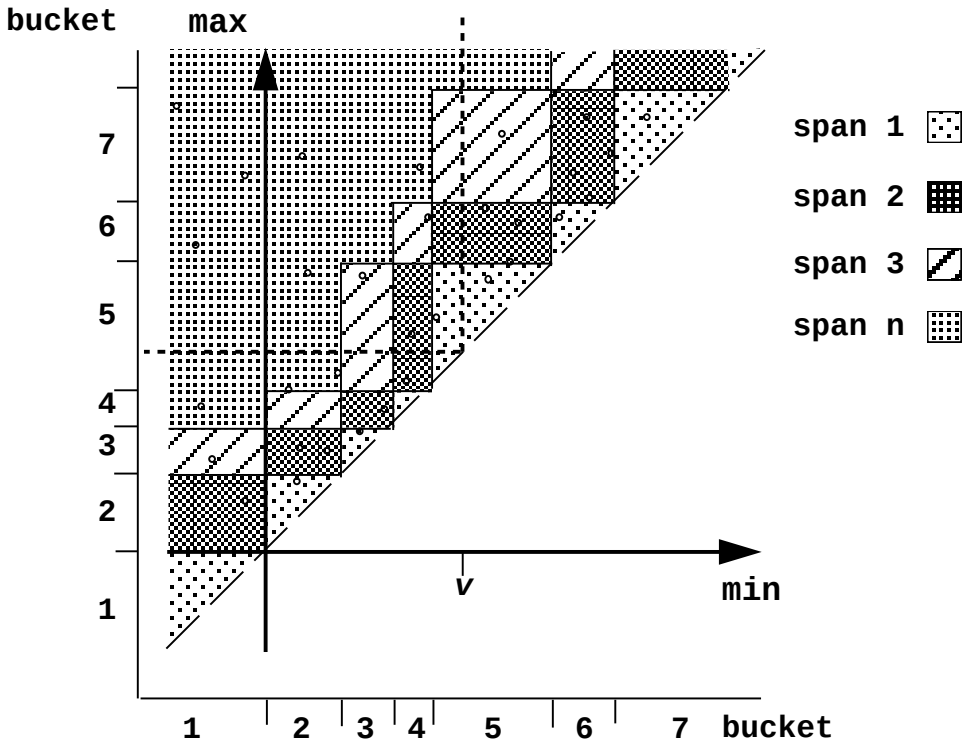


Figure 3.3. Span Filter. Based on the initial division of the value space into buckets, the cells are divided into several spans. There is only one span for all the cells which span more than three buckets (in this example. Each span is also divided into subareas based on the original division into buckets.)

3.2.1.2 The Active List

Figure 3.4 depicts Giles and Haimes' algorithm (see Section 2.1.7) over the span space. Though the algorithm does not explicitly partition the space in advance, the use of the global maximum cell span, Δ, does the same thing implicitly, as the width of the area that needs to be scanned is constant. When the change in the isovalue is greater than Δ, the algorithm must linearly scan *all* the cells in the range $(v - \Delta, v)$. All the cells in this range (the doted area in Figure 3.4) are put in the *active* list. The active list is then purged of all the cells in which the maximum value is below the given isovalue v. In our example all the cells in the doted area which are below the horizontal line at v will be purged. Since Δ depends on the data set, the algorithm has no control over the size of the scanned list. In two of our test cases, *Heart* and *Brain*, there are few cells on the boundary that have a very large span. This causes Δ to be so large that the algorithm must linearly scan approximately half of the data set. On the other hand Δ might be too

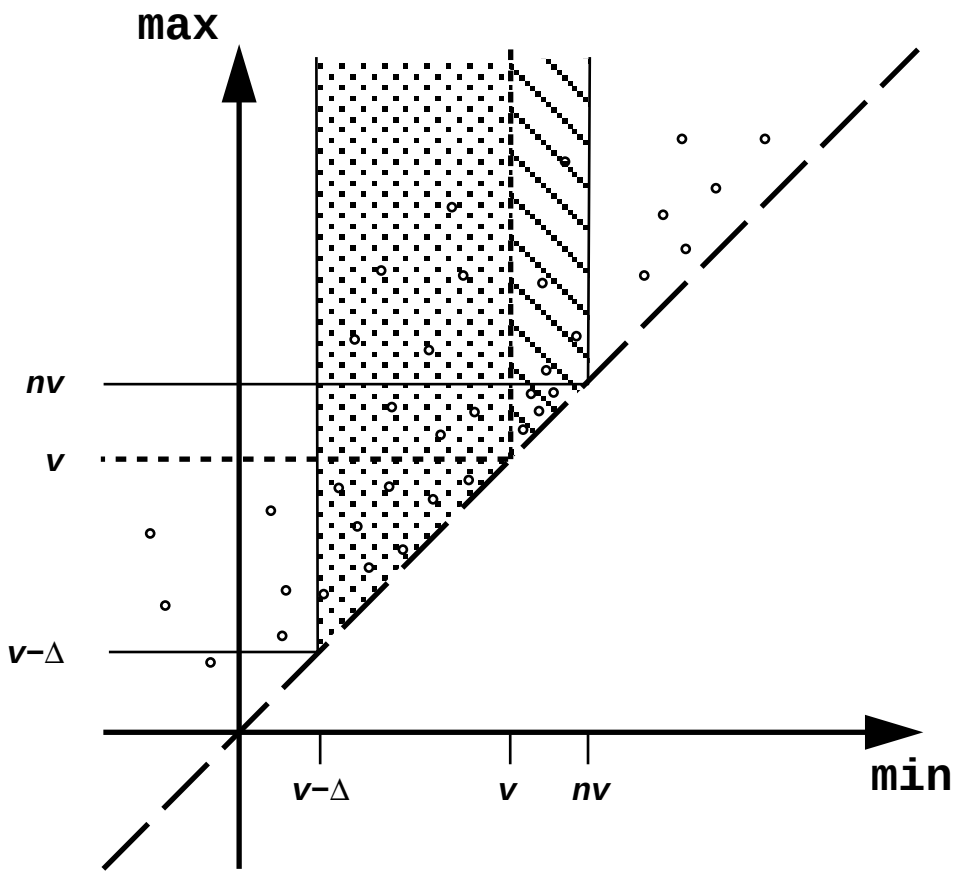


Figure 3.4. Active List. During initialization or when the change in the isovalue is larger than Δ, all the cells in the dotted area are put in the active list (for an isovalue of v). The cells which are below the horizontal line v will subsequently be removed. When the new isovalue nv is less than Δ from the current isovalue, only the cells in the striped area will be added to the active list. The active list will then be purged of all the cells that are in the dotted and striped areas and between the v and nv horizontal lines.

small such that the neighborhood search may not be used at all.

Using the span perspective, Figure 3.4, one can see that when the isovalue is changed from v to nv ($nv - v < \Delta$) the algorithm will scan all the cells in the striped band on the right and add them to the active list. The active list will then be purged of all the cells with a maximum value that is lower the the new isovalue nv. The cells that will be removed are those cells that lie between the v and nv horizontal lines. Note that most of the cells that will be discarded are those cells that are in the lower triangle in the striped area. This triangle is usually the most dense part of the striped band and therefore a large number of cells must be scanned and then discarded. If one scans across the entire

range of the data set, a typical change in the isovalue will be larger than 0.5%, wherwas, for a large data set, Δ will be much smaller, again not taking advantage of neighboring isosurfaces. Finally, the algorithm's complexity is still $O(n)$ in time.

3.2.1.3 Sweeping Simplices

The space decomposition for the sweeping simplices algorithm, as well as the marked cells for an isovalue pv, are shown in Figure 3.5. The solid dots are the marked cells. When a new isovalue is selected, all the cells that lie between the vertical lines pv and v are first marked. The cells that intersect the isosurface are those marked cells that lie above the horizontal line at v. Though sweeping simplices is faster than the active list algorithm and does not depend on a global Δ, its space decomposition is not optimal. Each of the groups whose range intersect the isovalue line must be linearly scanned and each such group contains an area outside the target isosurface region.

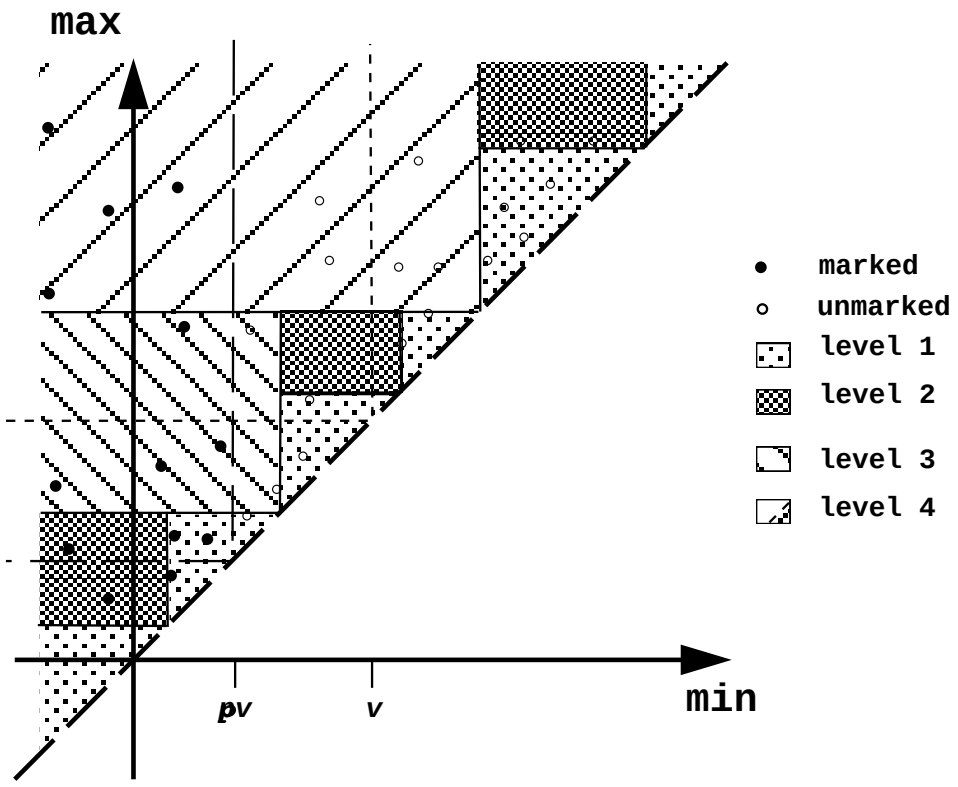


Figure 3.5. Sweeping Simplices.

3.2.2 New Extraction Methods Based On The Span Space

We now present new extraction methods that were developed based on the span space perspective. Chapter 4 presents an in depth description of the first such method that we published in conjunction with the span space. Hence, this method that achieved a near optimal worst case complexity is only briefly described here for completeness.

3.2.2.1 Near Optimal IsoSurface Extraction (NOISE)

The NOISE approach, Chapter 4, is based on the kd-tree [4] subdivision and is depicted in Figure 3.6. The points in the span space are recursively subdivided into two groups based on the median value. The division alternates between the span space two dimensions, i.e., the min and max values. Each division is done by selecting the point with the median value and designating it as the root of the current tree. The rest of the points are then grouped into two subtrees based on whether their value is smaller or greater than the current root value. Note that the size of the two subtrees can differ by at most one point as the division is based on the median value. The resulting kd-tree is thus a balanced tree, a feature that in Chapter 4 is used to eliminate the overhead of maintaining the kd-tree. Figure 3.6 depict a typical partitioning of the span space with a kd-tree. The first partitioning, along the min axis, is marked by root. Each of the resulting two sub-areas is split again, this time based on the max axis and is marked with 1.

The worst case complexity of the NOISE algorithm is only $O(\sqrt{n} + k)$. This result is based on Lee and Wong [16]. The kd-tree itself does not incur any overhead. Each node of the tree needs to contain only a pointer the appropriate data cell. To speed up the search over the kd-tree, each node can also contain the cell minimum and maximum values. The pros and cons of this additional information are examined in Chapter 4 as well.

3.2.2.2 Isosurfacing in Span Space with Utmost Efficiency (ISSUE)

The ISSUE algorithm [24] was designed mainly for massively parallel processors, though it can also be used as a serial algorithm. The underlying decomposition is based

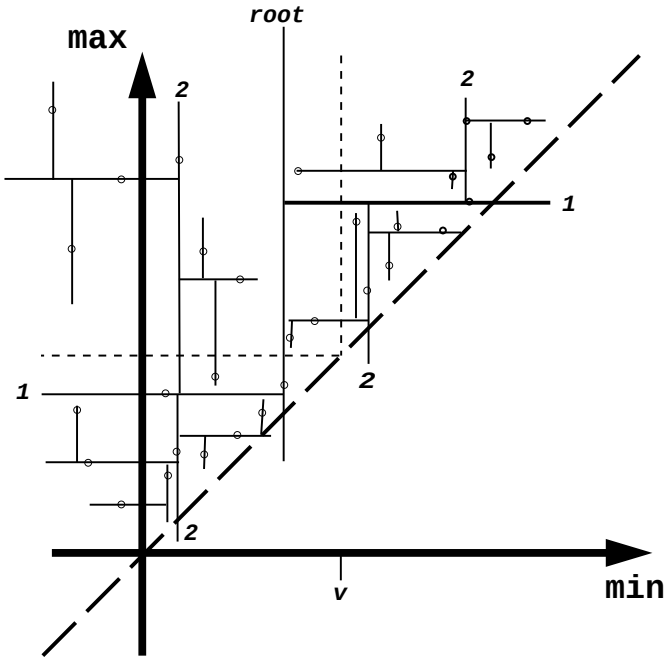


Figure 3.6. NOISE. The points in the span space are spilt recursively into two subregions, alternating between the min and max axis. Root marks the first partitioning follows by partitions 1, 2, and so on. Each partition designate one point (with the median value) and two equal subregions. Also depicted is a query based on a given isovalue v.

on a lattice subdivision of the span space into a two dimensional $L \times L$ lattice, Figure 3.7. The same subdivision is used for both the min and the max directions.

The search algorithm is based on the classification of the lattice cells into five cases based on their relative location to the isovalue. Without lost of generality, assume that for a given isovalue v, the point (v, v) is located in lattice cell (k, k). We note the indices of this lattice cell must be the equal to each other as the subdivision is the same in both direction. Each of the lattice cells (i, j) can now be classified as follows (see Figure 3.7):

1. $i > k$ or $j < k$: The isosurface does not pass through any of the cells in this region. Ignore all the cells in this region.

2. $i < k$ and $j > k$: The isosurface must pass though all of the cells in this region.

3. $i < k$ and $j = k$: All the cells in the region have a minimum value that is below the isovalue. However, the maximum criterion is still ambiguous. Only those cells with a maximum value higher then the isovalue intersect the isosurface.

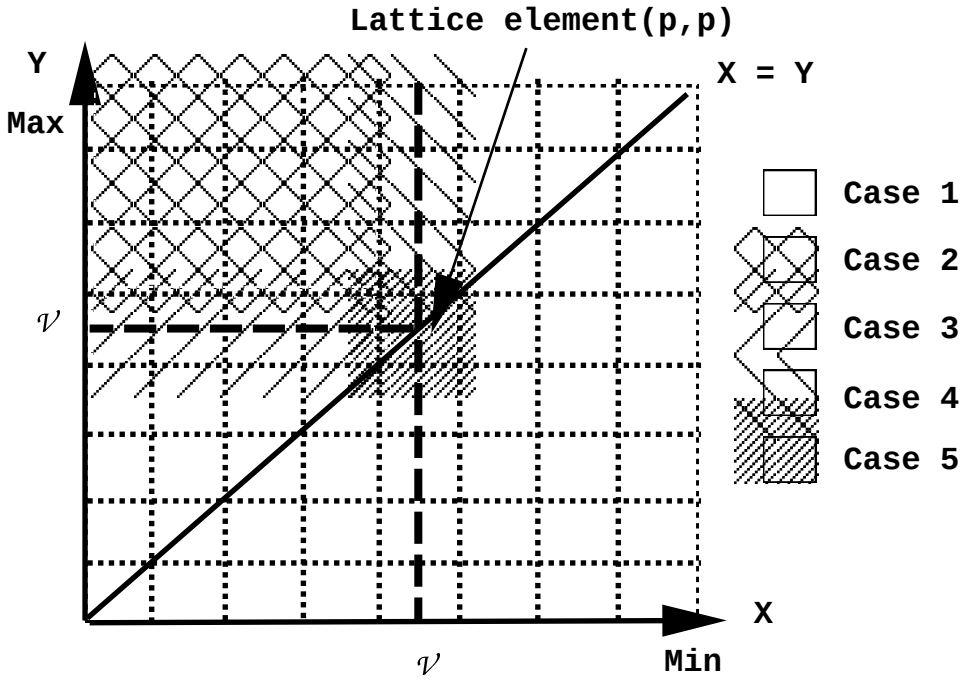


Figure 3.7. ISSUE: Lattice classification. The span space is divided using a lattice. For a given isovalue v, the lattice cells are classified into five cases. See text for more details on the meaning of each case.

4. $i = k$ and $j > k$: All the cells in the region have a maximum value which is above the isovalue. However, the minimum criterion is still ambiguous. Only those cells with a minimum value lower than the isovalue intersect the isosurface.

5. $i = k$ and $j = k$: This is the only region that requires a regular search as no a priori information is available for the cells in this region.

Out of these five classifications, cells in Case 1 region should be ignored. On the other hand, all of the cells in Case 2 regions intersect the isosurface and thus they require no search. For the cells in the particular region (k,k), Case 5, one needs to either check each of the cells individually or employ other search methods.

In the remaining Case 3 and 4, regions can be searched efficiently using a binary search over special data structures. First, note that all of the Case 3 regions are located in a single row while Case 4 regions are located on a single column. Second, the cells in each row (column) are sorted according to the cells maximum (minimum) value, since

the minimum (maximum) values have already been checked. This sorting can be done in an off-line pre-process stage after the lattice is defined. A single row or column can then be scanned using a binary search in an optimal time $O(\log l)$, where l is the number of cells in that row or column.

Analysis of the ISSUE algorithm leads, assuming a constant number of cells per region, to a complexity of $O(\log \frac{N}{L} + \frac{\sqrt{N}}{L} + K)$, where L^2 is the number of lattice regions. In general, the worst case complexity is $O(\sqrt{N} + K)$.

3.2.2.2.1 Parallel isosurface extraction. A parallel version of this algorithm for a Massively Parallel Processors (MPP), such as a Cray T3D, is based on using the serial ISSUE algorithm on each processor with only a subset of the data. A key requirement for effective parallel processing is optimizing the load balance, such that on average each processor performs the same amount of work.

Load balancing is achieved by careful distribution of the lattice regions to the different processors. Assuming there are $L \times L$ lattice regions and N processors, the $\frac{L \times (L+1)}{2}$ regions on or above the $min = max$ line are distributed in a round-robin fashion to the processors. Figure 3.8 shows an example with four processors and an 8x8 lattice regions. In terms of the lattice regions indices, region (i, j) is forwarded to processor $(j - 1 + \frac{(2L-1) \times (i-1)}{2}) \mod N$. This distribution leads to a fairly balanced work load as the regions in Cases 2 to 5 are evenly distributed.

The finer the resolution of the lattice region the better the load balance will be. However, this resolution is inversely proportional to the memory overhead. The ISSUE algorithm address this problem by employing two lattices, a fine lattice for the distribution of the cells and a coarse lattice for each processor. This method achieved a good cell distribution while maintaining low memory overhead overall.

3.2.2.3 Optimal Isosurface Extraction.

An optimal isosurface extraction method was developed in 1996 by Cignoni et al. [7] and is depicted in Figure 3.9. The novelty of this approach is the recursive division of the span space into three sections rather than the traditional binary division. Each such division is based on a single value and creates one semi-infinite rectangle and two triangular shaped areas. All the points in the middle rectangle area have their maximum

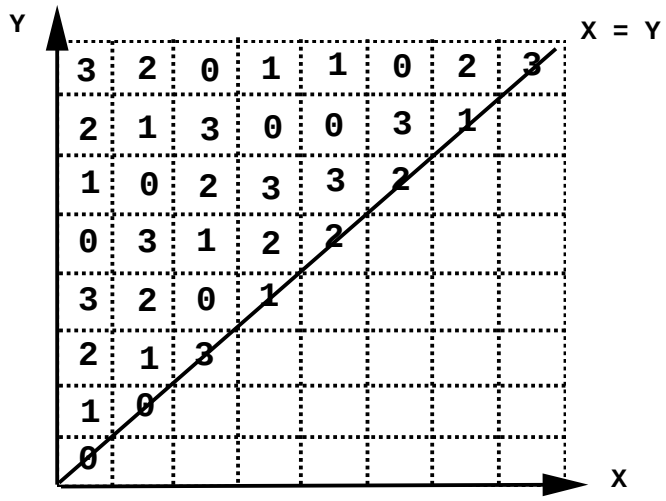


Figure 3.8. ISSUE: Lattice distribution. Numbers mark the processor that is assigned to each lattice cell.

value less than the splitting value and their minimum value larger than this value. To take advantage of this property, the points in the rectangle area are organized into two lists. One list is then sorted based on the points minimum value and the other list based on the points maximum value. The other two triangular shaped areas are recursively divided in the same fashion.The value at which each such division occurs should be selected such that the number of points in each section is roughly the same.

The search is performed by recursively descending this nested hierarchy. At each level, one of the triangular shared area is discarded, some of the points in the rectangular area are collected and the algorithm recursively check the triangular area.

The first triangular area can be discarded as the isovalue is either greater than all the points maximum value or bellow all the points minimum values. With respect to the rectangle area, the isovalue is either larger than the minimum values of all the points in the rectangle or smaller than their maximum values. For the points in the rectangle area, it is sufficient, therefore, to examine one of the sorted lists associated with this rectangle based on whether the isovalue is larger or smaller. Consulting the list requires one to only locate the first point (in $O(\log n)$) that satisfies the isovalue requirement. All the

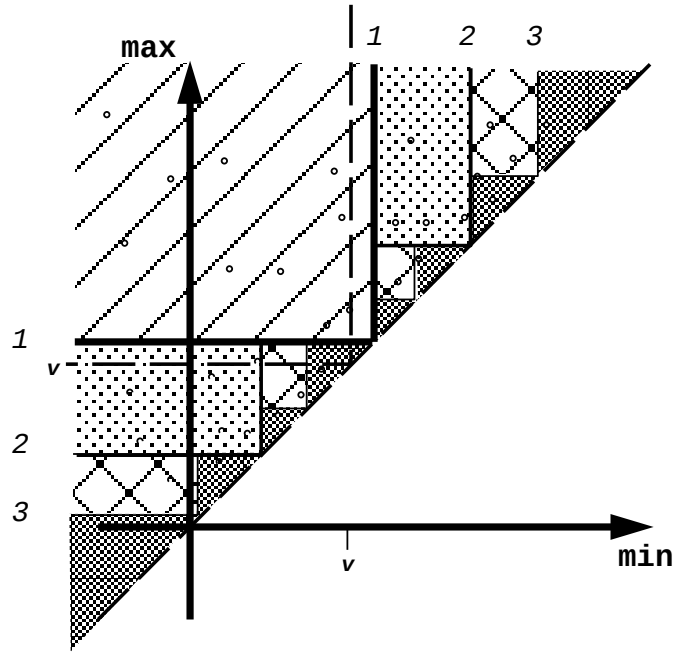


Figure 3.9. Optimal isosurface extraction. The span space is partitioned into three subareas, seen as a square and two triangles. Each triangular area is subsequently recursively partition into three subareas. The partition lines are marked with numbers. The points in the square regions are sorted in two lists, based on the min and max coordinates.

points in the list from that point until the end of the list must, by definition, satisfy this condition as well.

The memory requirement of this method is based on the need to have two sorted lists for each rectangle area. For each point in the span space there is an entry in one list sorted by the minimum value and a second entry in a list sorted by the maximum value. Each such entry is made of the point value (either minimum or maximum) and a pointer (or an id) to the actual cell. The total memory requirement is therefore on the order four times the size of the data.The time complexity of the optimal algorithm is $O(\log n + k)$.

CHAPTER 4

MAKING NOISE

A common obstacle for of all the interval methods is that the intervals are ordered according to *either* their maximum or their minimum value. Both the sweep algorithm and the active list method attempted to tackle this issue by maintaining two lists of the intervals, ordered by the maximum and minimum values. What has so far been missing, however, is a way to combine these two lists into a single list.

In the following, we present a solution to this obstacle. Using the span space as our underlying domain, we have employed a kd-tree as a means for simultaneously ordering the cells according to their maximum and minimum values.

4.1 Kd-Trees

Kd-trees were designed by Bentley in 1975 [4] as a data structure for efficient associative searching. In essence, kd-trees are a multidimensional version of binary search trees. Binary trees partition data according to only one dimension. Kd-trees, on the other hand, utilize multidimensional data and partition the data by alternating between each of the dimensions of the data at each level of the tree.

4.2 Kd-Tree Decomposition of the Span Space

Given a dataset we construct a kd-tree that contains pointers to the data cells and then use this kd-tree to rapidly search the dataset in order to answer isosurface queries. Figure 4.1 depicts a typical decomposition of a span space by a kd-tree. Each node in the tree holds one of the data values and has two subtrees as children. The subtrees are constructed such that all the nodes in one subtree, the *left* one for example, hold values that are less than the parent node's value, while the values in the *right* subtree are greater than the parent node's value. Starting at the root of the kd-tree, nodes at *even* depth level

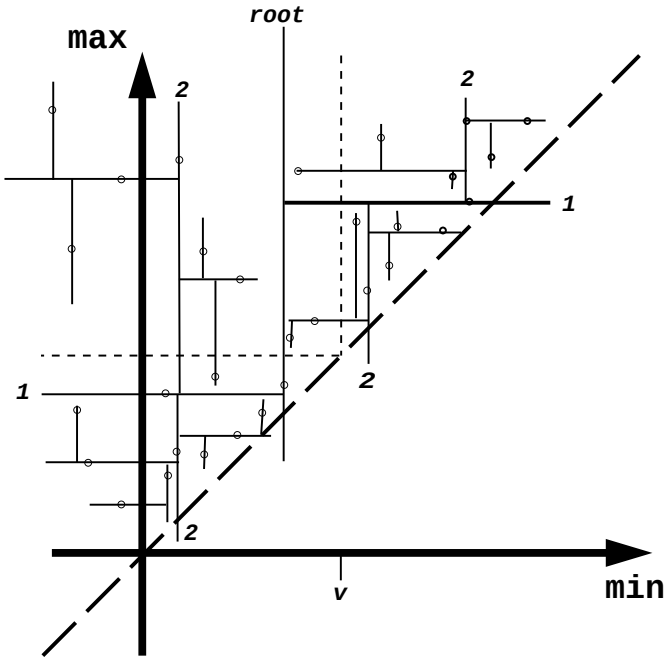


Figure 4.1. Kd tree. The points in the span space are spilt recursively into two subregions, alternating between the min and max axis. *Root* marks the first partitioning follows by partitions 1, 2 and so on. Each partition designate one point (with the median value) and two equal subregions. Also depicted is a query based on a given isovalue v.

are based on the minimum value of the dataset cell they refer to, while nodes at *odd* depth level refer to the cell maximum value.

4.2.1 Construction

The construction of the kd-trees can be done recursively in optimal time $O(n \log n)$. The approach is to find the median of the data values along the current dimension (either the *minimum* or *maximum* value of the corresponding cell) and store it at the current tree root node. The data are then partitioned according to the median and recursively stored in the two subtrees.

An efficient way to achieve $O(n \log n)$ time is to recursively find the median in $O(n)$, using the method described by Blum et al. [5], and partition the data within the same time bound.

A simpler approach is to sort the data into two lists according to the maximum and minimum coordinates, respectively, in order $O(n \log n)$. The first partition accesses the median of the first list, the *min* coordinate, in constant time, and marks all the data

points with values less than the median. These marks are then used to construct the two subgroups, in $O(n)$, and continue recursively.

Though the above methods have complexity of $O(n \log n)$, they do have weaknesses. Finding the median in optimal time of $O(n)$ is theoretically possible yet difficult to program [5]. The second algorithm requires sorting two lists and maintaining a total of four lists of pointers. Although it is still linear with respect to its memory requirement, it nevertheless poses a problem for very large datasets.

A simple (and elegant) solution is to use a Quicksort-based selection [22]. While this method has a worst case of $O(n^2)$, the average case is only $O(n)$. Furthermore, this selection algorithm requires no additional memory and operates directly on the tree. This algorithm performed *at least* four time faster on all of our application datasets in section 4.9.5 than the two sorted lists algorithm.

It is clear that the kd-tree has one node per cell, or span point, and thus the memory requirement of the kd-tree is $O(n)$.

4.2.2 Query

Given an iso-value, v, we seek to locate all the points in Figure 3.1 that are to the *left* of the vertical line at v and are *above* the horizontal line at v. We do not need to locate points that are *on* these horizontal or vertical lines if we assume nondegenerate cells, for which minimum or maximum values are not unique. This restriction will be removed later.

The kd-tree is traversed recursively when the isovalue is compared to the value stored at the current node alternating between the minimum and maximum values at each level. If the value at the node is to the smaller than the isovalue than only the left subtree should be traversed. Otherwise, *both* subtrees should be traversed recursively. For efficiency, two search routines are defined, *search-min-max* and *search-max-min*. The first name refers to the current dimension and the second name refer to the dimension the still needs to be searched. The importance of naming the second dimension will be evident in the next section, when we consider optimizing the algorithm.

Following is a short pseudo-code for the min-max routine.

```
SearchMinMax( isovalue, node )
{
   if ( node.min < isovalue ) {
      if ( node.max > isovalue )
         construct a polygon(s) from node
      SearchMaxMin( isovalue, node.right );
   }
   SearchMaxMin( isovalue, node.left );
}
```

Estimating the complexity of the query is not straightforward. Indeed, the analysis of the worst case was not developed until several years after Bentley introduced kd-trees (Lee and Wong [16]). Clearly, the query time is proportional to the number of nodes visited. Lee and Wong analyzed the worst case by constructing a situation where all the visited nodes are not part of the final result. Their analysis showed that the worst case time complexity is $O(\sqrt{n} + k)$. The average case analysis of a region query is still an open problem, though observations suggest it is much faster than $O(\sqrt{n} + k)$ [22, 3]. In almost all typical applications $k \sim n^{2/3} > \sqrt{n}$, which suggests a complexity of only $O(k)$. On the other hand, the complexity of the isosurface extraction problem is $\Omega(k)$, because it is bound from below by the size of the output. Hence, the proposed algorithm, NOISE, is optimal, $\theta(k)$, for almost all cases and is near optimal in the general case.

4.2.3 Degenerate Cells

A degenerate cell is defined as a cell having more then one vertex with a minimum or maximum value. When a given isovalue is equal to the extrema value of a cell, the isosurface will not intersect the cell. Rather, the isosurface will touch the cell at a vertex, an edge, or a face, based on how many vertices share that extrema value. In the first two cases, vertex or edge, the cell can be ignored. The last case is more problematic, as ignoring this case will lead to a hole in the isosurface. Furthermore, if the face is not ignored, it will be drawn twice.

One solution is to perturb the isovalue by a small amount, so that the isosurface will intersect the inside of only one of those cells. Another solution is to check *both* sides of the kd-tree when such a case occurs. While the direct cost of such an approach is not too high as this can happen at most twice, there is a higher cost in performing an equality test at each level.

4.3 Optimization

The algorithm presented in the previous section is not optimal with regards to either the memory requirement or search time. Several strategies to optimize the algorithm are presented next.

4.3.1 Pointerless Kd-Tree

A kd-tree node, as presented previously, must maintain links to its two subtrees. This introduces a high cost in terms of memory requirements. However, one can take advantage of the fact that in the NOISE case the kd-tree is completely balanced. At each level, one data point is stored at the node and the rest are equally divided between the two subtrees, which leads to the representation of a pointerless kd-tree as a one-dimensional array of the nodes. The root node is placed at the middle of the array, while the first $n/2$ nodes represent the left subtree and the last $(n-1)/2$ nodes the right subtree, as shown in Figure 4.2.

The memory requirements of a pointerless kd-tree, reduce to two real numbers per node, for minimum and maximum values, and one pointer back to the original cell for later usage. Considering that each cell for a three-dimensional application with tetrahedral cells has pointers to four vertices, the kd-tree memory overhead is even less than the size of the set of cells.

The use of a pointerless kd-tree enables one to compute the tree as an off-line pre-process and load the tree using a single read in time complexity of only $O(n)$. Data acquisition via CT/MRI scans or scientific simulations is generally very time consuming. The ability to build the kd-tree as a separate pre-process allows one to shift the cost of computing the tree to the data acquisition stage. Hence, reducing the impact of the initialization stage on the extraction of isosurfaces for large datasets.

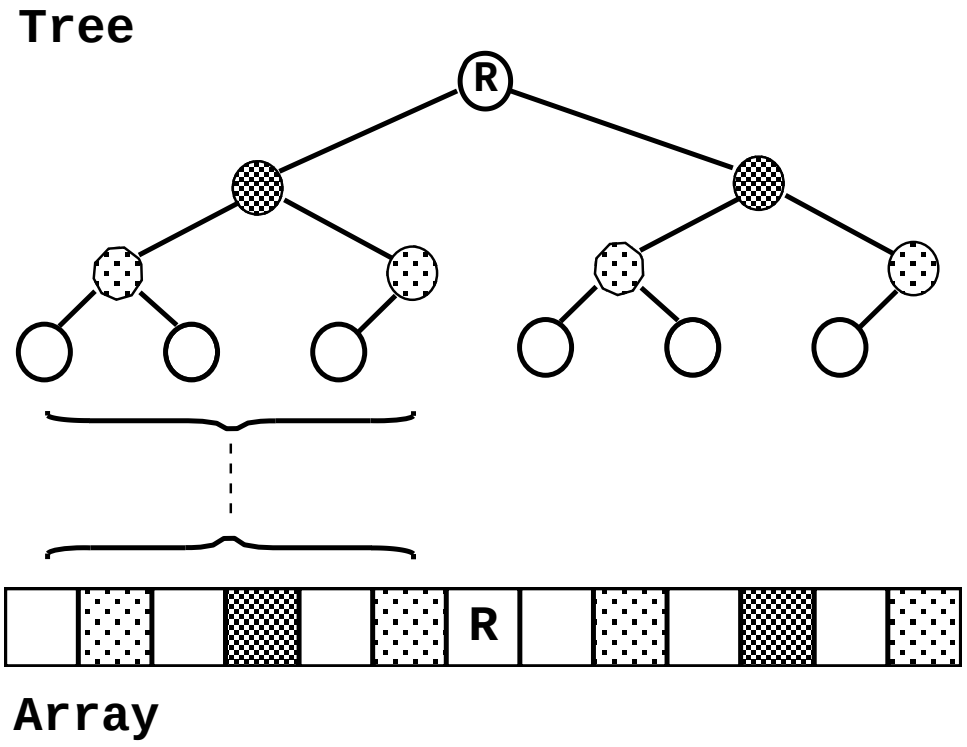


Figure 4.2. Pointerless Kd-tree. Top: A kd-tree with pointers. each level of the tree is marked with a different shade. Bottom: A pointerless tree is represented as an array.

4.3.2 Optimized Search

The search algorithm can be further enhanced. Let us consider, again, the min-max (max-min) routine. In the original algorithm, if the isovalue is less than the minimum value of the node, then we know we can trim the right subtree. Consider the case where the isovalue is greater than the node's minimum coordinate. In this case, we need to traverse *both* subtrees. We have no new information with respect to the search in the right subtree, but, for the search in the left subtree we *know* that the minimum condition is satisfied. We can take advantage of this fact by skipping over the odd levels from that point. To achieve this, we define two new routines, *search-min* and *search-max*, see Figure 4.3. Adhering to our previous notation, the name search-min states that we are only looking for a minimum value.

Examining the search-min routine, we note that the maximum requirement is already satisfied. We do not gain new information if the isovalue is less than the current node's minimum and again only trim off the right subtree. If the isovalue is greater than the

node's minimum, we recursively traverse the right subtree, but with regard to the left subtree, we now know that all of its points are in the query's domain. We therefore need only to *collect* them. Using the notion of pointerless kd-tree as proposed in Section 4.3.1, any subtree is represented as a *contiguous* block of the tree's nodes. Collecting all the nodes of a subtree requires only sequentially traversing this contiguous block.

A pseudo code of the optimized search for the odd levels of the tree, i.e., searching for minima is presented in Figure 4.3. The code for even levels, searching for maxima, is essentially the same and uses the same *collect* routine.

4.4 Count Mode

Extracting isosurfaces is an important goal, yet in a particular application, one may wish only to know how many cells intersect a particular isosurface. Knowing the number of cells that intersect the isosurface can help one give a rough estimate of the surface area of the isosurface on a structured grid and on a "well-behaved" unstructured grid. The volume encompassed by the isosurface can also be estimated if one knows the number of cells that lie inside the isosurface as well as the number of cells that intersect it.

The above algorithm can accommodate the need for such particular knowledge in a simple way. The number of cells intersecting the isosurface can be found by incrementing a counter rather than constructing polygons from a node and by replacing collection with a single increment of the counter with the size of the subtree, which is known without the need to traverse the tree. To count the number of cells that lie inside the isosurface, one need only look for the cells that have a maximum value below the isovalue.

The worst case complexity of the count mode is only $O(\sqrt{n})$. A complete analysis is presented in Appendix A.2. It is important to note that the count mode does not depend on the size of the isosurface. We shall show in Section 4.7 that such a count is extremely fast and introduces no meaningful cost in time. The count mode thus enables an application to quickly count the cells that intersect the isosurface and allocate and prepare the appropriate resources *before* a full search begins.

```
SearchMinMax( isovalue, node )
{
   if ( node.min < isovalue ) {
      if ( node.max > isovalue )
         construct polygon(s) from node;
      SearchMaxMin( isovalue, node.right );
      SearchMax( isovalue, node.left );
   } else
      SearchMaxMin( isovalue, node.left );
}

SearchMin( isovalue, node )
{
   if ( node.min < isovalue ) {
      construct polygon(s) from node;
      SearchSkipMin( isovalue, node.right );
      Collect( node.left );
   } else
      SearchSkipMin( isovalue, node.left );
}

SearchSkipMin( isovalue, skip_node )
{
   if ( skip_node.min < isovalue )
      construct polygon(s) from skip_node;
   SearchMin( isovalue, skip_node.right );
   SearchMin( isovalue, skip_node.left );
}

Collect( sub_tree )
{
   sequentially construct polygons for all nodes in this sub_tree
}
```

Figure 4.3. Optimized search.

4.5 Neighborhood Search

The Sweeping Simplices and the Active List algorithms were designed to take advantage of coherence between isosurfaces with close isovalues. We present a variant of the proposed algorithm that also takes advantage of such coherence.

As opposed to the previous methods that decompose the space specifically for small changes in the isovalue, the kd-tree decomposition can be used as is. This, in turn, means that at any time, either the regular or the neighborhood search can be performed over the same data structure and thus we can choose which one will likely be the best based on the current estimate. To find the new set of cells requires performing two searches. First the kd-tree is searched for cells that need to be removed. A second search is then performed to find new cells to add to the list. Figure 4.4 depicts a pseudo-code for a part of the second search.

The neighborhood search can benefit when the change in the isovalue is small and only a small number of cells needs to be added or removed, especially in the count mode. However, there are several disadvantages in using this type of search, as was the case in previous methods. First, an active cell list must be maintained that adds more overhead both in time and memory. Second, each node in the kd-tree must maintain yet another pointer to the cell entry in the active list so that it can be removed quickly without traversing the active list. Finally, if the number of cells that belong to *both* the current and the new cell list is small, the effort to find the new isosurface is doubled.

4.6 Triangulation of Tetrahedral cells

Once a cell is identified as intersecting the isosurface, we need to approximate the isosurface inside that cell. Toward this goal, the marching cubes algorithm checks each of the cell vertices and marks them as either *above* or *below* the isosurface. Using this information and a lookup table, the algorithm identifies the particular way the isosurface intersects the cell. The marching cubes and its many variants are designed for structured grids, though they can be applied to unstructured grids as well.

We propose a new algorithm for unstructured grids of tetrahedral cells. If an isosurface intersects *inside* a cell, then the vertex with the maximum value *must* be above the isosurface and the vertex with the minimum value *must* be below it.

```
NearSearchMinMax( pv, v, node )
{
   if ( node.min < pv )
      NearSearchMaxMin( pv, v, node.right );
   else
   if ( node.min > v )
      NearSearchMaxMin( pv, v, node.left );
   else
   {
      if ( node.max > v )
         add node;
      NearSearchMaxMin( pv, v, node.right );
      NearSearchMaxMin( pv, v, node.left );
   }
}
```

Figure 4.4. Neighborhood search - pseudo-code.

To take advantage of this fact, we reorder the vertices of a cell according to their ascending values, for example v1 to v4, a priori, in the initialization stage. When the cell is determined to intersect the isosurface, we need to compare only the isovalue against the two middle vertices *at most*. Since there are only three possible cases, only v1 is *below* the isosurface, only v4 is *above* the isosurface, or {v1,v2} are below and {v3, v4} are above see Fig. 4.5. Moreover, the order of the vertices of the approximating triangle(s), such that the triangle(s) will be oriented correctly with respect to the isosurface, is known in advance at no cost. We can further take advantage of the fact that there are only four

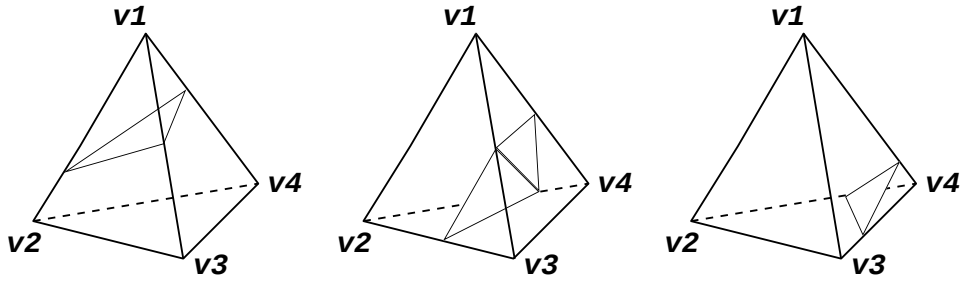


Figure 4.5. Triangulation for tetrahedra. The vertices are numbered according to ascending values.

possible triangles for each cell and compute their normals a priori. This option can improve the triangulation time dramatically yet it comes with a high memory cost.

4.7 Results

To evaluate the proposed algorithms, extensive tests were done on various datasets. The tests were carried on SGI (R4400, 150MHz) workstations with 256Mb and 640Mb of memory.

4.7.1 The Datasets

Several datasets from a variety of sources were used. Table 4.1 shows the characteristics of these models. The first three datasets consist of bioelectric field problems solved using the finite element method on unstructured tetrahedral grids, Figures 4.6, 4.7 and 4.8. *Head* is a 128^3 MRI scan of a human head, Figure 4.9. The FD, Fluid Dynamics, dataset is computed from a 256^3 spectral CFD simulation, Figure 4.10. Subsampled sets of this large dataset of sizes 32^3, 64^3 and 128^3, were also used.

4.7.2 Benchmarks

The algorithm was tested both with respect to CPU time and its complexity relative to a given dataset. Each test included 1000 random value isosurface extractions. Table 4.2 shows the distribution of the number of cells in the isosurfaces for the different models. The *Brain* model is an example of a nonuniform cell size and position distribution.

Table 4.1. Datasets

	Source	Type	Vertices	Cells
Heart	FEM	U-grid	11504	69892
Torso	FEM	U-grid	201142	1290072
Brain	FEM	U-grid	74217	471770
Head	MRI	S-grid	2M	2048383
FD-32	FEM	S-grid	32K	29791
FD-64	FEM	S-grid	256K	250047
FD-128	FEM	S-grid	2M	2048383
FD-256	FEM	S-grid	16M	16581375

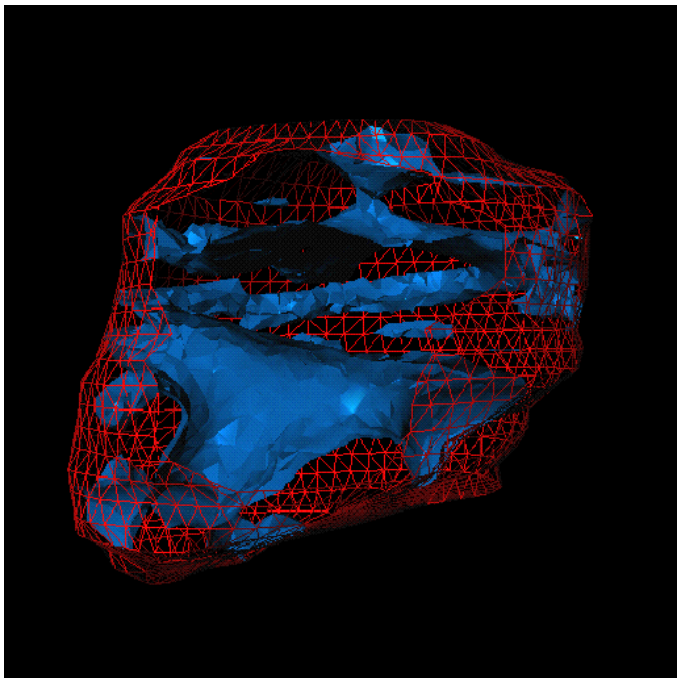


Figure 4.6. Heart: Iso-surfaces of constant voltage from a finite element simulation of cardiac defibrillation within the ventricles of the human heart.

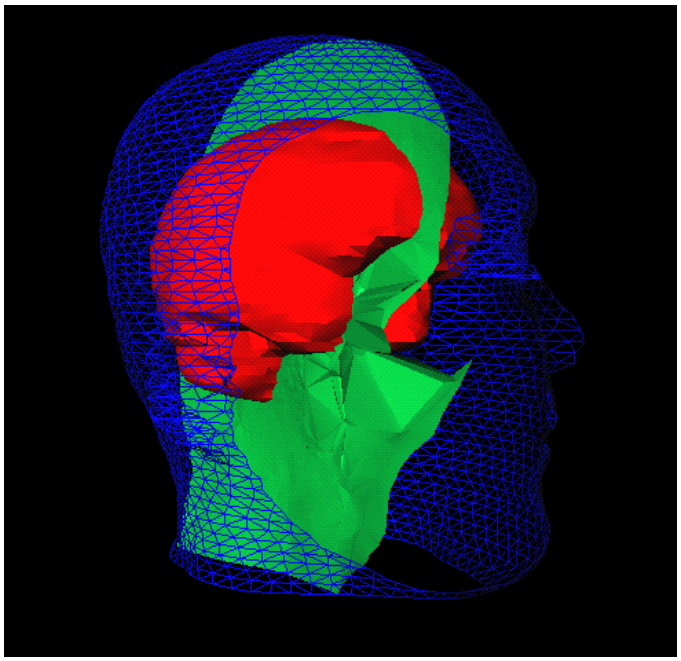


Figure 4.7. Brain: An iso-surface of constant voltage from a finite element simulation of temporal lobe epilepsy in a model of the human skull and brain.

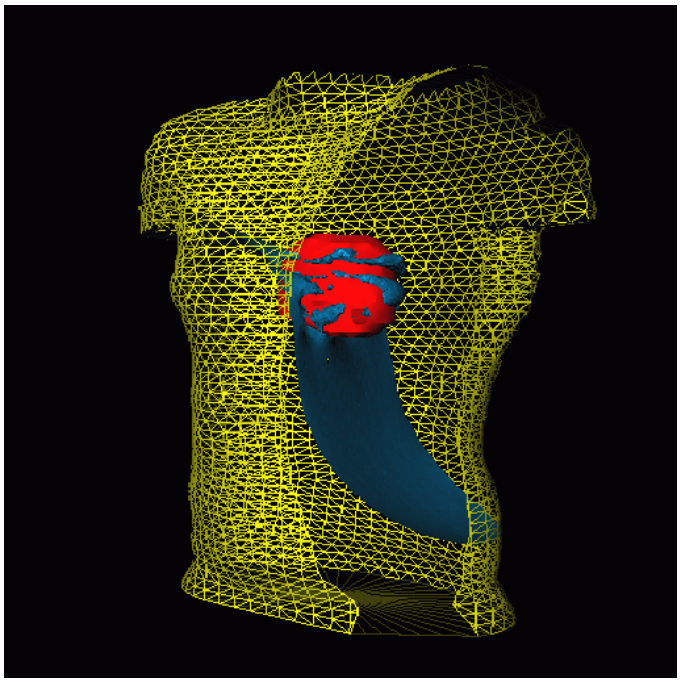


Figure 4.8. Torso: An iso-surface of constant voltage from a finite element simulation of the voltage distribution due to the electrical activity of the heart within a multichambered model of the human thorax.

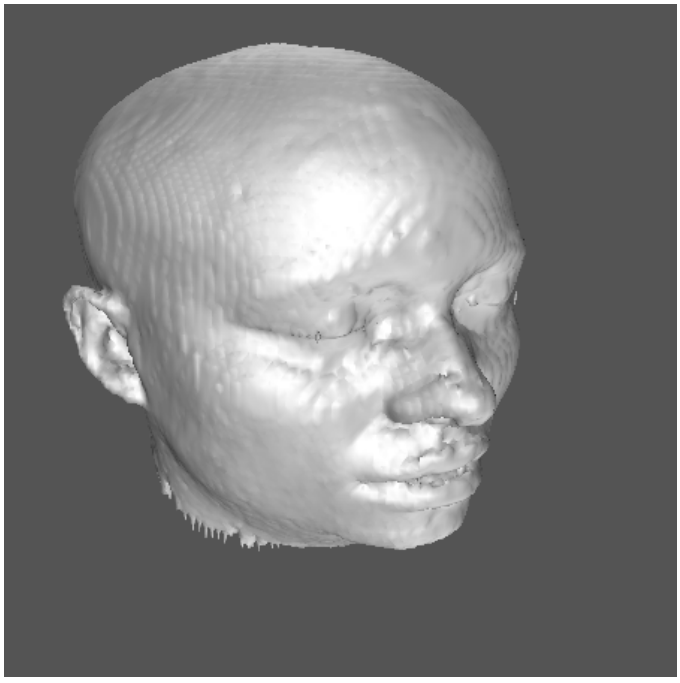


Figure 4.9. Head: Iso-surface from a 128^3 MRI scan.

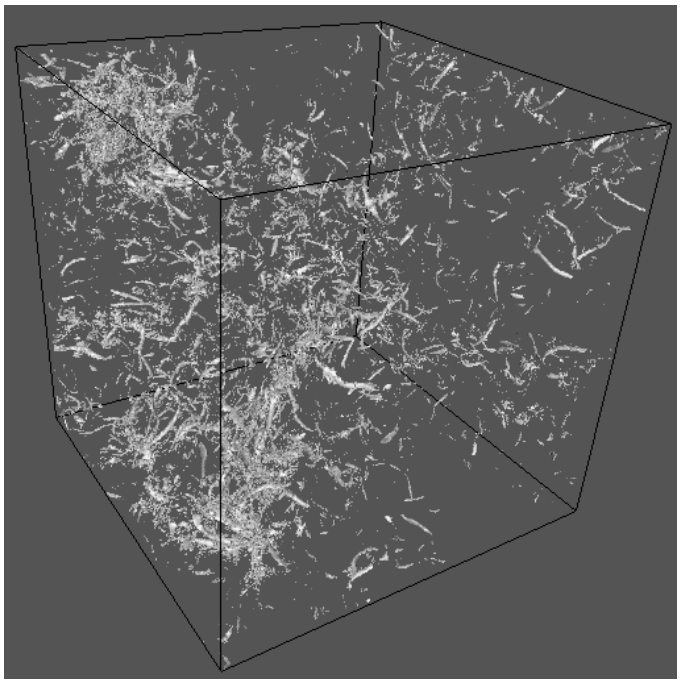


Figure 4.10. Turbulent flow in a fluid dynamic simulation representing the magnitude of fluid velocity and shows the onset of turbulence. The elongated structures are vortex tubes.

Some of the cells had very large span that would have caused worst case performance in previous isosurface extraction algorithms. Two tests were performed on this model, first using isovalues from the entire model domain (dataset *full*) and a second checking only a small dense area (dataset *partial*).

This work concentrates on finding the cells that intersect an isosurface and performing fast triangulation on tetrahedral cells. Therefore the triangulation times of the structured grid model were not measured. For these datasets, a call to an empty stub function was issued for each cell that intersects the iso-surface, therefore introducing some cost per intersected cell.

4.8 Analysis

Table 4.3 shows the *average* total number of tree nodes that were actually examined compared to the number of nodes that intersected the isosurface. Also presented are the *average* number of tree nodes that were examined but did not intersect the isosurface, i.e. the overhead of searching the tree, and the *maximum* number of cells that were collected

Table 4.2. Statistics I

	Cells in Isosurfaces		
	min	max	avg
Heart	0	23087	1617
Torso	32	31011	8001
Brain:			
partial	5287	26710	10713
full	12	14756	25
Head	8	610291	61091
FD-32	0	28541	3074
FD-64	0	230562	24720
FD-128	0	1680341	172247
FD-256	8	11172708	1088128

Table 4.3. Statistics II

	Found	Examined	Extra	Collected
Heart	1617	687	473	17472
Torso	8001	3487	2679	20156
Brain :				
partial	10713	2295	1570	14742
full	25	2043	2020	7370
Head	61091	4568	3735	512095
FD-32	3074	550	410	7447
FD-64	24720	1547	834	62511
FD-128	172247	4489	3405	512095
FD-256	1088128	12787	6940	4145343

at one time during the search. For tetrahedral cells the number of triangles per isosurface is about 33% higher than the number of cells that intersect the isosurface.

The algorithm consistently examined many fewer nodes than the size of the extracted isosurface. The only exception was the full Brain dataset where the average isosurface was more or less empty. Even in this pathological case, the number of cells that were examined was small, less than half a percent. This is a case where the algorithm is not optimal as $k < \sqrt{n}$, yet the overhead is negligible. Overall, the overhead of examining

extra nodes was kept at a minimum and the collection scheme achieved excellent results.

The complexity of the *search phase* was kept at $3\sqrt{n}$, which does not depend on the size of the resulting isosurface as predicted by the count mode analysis. CPU run times are shown in Table 4.4. The initialization step is measured in seconds while the *count* and *search* are in milliseconds. All numbers represent the average run time per query. The search includes triangulation for the unstructured grid datasets only, using the proposed fast triangulation algorithm, see Section 4.6. The time requirements for the count mode was kept to a few milliseconds, even for very large datasets with correspondingly large numbers of isosurfaces. The search optimization has clearly benefited from the collect routine, as is evident by the large collected blocks.

4.9 Application - Mantle Convection

4.9.1 Introduction to Mantle Convection Simulation

Solid state convection within the Earth's mantle determines one of the longest time scales of our planet. The Earth's mantle, the 2900 km thick silicate shell that extends from the iron core to the Earth's surface, though solid, is deforming slowly by viscous creep over long time periods. While gradual in human terms, the vigor of this subsolidus

Table 4.4. CPU Time

	Build	Count (msec)		Search† (msec)	
	(sec)		per cell		per cell
Heart	7.6	0.4	5.7×10^{-6}	7.0	4.3×10^{-5}
Torso	27.1	2.2	1.7×10^{-6}	43.8	5.5×10^{-5}
Brain :	7.6				
partial		1.5	3.2×10^{-6}	53.9	5.0×10^{-5}
full		1.0	2.1×10^{-6}	1.1	44.0×10^{-5}
Head	35.2	3.0	1.4×10^{-6}	31.4	1.5×10^{-5}
FD-32	0.2	0.3	10.4×10^{-6}	1.0	3.4×10^{-5}
FD-64	2.3	0.9	3.5×10^{-6}	7.7	3.1×10^{-5}
FD-128	22.6	2.9	1.4×10^{-6}	69.2	3.4×10^{-5}
FD-256	203.8	8.9	0.5×10^{-6}	420.4	2.5×10^{-5}

† Search times include triangulation for unstructured grids only.

convection is impressive, producing flow velocities of 1-10 cm/year. Plate tectonics, the piecewise continuous movement of the Earth's surface, is the prime manifestation of this internal deformation, but ultimately all large scale geological activity of our planet, such as mountain building and continental drift, must be explained dynamically by mass displacements within the mantle.

A major problem for researchers in computational mantle dynamics is to resolve the Earth's outer 100 km deep skin, or lithosphere. This lithosphere is an integral part of the mantle and thus a 100 km wide spatial resolution has to be achieved throughout the volume. The resulting computational problem is formidable and numerical discretizations with 1-10 million grid points have to be formulated to resolve the mantle volume on scales of 50 km or less. The resulting computational problem has been largely intractable on conventional sequential computers, as it is necessary to follow the time evolution of pressure, temperature and velocity over the entire volume, requiring gigabytes of memory and computational speeds of many gigaflops. However, such problems are well in reach of modern parallel computers, such as the massively parallel Cray Research, Inc. T3D system.

To address this problem, a three-dimensional spherical mantle dynamics code TERRA was used, which solves the Navier-Stokes equations in the infinite Prandtl number limit using a multigrid approach [1]. Discretization of the spherical shell is based on subdivision of the regular icosahedron producing a data structure that is well suited for modern parallel hardware using domain decomposition and message passing [2]. A message passing version of TERRA runs on the 256 processor CRAY T3D at the Advanced Computing Laboratory (ACL) at Los Alamos National Laboratory. On this machine the numerical modeling code shows excellent parallel performance, displaying a communication overhead of less than 10%. The computational memory afforded by the T3D has allowed investigation of convection employing a numerical grid of more than 20 million finite elements. Thus, it was possible to resolve a large range of dynamical length scales within the mantle.

4.9.2 Parallel Visualization Tools

Visualization of the vast simulation data is a serious challenge, as it is necessary to display the large-scale flow without compromising resolution of small scale structure. Small scale structure is generated primarily in thermal boundary layers, such as the lithosphere at the top of the mantle, but swept around by the large scale flow throughout the mantle. Moreover the temporal evolution must be visualized by displaying long-time series of data, requiring capacity many thousand times that of the individual time step.

To address this problem, visualization tools were developed that run on the massively parallel computer where the data were generated. This allows for both a rapid and high resolution display of simulation results too large for visualization on even high-end graphics workstations. In addition, running the visualization tools on parallel computers that generate the massive data avoids time consuming and cumbersome data transfers of the simulation results. The ability to display fine simulation details, such as the very localized generation and evolution of thermal structures along major boundary layers, greatly enhances the physical interpretation and presentation of these new high resolution convection experiments.

The parallel visualization tools consist of an isosurface extractor, a parallel polygon renderer, and a parallel slicer that can interpolate arbitrary planar slices through field data. These tools use a message passing and active message programming model [11]. The tools operate directly on the TERRA grid structure. While the TERRA grid is not a structured grid, the recursive subdivision basis of the grid allows the grid geometry to be implicitly represented rather than explicitly stored, saving memory and allowing for efficient geometric queries of the grid.

4.9.3 Parallel Rendering

Many researchers have studied parallel algorithms for polygon and volume rendering in recent years. Molnar et al. [20] provided a useful taxonomy for parallel rendering which classifies parallel rendering methods as sort-first, sort-middle, or sort-last according to where interprocessor communications occurs in the rendering pipeline.

The T3D parallel renderer uses a sort-middle based rendering algorithm. Both the data domain and the image are partitioned evenly among the processors. Each processor

first handles the geometric processing for the portion of the data it holds: isosurface extraction, arbitrary slicing and geometric transformation. The resulting geometric primitives are partitioned into scanline segments according to the portion of screen space they cover and sent to the processor responsible for that portion of the image using an active message communications model. Scanline segments are buffered and sent in groups to amortize the cost of a message over several scanline segments.

When the active message arrives at its destination processor, a handler function is invoked that completes the rasterization of the primitives it contains. Opaque scanline segments are directly z-buffered. Transparent scanline segments are buffered and handled after all processors complete geometric processing. The transparency segments are first depth sorted via a Newell-Newell-Sancha depth sort then composited front to back [10].

4.9.4 Parallel Isosurface Extraction

The TERRA datasets can be characterized as both structured and unstructured. The geometry of a cell can be inferred quickly from its index and thus it does not have to be saved explicitly for each cell. On the other hand, the overall structure of the dataset makes it difficult to utilize structured methods such as Wilhelms and Van Gelder's octree technique and algorithm.

The parallel version of NOISE takes advantage of the flexibility of the algorithm with respect to the structure of the dataset and the sort-middle parallel renderer discussed earlier. The data distribution is left to the renderer and NOISE is applied locally to the data on each processor. Isosurface extraction is then initiated by distributing only the new isovalue. The extracted local isosurface on each processor is temporarily added to the dataset to be rendered. It is then left to the parallel renderer to distribute the rendering of the isosurface to the other processors, a task which the renderer performs for the rest of its local geometric data.

The integration of the isosurface extraction and the parallel renderer allows us to take advantage of the renderer transparency capability. The use of transparency and the rapid isosurface extraction achieved by the algorithm also enable the rendering of several isosurfaces at the same time. Each such isosurface is extracted independently

from the others. The NOISE algorithm can determine, virtually at no cost, the amount of storage needed for the isosurface before it is computed. The advantage of this capability is refrained from allocating memory by reusing the memory allocated to the previous isosurface if it is large enough.

4.9.5 Results

Figures 4.11 and 4.12 illustrate two examples from the TERRA simulation. These figures render a static data set at a resolution of 1.25 million grid cells.

Figure 4.11 shows the temperature field over the spherical model of the Earth. In this simulation the viscosity (or stiffness) of the mantle fluid increases with depth by a factor of 30, as suggested by geophysical observation. This one parameter change results in a dramatic difference in convection structure displaying elongated downwellings from the upper surface instead of pointlike patterns typical for isoviscous flow [6]. Such elongated downwellings are analogous to Earth's linear subduction zones where plates dive under one another.

The outer surface of the sphere is a spherical radial shell cutting through the grid structure. Note that the sphere is hollow and an inner shell is shown as well. The

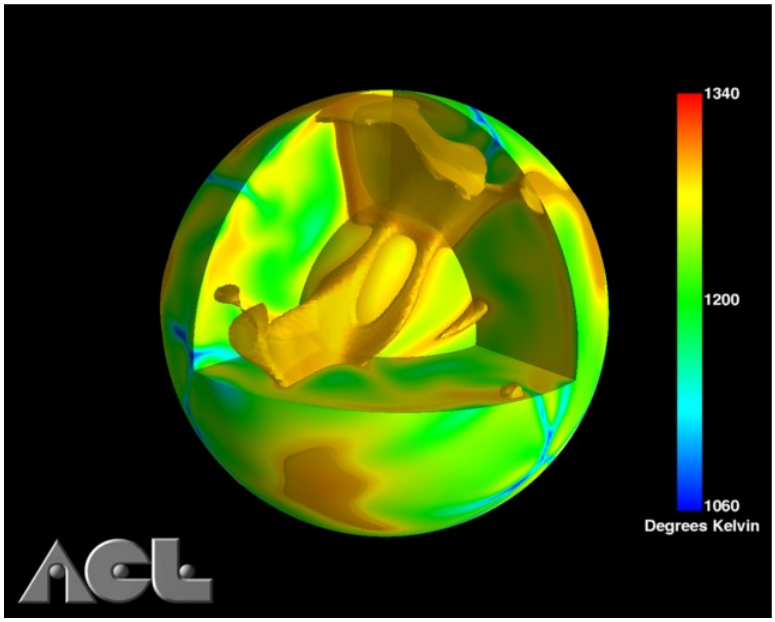


Figure 4.11. Slicing and isosurfacing of the temperature field.

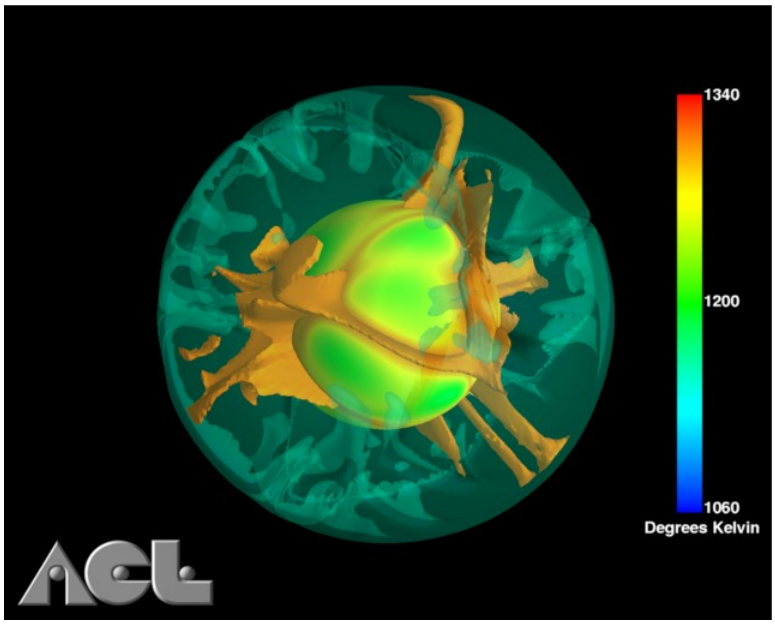


Figure 4.12. Two iso-temperature surfaces, with transparency.

simulation model covers only the outer 50% of the Earth's diameter where the mantle exists. A "wedge" has been removed by three slicing planes. Within the wedge opening an isosurface has been extracted with a "hot" temperature value. The isosurface and grid slices give insight on how hot material convects upwards through the Earth mantle.

Figure 4.12 again shows the temperature field with two isosurfaces over an inner spherical radial shell. The outer blue transparent isosurface uses a relatively cold temperature, indicating where cold material moves back toward the interior of the mantle. The inner orange opaque surface is a relatively high iso-temperature surface and again illustrates hot material moving outwards.

CHAPTER 5

ISOSURFACE EXTRACTION AND THE VISUALIZATION PIPELINE

Within the last ten years, isosurface extraction methods have advanced from an off-line, single-surface extraction process into an interactive, multisurface visualization tool. Interactivity is especially important in exploratory visualization where the user has no a prioriknowledge of any underlying structures in the data. A typical data exploration session therefore exhibits many isovalue changes in search of interesting features. In addition, global views are used to place an isosurface in the context of the entire dataset as well as detailed closeups of small sections of interest. Achieving interactivity in such a highly demanding environment is especially hard when very large datasets and isosurfaces with high depth complexity are involved.

Recently developed methods such as NOISE, Chapter 4, provide rapid localization of isosurfaces in very large datasets. The driving force behind these methods was the realization that more time was being spent locating an isosurface than the time needed for constructing it. These methods were so successful that they practically eliminated the localization problem altogether. Yet, they do not differentiate between small, simple isosurfaces and large, complex ones. In essence, these methods concentrate on their input and not on their output. Isosurface extraction is viewed as a separate and isolated problem from the complete visualization process. Even when it became apparent that isosurfaces could become too large to interactively handled by the graphics hardware, the typical solution [23] was to reduce the size of the isosurface based on a static criterion and not on a dynamic, performance-based, one.

This chapter examines isosurface extraction topics in the context of a complete visualization process and the major milestones in the last decade with respect to a pipeline

model of this process. The discussion leads to a new approach to isosurface extraction in large and complex datasets, namely view-dependent isosurface extraction. A view-dependent approach provides a means of culling sections of the isosurface that are occluded from the user's view point. Furthermore, sections of the dataset that are not visible to the user can be ignored during the search phase, eliminating the need to construct these geometries in the first place. We propose two implementations of the view-dependent paradigm in Chapter 6. These methods illustrate two different acceleration methods and the pros and cons of each.

5.1 Introduction

The complexity of isosurface extraction algorithms depends on the size of the dataset, n, and the size of the isosurface, k. Initially, isosurface extraction algorithms had a complexity of $O(n)$ and research effort was mainly directed at extracting more accurate isosurfaces. As interactivity became an important goal and the size of the datasets grew, attention shifted toward reducing the extraction's dependence on the size of the datasets. While faster algorithms were introduced, many still posed a complexity of $O(n)$. In recent years algorithms have improved to achieve a worst case complexity of $O(\sqrt{n}+k)$ and $O(\log n + k)$. As the dependency on the size of the original dataset was minimized, the size, k, of the extracted isosurface became the major factor. Rendering a very large isosurface presents a great challenge even on high-end graphics workstations. An even larger toll has to be payed when remote visualization is involved. To answer this challenge, current research efforts aim to simplify the geometry of the isosurface *after* the isosurface is extracted and *before* it is rendered or transmitted over a network. In effect, the quest is to reduce the complexity of rendering an isosurface to a sublinear complexity with respect to size of the isosurface. However, these methods do not address the effort spent to extract and construct the isosurface in the first place. Furthermore, these methods are slow and have a complexity at least linear with k.

5.2 The Visualization Pipeline

Figure 5.1 depicts the visualization process in what we term *The Visualization Pipeline* or the *V-Pipe*. The V-Pipe is more than a semantic nuance. It is intended to emphasize the

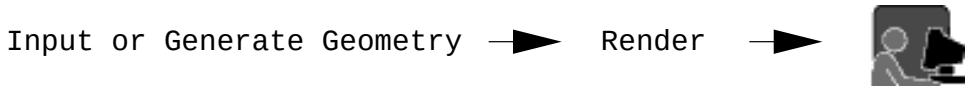


Figure 5.1. The visualization pipeline.

fact that many of the operations that constitute the visualization process can be performed in a pipeline fashion much like the well known and successful graphics hardware pipeline architecture.

In the context of isosurface extraction, the visualization pipeline is composed of two stages; namely extraction of the isosurface and rendering, Figure 5.2. Note that with the right architecture each of these stages can be done in a parallel, pipelined fashion. The best results will be achieved when the loads on each of these components are similar and no stalls occur in the pipeline.

5.2.1 Bottlenecks and Pipeline Stalls

We now examine the bottlenecks in the isosurface extraction process from the pipeline perspective. The first bottleneck identified in the past was the need to examine the entire dataset in the extraction process. The original marching cubes algorithm examines each and every cell in the dataset to determine if the isosurface intersects it. If an intersection is found a triangulation of isosurface inside the cell is generated. The dependency of the marching cubes algorithm on the size of the data can be reduced by the introduction of a filter between the data and the construction stage. The extraction stage is split into two separate stages: a search stage and a construction stage. Furthermore, the interaction between the data and the search phase that might have a higher complexity $O(n \log n)$, (as in NOISE), can be done as an off-line, preprocessing step.

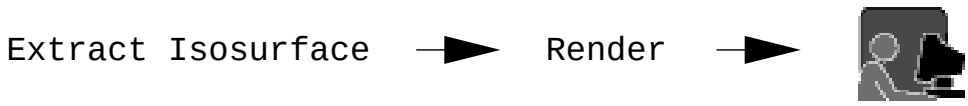


Figure 5.2. The visualization pipeline: Isosurface extraction.

The new pipeline configuration is depicted in Figure 5.3. Note that the construction stage is linear, $O(k)$, with respect to the size of the isosurface. The bottleneck has therefore shifted from the extraction stage to the search phase that depends linearly on the size of the dataset, n. Recent research effort, which optimizes the search stage have eliminated this bottleneck by reducing the complexity to $O(\sqrt{n}+k)$ [17] and even $O(\log n + k)$ [7].

Effectively, the dependency of the isosurface extraction on the size of the data set is all but gone, as the size of a typical isosurface, $O(n^{2/3})$ is much larger than then either $\sqrt{n}$ or $\log n$. Furthermore, the triangulation of a cell is a long process compared to the time it takes to identify that the cell intersects the isosurface. From the perspective of a pipeline paradigm, the search stage outpaces the construction stage even though its worst case complexity is higher. Since the V-Pipe is an asynchronous process, a buffer can be introduced between the search stage and the construction stage and thus the hazard of stalling the search stage is removed. Next, we assume that the triangulation of a cell is independent of other cells as is the case for the marching cube algorithm. In order to improve the performance under such an assumption, one needs to use multiple components of the second stage. In other words, this stage is performed in parallel, as shown in Figure 5.4. Note that only the construction stage needs to be parallelized and not the search stage (based on the performance of the NOISE algorithm presented in Chapter 4). This observation leads to the conclusion that effort should not be spent on parallelizing the search stage.

With the introduction of the parallel construction phase the bottleneck shifts toward the end of the pipeline and into the display phase as the shear size of the geometry of an

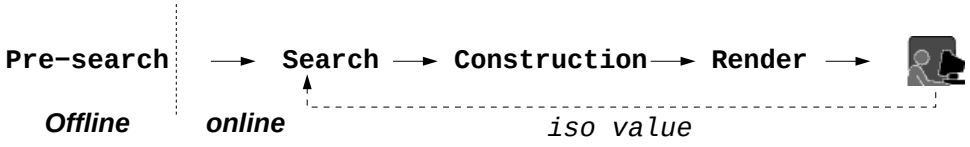


Figure 5.3. The visualization pipeline: The extraction stage is split into three stages. The presearch stage that analyzes the data in order to accelerate the search, can be performed once as an offline stage.

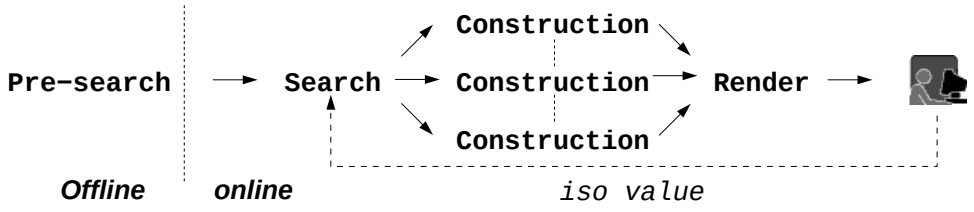


Figure 5.4. The visualization pipeline with a parallel triangulation phase.

isosurface can overwhelm even high-end graphics engines. The problem worsens when remote visualization is considered as the bandwidth over a local area netword (LAN) or the Internet are much smaller.

The current approach to this last bottleneck is to reduce the size of the geometry that needs to flow through these stages of the pipeline. Both lossless and lossy, as well as progressive compression methods, have been used to this end. To date, the best solution is still an open problem. Compression can be applied as either a post-process on the extracted *surface* geometry or as a preprocess on the original *volume* data, as seen in Figure 5.5.

Surface reduction has the advantage of being mainly a two and a half dimensional problem (a two-dimension problem embedded in a three-dimension world) as well as having the most refined surface as a baseline for estimating the approximation error. On the other hand, isosurfaces might be non-manifold and the extraction of the most refined isosurface maybe too costly and may not depend linearly on the size of the isosurface. The benefits of volume reduction are the reduced size of the dataset that needs to be processed as well as the ability to perform this stage off-line. Nevertheless,

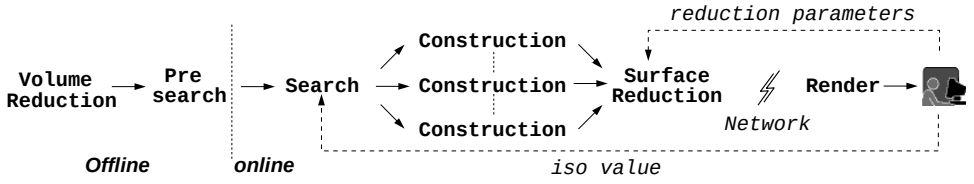


Figure 5.5. The visualization pipeline with reduction for remote visualization.

volume reduction is a much more difficult problem and it is performed a priori to the extraction phase. It is therefore much harder to estimate the introduced errors as well as the implication on the resulted topology and detail preservation.

5.3 A View Dependent Approach

Based on the V-Pipe perspective presented in the previous section, it is clear that the primary task is to reduce the amount of data that flow through the pipeline. Furthermore, this reduction will contribute the most when it is applied at an early stage. As such, the reduction can help accelerate all the other stages with little impact if any on their algorithm or implementation.

An approach that fits nicely into this paradigm is *view dependent visualization* and is depicted in Figure 5.6. The current view parameters can be introduced as early as the search stage and can drasticly reduce the data flow in scenes of high depth complexity, as seen in Figure 5.7. The potential saving of such an approach is shown in Figure 5.8. Note the large section of the isosurface, which represents the internal organs of the head, yet is not part of the view-dependent isosurface.

There are many benefits for such an approach throughout the V-Pipe stages:

Search - The search stage will not be required to access the entire dataset or even to access every cell that intersect the given isosurface. When a hierarchical search is used, the search tree can be pruned whenever the footprint of the current node (meta-cell) on the screen becomes smaller than a pixel.

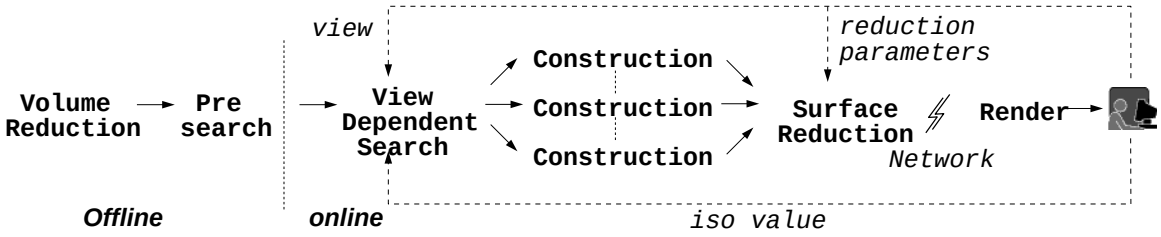


Figure 5.6. The visulaization pipeline: View dependent extraction. The current view parameters are sent back to the search stage, which in turn examines only the visible portion of the dataset.

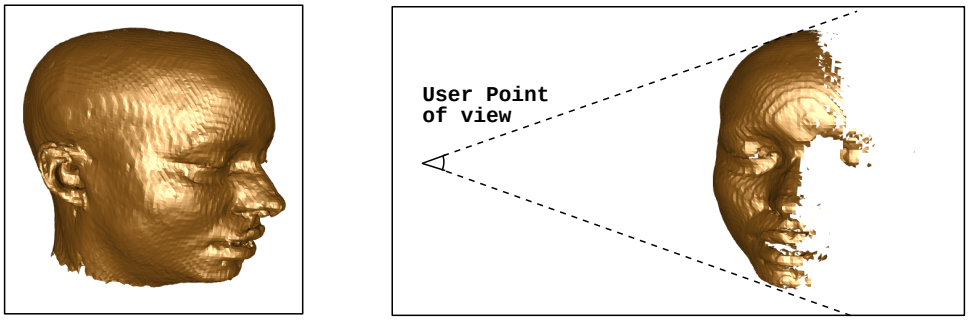


Figure 5.7. Left: The user view. Right: The same isosurface from a 90 degree angle to the user view, illustrating the incomplete reconstruction.

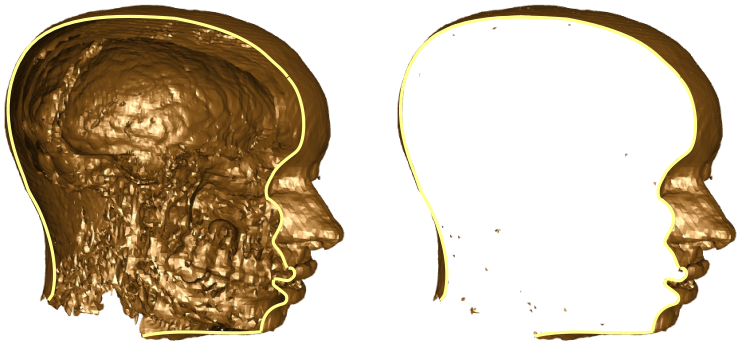


Figure 5.8. Extracted isosurface. A cut plane through the full and view-dependent isosurfaces extracted from the same view point as in Figure 5.7. Note the large internal structures that are part of the full isosurface but are not part of the view-dependent isosurface

Construction - Fewer triangles are constructed, leading to higher throughput.

Surface Reduction - Reduction algorithms are fairly slow and depend heavily on the size and complexity of the input. New algorithms can be developed that take into consideration the current orientation and footprint of the triangles with respect to the screen. Examples include such cases as a pair or triangles, which are far from the screen and thus appear very similar even though their orientation would prohibit a merge in the general case. Other benefits can be along a silhouette where the

direction perpendicular to the screen is much more important then the direction parallel to the screen.

Transmission - Better utilization of the available network bandwidth.

Rendering - A natural companion to image based rendering techniques.

These benefits are especially valuable in scenes with high depth complexity or small screen footprint. In practice, the price for these benefits is payed during the more complicated search phase. The major obstacle is to distinguish between visible and hidden sections in a short time. Another key issue is the characteristics of the dataset, mostly its depth complexity and its foot print on the screen. View dependent extraction will benefit from high-depth complexity and large triangles. When most of these conditions are not met, this approach might lead to poor performance. On the other hand, this approach does not prohibit the use of reduction later on in the pipeline, i.e., it is a complementary approach.

Figure 5.9 depicts a relative (not to scale) comparison between isosurface extraction methods with respect to the search, construction and display phases. The chart represent the theoretical time complexity and is split between methods that extract the entire isosurface (on the left) and view dependent methods that extract only portion of the isosurface (on the right).

The *construct* and *display* time requirements of the full extraction methods are all the same respectively as they all extract the entire isosurface. The differences are apparent in the time requirement of the *search* phase. The view dependent approaches, depicted on the right, show different characteristics. Since these methods extract only the visible portion of the isosurface their *construct* phase is much faster. Further more, a ray tracing method generates only images and not a geometry and thus its construct phase is negligible. In addition, ray tracing rendering is done in software, that removes the requirement for a powerful graphics hardware. On the other hand, ray tracing has high CPU requirement and thus is more suitable massively parallel processors (MPP) machines. In contract a view dependent method that generate geometry can take advantage of existing graphics hardware and has a lower CPU requirement. In addition, the extracted

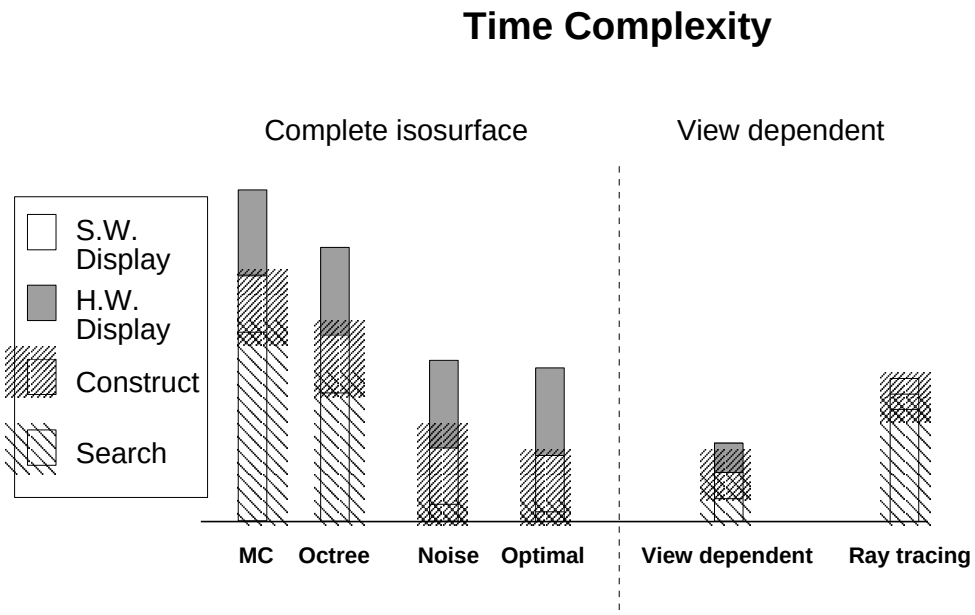


Figure 5.9. Complexity comparison.

geometry provides other information (such as normals) which can be useful for remote visualization.

5.4 Isovalue Change versus View Change

Isosurface extraction differs from other computer graphics problems in that the geometry is not known a priori. To make the matter worse, the geometry is dynamic as it is based on a changing isovalue. Some of the previous works on isosurface extraction tried to take advantage of spatial coherency under the assumption that isosurfaces similar isovalues should be spatially similar or at least intersect more or less the same data cells. While this assumption can help to accelerate the search phase it does not perform well in constructing the isosurface.

The introduction of view dependent isosurface extraction presents a new challenge. A visible isosurface will change either by changing the isovalue *or* by changing the view point. The latter is a known topic in the computer graphics community and much research has been done on it. The crucial difference between view change and isovalue change is that the former has more information, namely the geometry used for the previous view

point. Here, spatial coherence is much more apparent and is valid in the sense that even though only part of the geometry from the previous view may be visible, it is still correct.

CHAPTER 6

VIEW DEPENDENT ISOSURFACE EXTRACTION

Chapter 5 presented a novel, view dependent isosurface extraction approach, as illustrated in Figure 5.7. A view-dependent approach reduces the search, construction and display by visiting only the cells that contain the *visible* portion of the isosurface from a given view point.

This chapter proposes a paradigm for the view dependent approach and two implementations of it. The proposed paradigm is based on a hierarchical front-to-back traversal of the dataset with dynamic pruning of sections that are hidden from the view point by previously extracted sections of the isosurface. This work explores the middle ground between a mostly hardware based (e.g., marching cubes + Z-buffer) and a purely software (e.g., ray-tracing) algorithm for isosurface extraction. Our goal is to reduce the load on the network and/or graphics hardware by performing *some* of the visibility tests in software. The approach leads to an output sensitive method that can reduce the load of other components in the visualization pipeline such as transmission of the isosurface geometry over a network.

Section 6.1 presents the algorithms, drawing on recent innovations in the area of visibility algorithms. Sections 6.2.2–6.2 detail these algorithms as well as some proposed modifications. Results are presented in Section 6.2.4.

6.1 The Algorithm

The proposed method is based on the observation that isosurfaces extracted from very large datasets tend to exhibit high depth complexity for two reasons. First, since the datasets are very large, the projection of individual cells tend to be subpixel. This leads to a large number of polygons, possibly nonoverlapping, being projected onto individual

pixels. Secondly, for some datasets, large sections of an isosurface are internal and thus, are occluded by other section of the isosurface, as illustrated in Figure 5.8. These internal sections, common in medical datasets, can not be seen from any direction unless the external isosurface is peeled away or cut off. Therefore, if one can extract just the visible portions of the isosurface, the number of rendered polygons will be reduced resulting in a faster algorithm. Figure 6.1 depicts a two-dimensional scenario.

The proposed algorithm, which is based on a hierarchical traversal of the data and a marching cubes triangulation, exploits coherency in the object, value, and image spaces, as well as balancing the work between the hardware and the software. We employ a three step approach depicted in Figure 6.2. First, we augment Wilhelms and Van Gelder's algorithm by traversing down the octree in a front-to-back order in addition to pruning empty subtrees based on the min-max values stored at the octree nodes. The second step employs coarse software visibility tests for each [meta-] cell which intersects the isosurface. The aim of these tests is to determine whether the [meta-] cell is hidden from the user by previously extracted sections of the isosurface (thus the requirement for a front-to-back traversal). Finally, the triangulation of the visible cells are forwarded to the graphics accelerator for rendering by the hardware. It is at this stage that the final and exact [partial-] visibility of the triangles is resolved. A data-flow diagram is depicted in

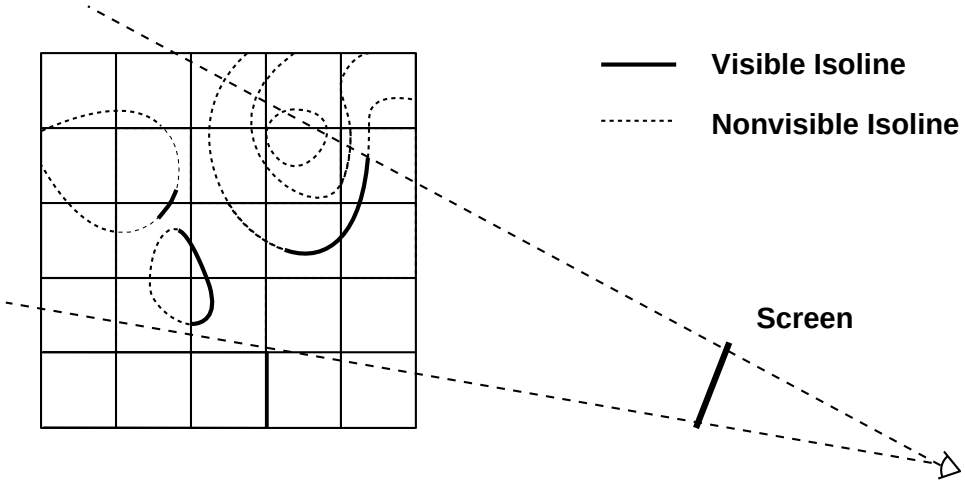


Figure 6.1. A two-dimensional scenario of isosurface extraction.

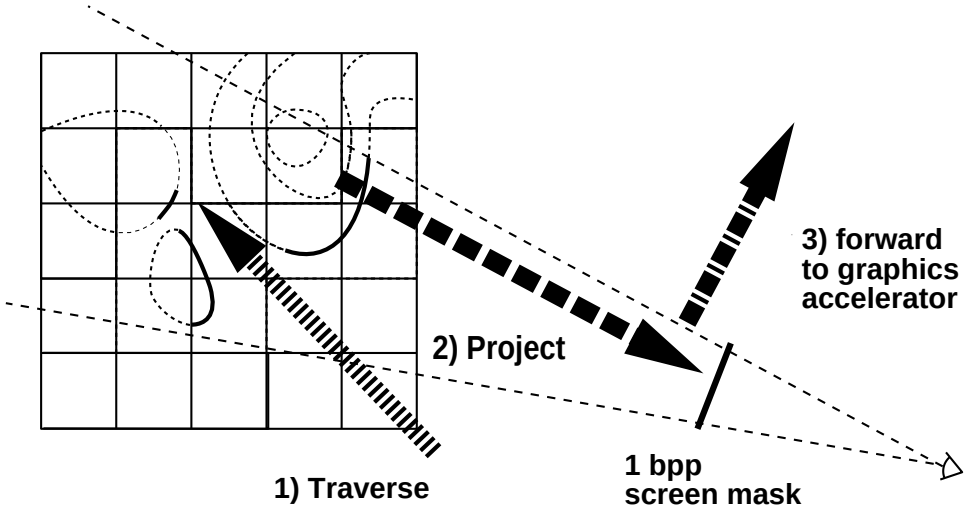


Figure 6.2. The three-step algorithm.

Figure 6.3. The algorithm uses both object space and value space acceleration methods as previous algorithms. The visibility test, however, is done both in software and in hardware as opposed to the traditional methods of extracting all the geometry and only on the special graphics hardware to resolve the visibility. The key idea, as shown in the data-flow diagram, is to perform preliminary visibility tests in software and thus avoid the generation of large sections of the isosurface which will not be visible by the user.

6.1.1 Visibility

Quickly determining whether a meta-cell is hidden and thus can be pruned is fundamental to this algorithm. This is implemented by creating a virtual screen with one bit per pixel. We then project the triangles, as they are extracted, on to this screen and set those bits which are covered, providing an occlusion mask (see Figure 6.4).

Additional pruning of the octree nodes is accomplished by projecting the meta-cell on to the virtual screen and checking if any part of it is visible, i.e., if any of the pixels it covers are not set. If the entire projection of the meta-cell is not visible, none of its children can be visible.

Note that it is important to quickly and efficiently classify a cell as visible. A hidden cell and all of its children will not be traversed further and thus can justify the time

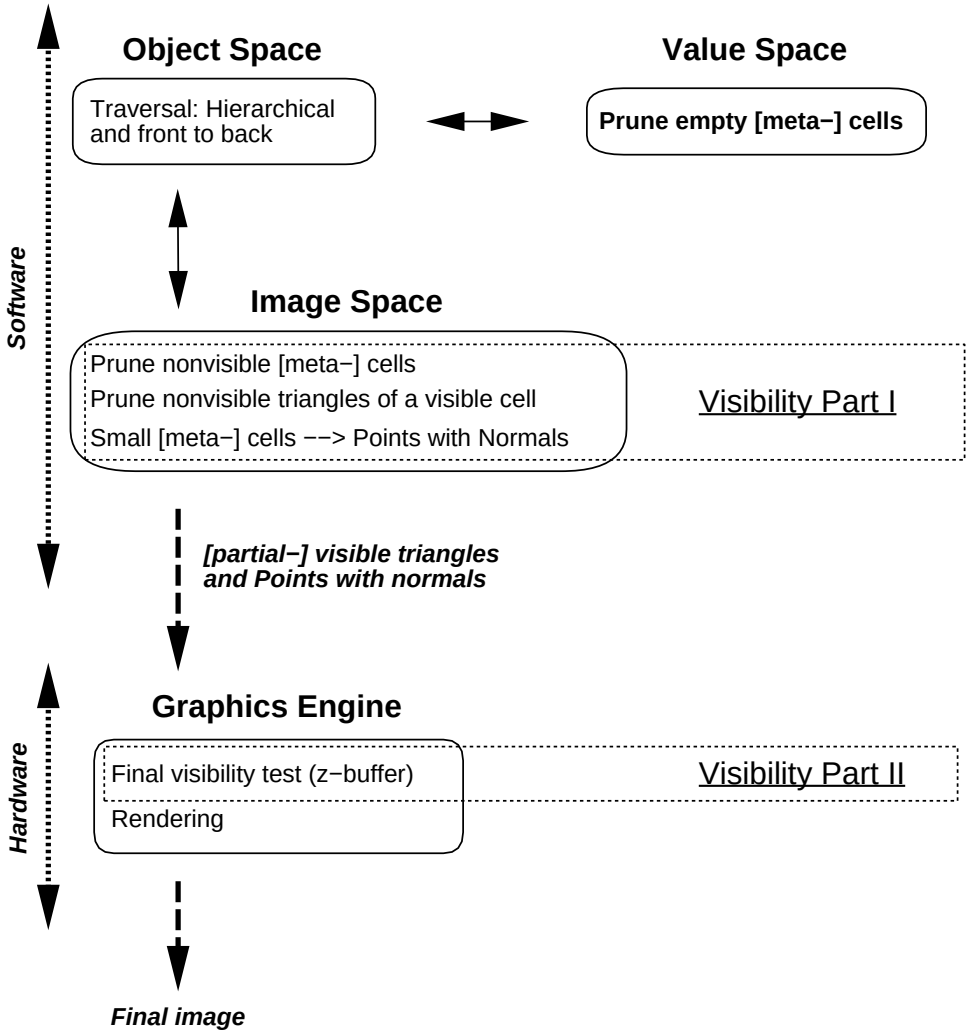


Figure 6.3. The algorithm data flow. In addition to object and value space pruning methods, the algorithm also performs some of the visibility tests in software and thus reduce the size of the the geometry that need to be extracted as well as rendered.

and effort invested in the classification. A visible cell, on the other hand, does not gain any benefit from this test and the cost of the visibility test is added to the total cost of extracting the isosurface. As such, the cell visibility test should not depend heavily on the projected screen area otherwise the cost would prohibit the use of the test for meta-cells at high levels of the octree - exactly those meta-cells that can potentially save the most.

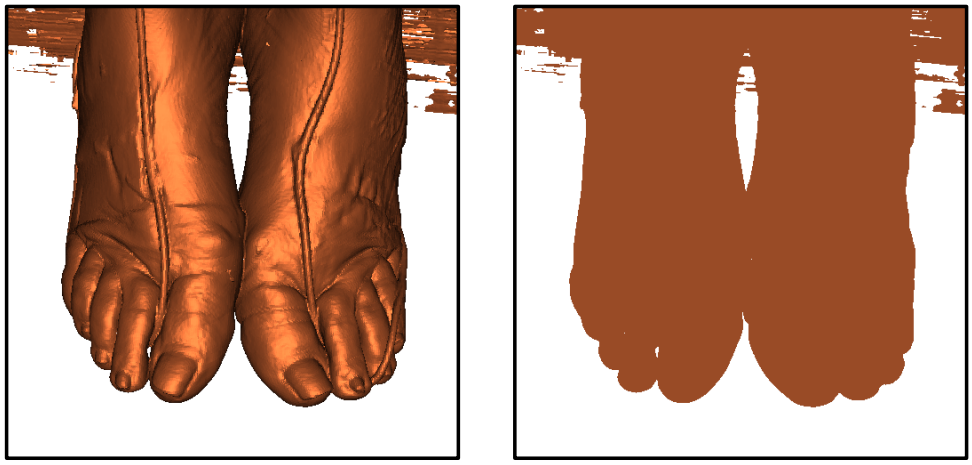


Figure 6.4. An image and its occlision mask.

6.2 A WISE Method

6.2.1 Overview

Based on an analysis of the visualization pipeline we proposed a view dependent iso-surface extraction approach in Chapter 5. Section 6.1 has depicted a three-step paradigm for this approach, which requires rapid visibility tests, part of which are performed in software.

Two components influence the visibility cost, namely the cost of projecting a point, triangle, or a meta-cell on to the screen and the cost of either scan-converting triangles or determining if a meta-cell projected area contains any unset pixels.

These costs are addressed in sections 6.2.2 and 6.2.3, and two approaches are used. First, hierarchical tiling is employed for the virtual screen. Second, the projection cost is reduced using a variation of shear-warp factorization.

6.2.2 Image Space Culling

Hierarchical tiles [11] are employed as a means for fast classification of meta-cells and determining the coverage of extracted triangles. The hierarchical nature of the algorithm ensures that the cost of either of these two operations will not depend highly on their projected area.

6.2.2.1 Hierarchical Tiles

Hierarchical tiles provide a mechanism for accelerating software based rendering. The notion of tiles is based on the projection of one polygon at a time in a front-to-back order and rendering only those pixels that contribute to the final image. The front to back order ensures that each pixel is rendered only once. The representation of these basic tiles in a hierarchical structure provides a mechanism reducing the number of tiles against which a polygon needs to be compared, leading to an even faster execution time.

A coverage map (a tile) is a rectangle bitmap (such as 8x8) in which each bit represents a pixel in the final image. The algorithm is based on the premise that all the possible coverage of a single edge crossing a tile can be precomputed and tabulated based on the points where the edge intersects the tile boarder as seen in Figure 6.5. The coverage pattern of a convex polygon for a particular tile of the image is computed by combining

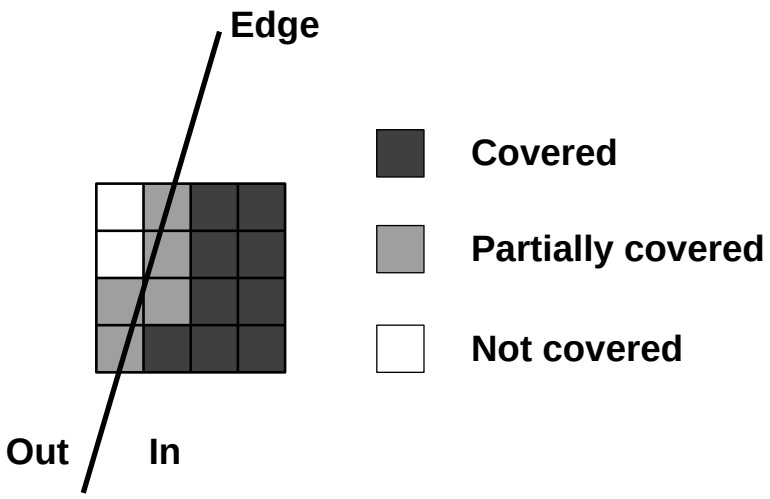


Figure 6.5. An edge tile.

the coverage maps of the polygon edges. The coverage map of a triangle can thus be computed from three precomputed tiles with no dependency on the number of pixels the triangle actually covers,[1] see Figure 6.6.

Rendering a polygon amounts to computing the coverage map of the polygon for each tile in the image and isolating the tile pixels that currently are not covered but are in the polygon coverage map, as indicated in Figure 6.7. In order to accelerate the rendering, the tiles are organized in a hierarchical structure in which each meta-tile represents a block of [meta-] tiles. Under this structure, a polygon is projected onto the top meta-tile and only those subtiles in which the polygon might be visible are checked recursively, leading to a logarithmic search.

6.2.2.2 Hierarchical Visibility Mask

The following implementation differs from the one proposed by Greene in that it does not actually render the visible portion of a projected triangle. Rather, we mark the triangle as visible and forward it to the graphics hardware. It is then left to the graphics accelerator to determine which pieces of the triangle are actually visible and correctly

[1] We refer the reader to the work by Greene [11] for a detailed explanation on how the three states (Covered, Partially-covered and Not-covered) can be represented by two tile masks and the rules for combining coverage maps.

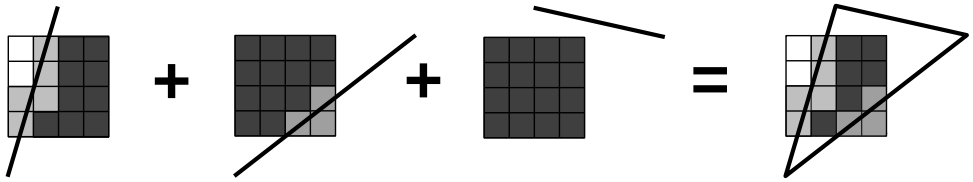


Figure 6.6. A triangle tile coverage map.

render them. Figure 6.8 shows two examples of isosurfaces and their hierarchical visibility masks. These examples illustrate the benefit of visibility masks when the object fills most of the user view. Note, however, that even in the case of the isosurface for the bone, some of the top level pixel are used.

One should note that it is not possible to determine a priorithe front-to-back relations between the triangles inside a single cell. It is therefore mandatory to accept all or none of the triangles, even though they need to be projected on the hierarchical tiles one triangle at a time. Figure 6.9 shows the classification of the cells as well as the portions of the isolines which are extracted. Note that the entire isoline section in a visible cell (shown in light gray) is extracted. The nonvisible portions will be later removed by the graphics accelerator.

An additional feature we employ limits recursion down the octree once the size of a meta-cell is approximately the size of a single pixel. Instead, we forward a single point and a normal to the graphics hardware, similar to the dividing cubes method [8]. The normal is estimated by the gradient of the field. The advantage of this method is that the single point potentially represents a large number of polygons since the meta-cell that projects to a pixel may still be high in the octree.

6.2.3 Warped IsoSurface Extraction (WISE)

A key component in the visibility test is the projection of a point, a triangle or a meta-cell onto the screen. In general, the perspective projection of a point is a 4x4 transformation followed by two divide operations, for a total of 16 multiplications, 12 additions and 2 divisions per vertex. Clearly, the cost of performing such transformation for each and every vertex of the projected meta-cells and triangles is too high. In addition,

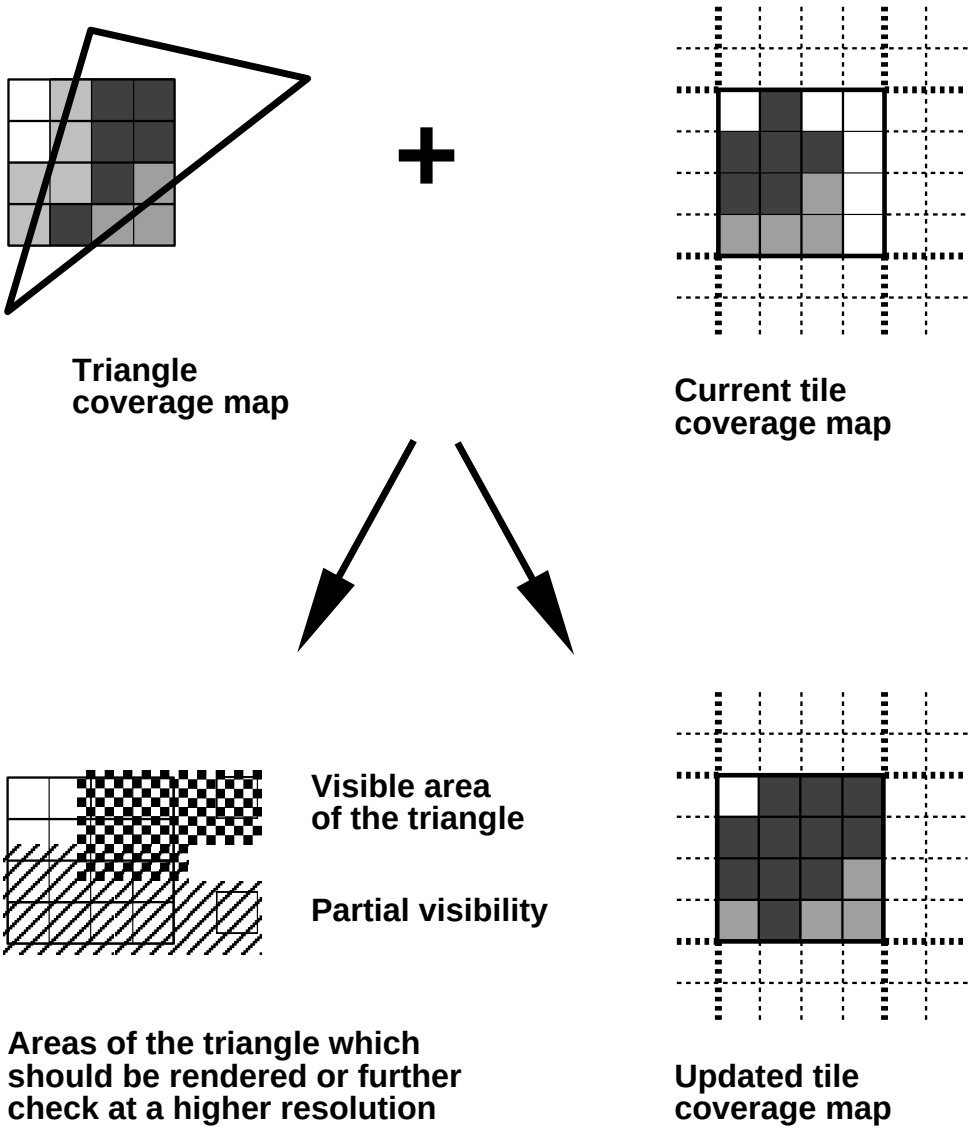


Figure 6.7. Application of a triangle coverage map and an image tile coverage map.

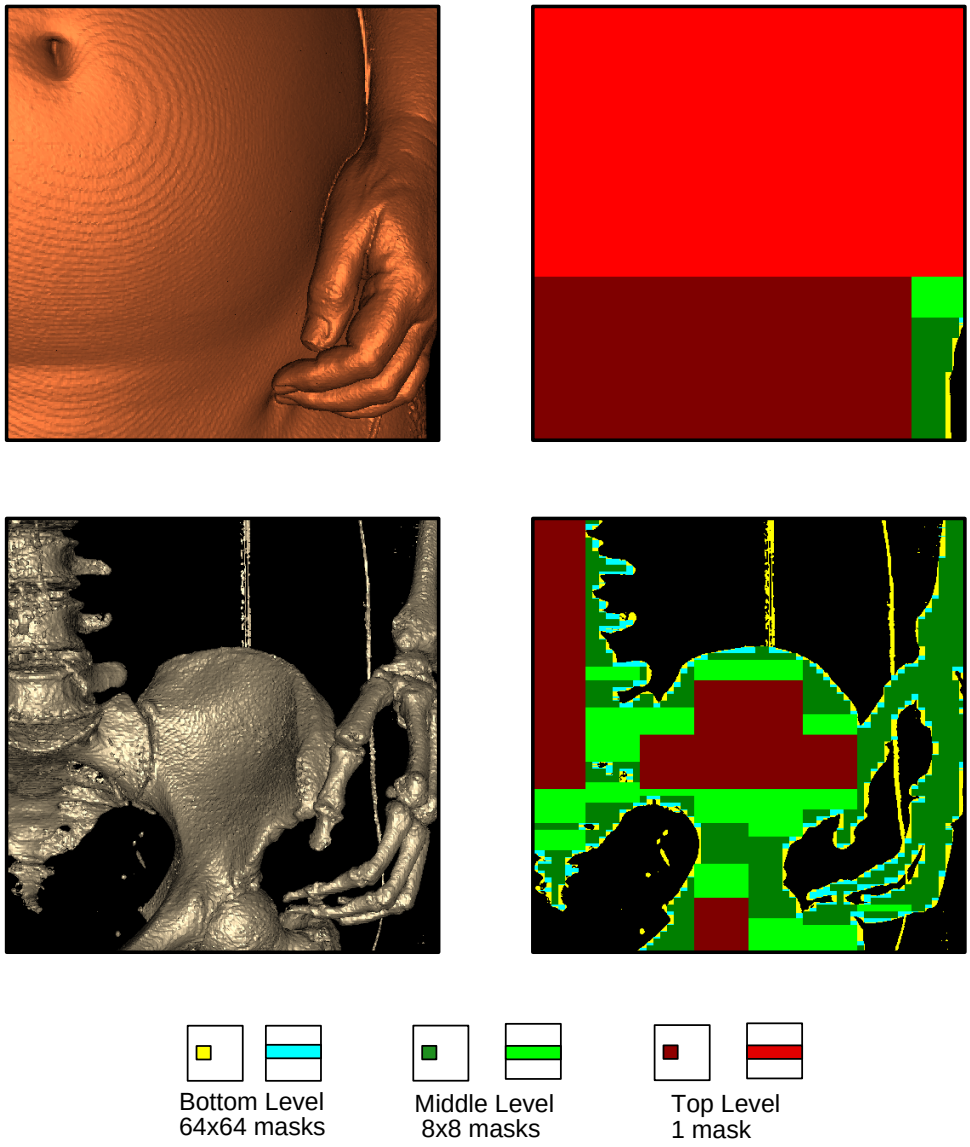


Figure 6.8. Two isosurfaces from the same point of view and their corresponding hierarchical visibility masks. Instead of showing a pyramid of the hierarchical mask levels, the levels are depicted by the pixels they cover. Each pixel is colored based on the highest level in the hierarchy in which it is covered. In addition, different colors are used when an entire row (eight pixels) in a mask is covered.

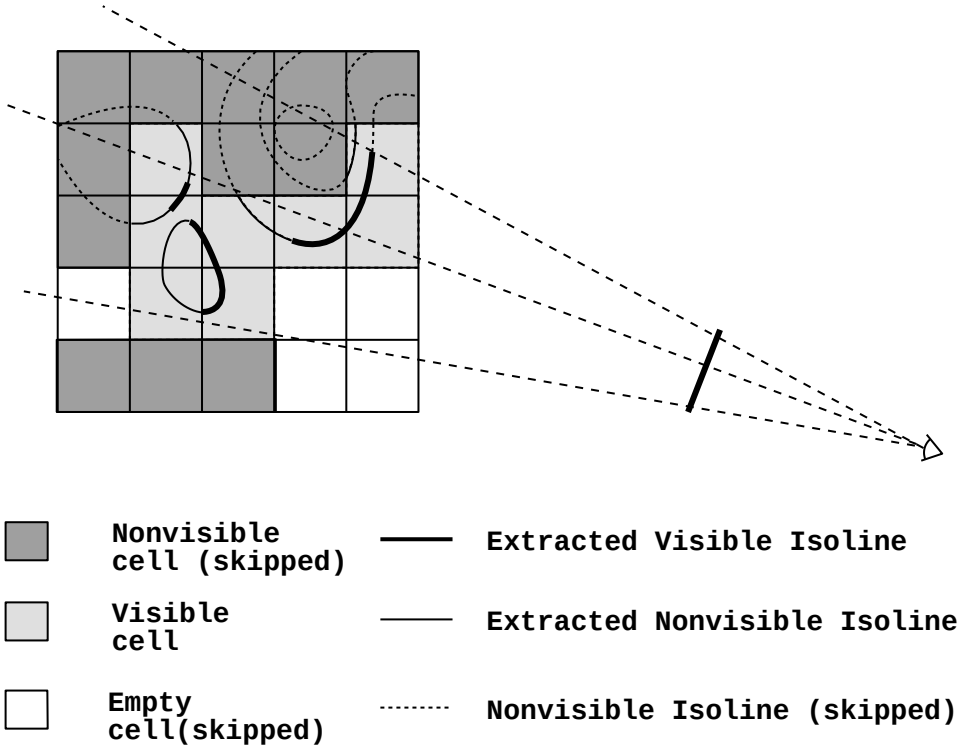


Figure 6.9. Cells and isolines visibility.

the nonlinearity of the perspective transformation prohibits the use of a precomputed transformation table. To accelerate this critical step, we take advantage of the shear-warp factorization of the viewing transformation.

6.2.3.1 Shear-Warp Factorization

In 1994, Lacroute [14, 15] presented a volume rendering method that was based on the shear-warp factorization of the viewing transformation. The underlying idea is to factor the viewing transformation into a shear followed by a warp transformation. The data are first projected onto a sheared object space that is used to create an intermediate, albeit warped, image. Once this image is complete a two-dimensional warping transformation is applied to create the correct final image. Figure 6.10 illustrates the shear-warp transformation for an orthographic projection. For a perspective projection the slices need also to be scaled as seen in Figure 6.11.

The advantage of this method is that the intermediate image is aligned with one of the

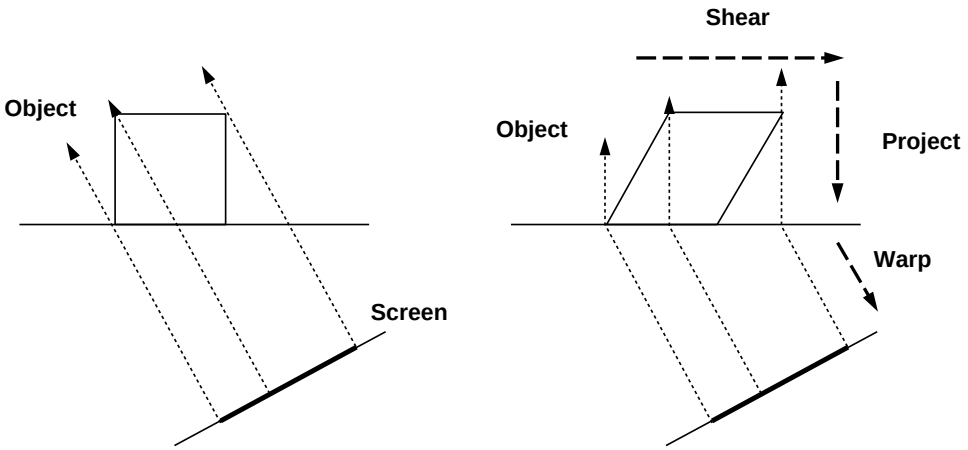


Figure 6.10. Shear-warp in orthographic projection.

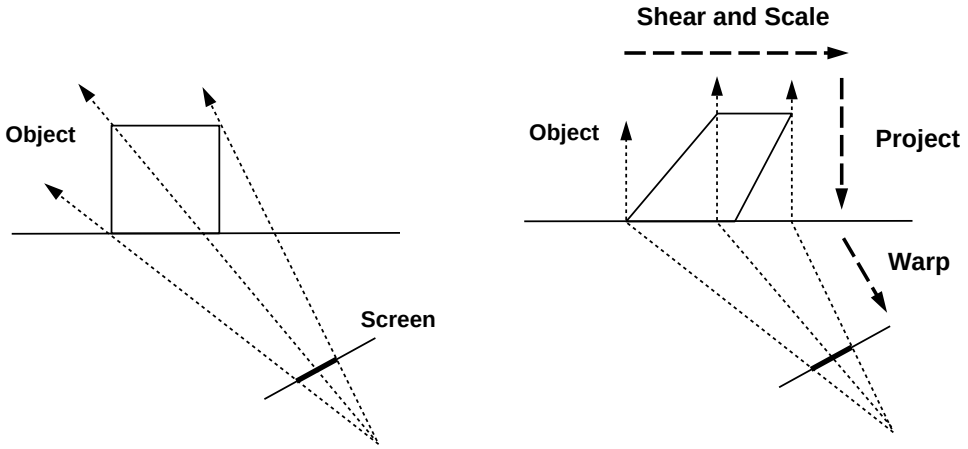


Figure 6.11. Shear-warp in perspective projection.

dataset faces. This alignment enables the use of a parallel projection of the 3D dataset. The warp stage is then applied to a two-dimensional image rather than to each data point.

6.2.3.2 Shear But No Warp

We now note that the visibility on the image plane and on the warped projection plane are the same, see Figure 6.12. In other words, any point in the dataset that is visible on the image plane is also visible on the warped projection plane and similarly, points which would be occluded on the image plane are also occluded on the warped plane.

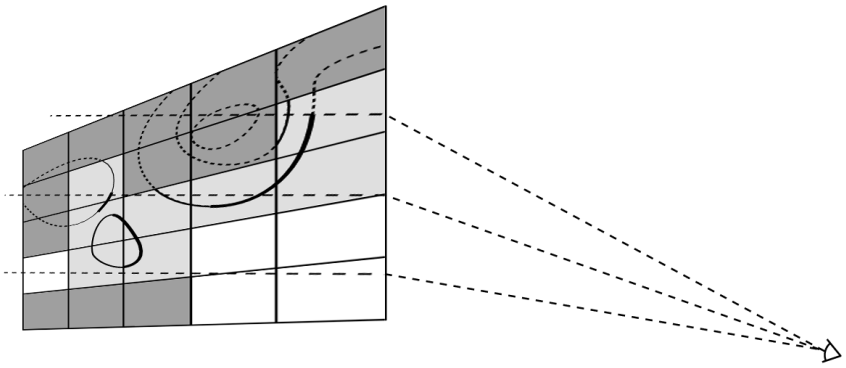


Figure 6.12. Warped space.

It is therefore sufficient to perform the visibility tests on the warped projection plane. The advantage of this approach is twofold. First, the perspective projection is removed. Second, since the shear and scale factors are, with respect to the current view point, constant for each slice, we can precompute them once for each new view point.

Let $[X, Y, Z]$ be the coordinate system of the dataset and let $s = [s_x, s_y, s_z]$ be the scaling vector of the data with respect to this coordinate system. Let us assume, without loss of generality, that the current warped projection plane is $Z = 0$. We first transform the current eye location onto the $[X, Y, Z]$ coordinate system and then precompute the shear and scale coefficients,

PrecomputeWarpParameters(eye, s)

foreach z

$f = z * s_z / z * s_z - eye_z$

$scale_x[Z] = (1 - f) * s_x$

$scale_y[Z] = (1 - f) * s_y$

$shear_x[Z] = f * eye_x$

$shear_y[Z] = f * eye_y$

The projection of any grid point $p(x, y, z)$ can now be computed as,

ProjectGridPoint(p)

$$x = p_x * scale_x[p_z] + shear_x[p_z]$$
$$y = p_y * scale_y[p_z] + shear_y[p_z]$$

for a total of two multiplications and two additions per vertex.

While the Z coordinate of every grid point is known in advance and thus the shear and scale factor can be precomputed for each new view point, the same does not hold true for the vertices of the isosurface triangles. However, since the projection onto the warped projection plane is orthographic it can be shown (see Appendix B) that a vertex projection is,

ProjectVertex(p)

$$f = p_z/(z - eye_z)$$
$$x = p_x + f * (eye_x - p_x)$$
$$y = p_y + f * (eye_y - p_y)$$

for a total of two multiplications, five additions and one division.

An example of an image and its warped visibility mask is shown in Figure 6.13.

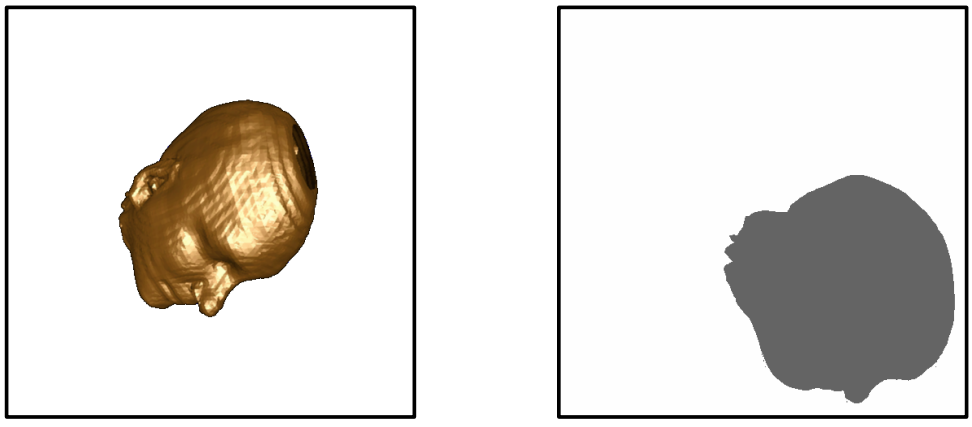


Figure 6.13. An image and its warped visibility mask.

6.2.4 Results

We tested our method with two datasets. The first is a small dataset (64^3) of a CT scanned head where most of the internal structure, e.g., the brain, was removed by hand segmentation. This results in lower depth complexity. The larger dataset (256^3) has a large number of internal structures that normally would be extracted as isosurfaces when one extracts the skin. We ran experiments in two scenarios. In one scenario, the isosurfaces were extracted and rendered on the same high-end machine, an SGI $Onyx2$ (using a single CPU). In the second scenario, we used a lower-end machine (SGI Indigo2 Extreme connected with a 100-BaseT switched Ethernet) for the rendering phase and the $Onyx2$ for the isosurface extraction.

The results for the first scenario, Table 6.1, show that for larger and more complex isosurfaces the new method still outperforms extraction of the full isosurface, though the performance falls short when only a change of the view point is considered.

The results for the second scenario, Table 6.2, demonstrate the advantage of this method. Even when the time for the extraction is added to each new view position, the new method out performs a regular full octree traversal and extraction due to the LAN bandwidth limitations.It is important to note the large reduction in the size, up to a factor of 15, of the extracted isosurface.

Table 6.1. Scenario I: Local Visualization

method	view	extraction time(sec)	polygons	points	rendring time(sec)
Small dataset					
Octree	any	0.48	46,222		0.16
VD	normal	0.79	9036		0.06
VD	closeup	0.59	7995		0.02
VD	zoom out	0.60	5938	1,112	0.02
Large dataset					
Octree	any	3.86	353,868		1.28
VD	normal	2.56	22,405		0.04
VD	closeup	0.85	5,588		0.03
VD	zoom out	0.99	1,080	7,888	0.02

Table 6.2. Scenario II: Remote Visualization

method	view	extraction. time(sec)	polygons	points	rendring time(sec)
Small dataset					
Octree	any	0.42	46,222		1.35
VD	normal	0.79	9184		0.27
VD	closeup	0.31		2735	0.05
VD	zoomout	0.40	2154	2319	0.11
Large dataset					
Octree	any	3.57	342,640		10.58
VD	normal	2.31	20,330		0.60
VD	closeup	1.14	6,078	5,374	0.30
VD	zoomout	0.38		7,364	0.12

6.3 The SAGE

A view dependent approach to isosurface extraction provides a fast and economical imaging of complex isosurfaces. This approach is especially suited for applications such as remote visualization where many isosurfaces are generated and transmitted over a network.

The WISE algorithm, presented in Section 6.2, provides a particular implementation of the view dependent approach. The performance of the WISE algorithm demonstrates the potential benefits of such an approach. The two most prominent weaknesses of the WISE method are the ratio of triangle intersections per screen cell and the fill rate of the screen tiles hierarchy.

In the following we present a new approach to view dependent isosurface extraction that aims at addressing these weaknesses. This approach is based on the WISE method and lessons learned from it and assumes that most of the extracted triangles are fairly small and that the contribution of each triangle to the final image is also small. Based on these assumptions, the triangles are applied to the screen hierarchy in a bottom-up approach (most refined to less refined). The bottom-up approach helps to restrict the rendering of a triangle to a very small part of the screen hierarchy data structure. The visibility tests of the data meta-cells are still performed in a top-down fashion in order to take advantage of the relative large size of their footprint on the screen.

Another potential inefficiency in the WISE method was inherent from the hierarchical coverage masks. The original method, proposed by Greene [11], is restricted to *convex* polygons. In contrast, polygonal isosurface extraction based on the marching cube approach, generates a group of triangles per cell that as a group are not necessarily convex. Due to the restriction of Greene's method to convex polygon, the WISE method had to project each triangle separately. To address this problem, the SAGE method scan converts the triangles rather then using coverage masks. The scan conversion approach can handle many polygons at the same time, none of which need to be convex.

Finally, the use of the exact screen resolution provides an easier way of identifying [meta-] cells that are smaller than a single screen pixel. These [meta-] cells can be rendered as points that reduce both the extraction time and memory.

6.3.1 A WISE lesson

The WISE algorithm performed well for isosurfaces when the projected triangles covered a large area of the screen. Using the hierarchical tiles enabled masking large areas at higher levels of the hierarchy. Table 6.3 shows a comparison between the area of a projected triangle and the average number of times the triangle was intersected against the hierarchy tiles. The table also shows the average number of checks that where performed after the projection in order to keep the hierarchy consistent. As the size of the triangles shrink so do the the number of intersections and updates per triangles. Yet, even with the use of the look-ahead the number of operations per cell is large. In addition, large datasets tend to produce many small polygons.

Another limitation of the WISE algorithm is the restriction to the use of triangles. This restriction is due to the requirement of the tiles method that the projected polygons must be convex. The marching cube algorithm generates between one and four triangles per cell with an average of about 2.05 triangles per cell for the datasets used in this work. Since a cell can be viewed from any direction it is not possible to determine a prioriif the projection of more than one triangle will be a convex or a concave polygon. The WISE algorithm, thus, projects each and every triangle separately.

Furthermore, as each triangle edge is shared by two triangles, the edge is projected and intersected against the hierarchy twice. The only exceptions are silhouette edges, which are rendered only once. The use of triangles therefore does not permit elimination of those edges that are shared between triangles in the same cell and that form a convex polygon when projected onto the screen.

Table 6.3. Operation per Triangle Area

triangle area	intersections	checks
30	38	6.75
7	22	4.56

6.3.2 A Bottom Up Approach

To alleviate the problem of projecting many small triangles down the hierarchical tile structure, SAGE employs a bottom-up approach, Figure 6.14. This approach is based on the observation that the contribution of a small triangle is limited to only a small neighborhood in the hierarchy, i.e., few tiles at the lowest level. This contribution will also be limited in the number of levels in the hierarchy.

The bottom-up approach is realized by projecting the triangles directly on to the bottom level, which is at the screen resolution. Only the tiles that are actually changed by the projection of the triangle will be further checked to see if they cause changes up the hierarchy. Since the contribution of the triangle is assumed to be small, its effect up the hierarchy will also be minimal.

Visibility checks of the dataset meta-cells involve, on the other hand, large convex polygons and thus are suitable for a top-down culling approach. Furthermore, these tests need determine only if any part of a projected cell is visible, i.e., whether there is at least one unset pixel that is covered by the cell projected polygon. As with the WISE algorithm, it is important to reduce the time spent on the visibility culling, especially for cells that are partially visible. The idea is that a [partially-] visible meta-cell will be further checked by applying the visibility test on its children. A nonvisible meta-cell, on the other hand, provides a benefit by eliminating the need to examine its children. It is therefore appropriate to invest more test resources on nonvisible meta-cells. One does

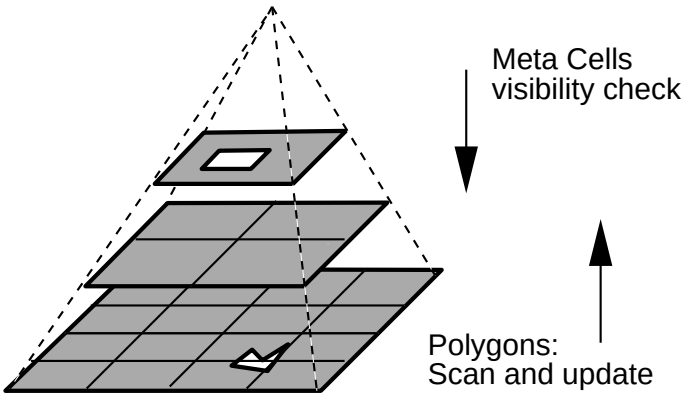


Figure 6.14. Bottom-up and top-down usage in SAGE.

not know if the meta-cell is visible or not in advance, which is why the visibility tests were devised. The point is that a visibility check should be able to determine that a cell *is* visible quicker than if the cell is *not* visible.

To address this concern, the visibility tests in SAGE are performed in a top-down approach using the bounding box of the projected cell. Figure 6.15 shows an example of a bounding box of a projected cell inside a particular screen tile. In addition to the bounding box, Figure 6.15 shows the inner and outer coverage masks of this bounding box. These coverage masks are aligned with the subtiles. If any of the subtiles under the inner coverage mask is not completely covered, than the cell is considered visible. If any of the sub tiles under the outer mask (but not under the inner mask) are not covered than the cell might be visible and these subtiles may need to be checked further. Note that this visibility test is an over estimator and may classify a cell as visible when it is not, yet it will not misclassify a visible cell as hidden.

6.4 Fast Estimates of a Bounding Box of a Projected Cell

The use of the visibility tests adds an overhead to the extraction process that should be minimized. The meta-cell visibility tests can be accelerated by approximating the screen area covered by a meta-cell rather than computing it exactly. In general the projection of

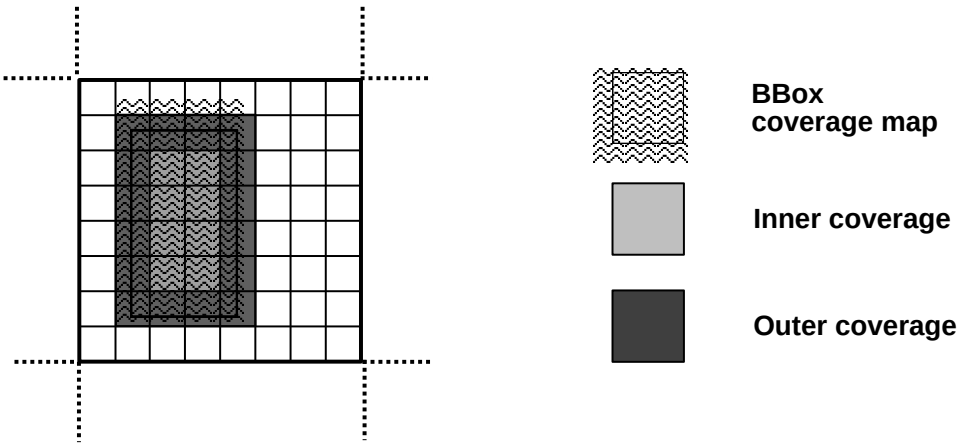


Figure 6.15. Tile coverage of a bounding box of a projected cell.

a meta-cell on the screen has a hexagon shape with nonaxis aligned edges. We reduce the complexity of the visibility test by using the axis aligned bounding box of the cell projection on the screen as seen in Figure 6.16. This bounding box is an overestimate of the actual coverage and thus will not mis-classify a visible meta-cell, though the opposite is possible.

The problem is, thus, how to find this bounding box quickly. The simplest approach is to project each of the eight vertices of each cell on to the screen and compare them. This process involves eight perspective projections and either two sorts (x and y) or 16 to 32 comparisons.

The solution in SAGE is to approximate the bounding box as follows. Let P be the

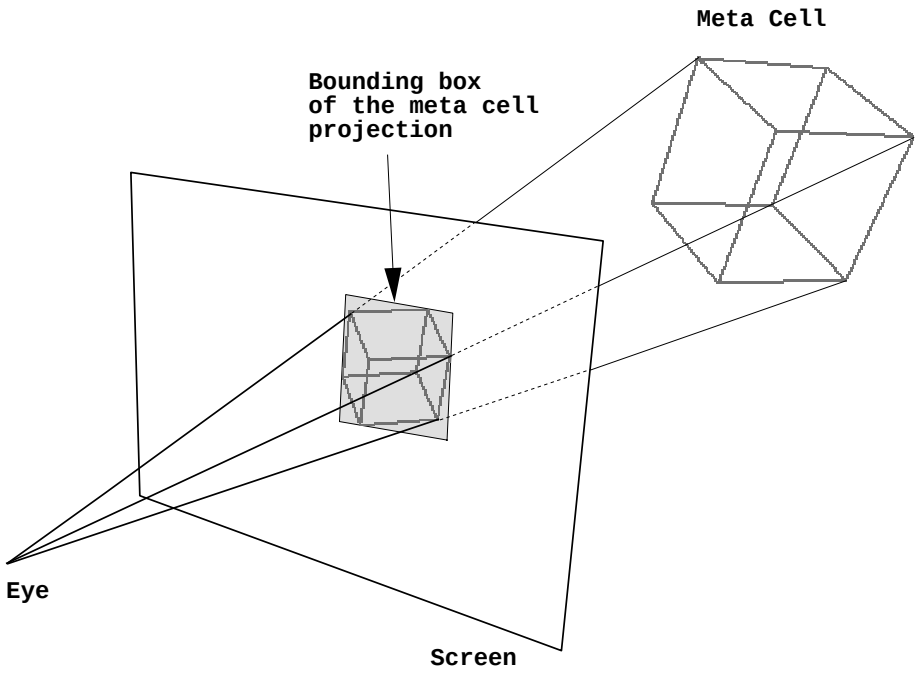


Figure 6.16. Perspective projection of a meta-cell, the covered area and its bounding box.

center of the meta-cell in object space. Assuming the size of the meta-cell is (dx, dy, dz), we define the eight vectors

$$D = (\pm\frac{dx}{2}, \pm\frac{dy}{2}, \pm\frac{dz}{2}, 1)$$

The eight corner vertices of the cell can be represented as

$$V = P + D = P + (\pm D_x, \pm D_y, \pm D_z)]$$

Applying the viewing matrix M to a vertex V amounts to:

$$VM = (P + D)M = PM + DM$$

After the perspective projection the x screen coordinate of the vertex is:

$$\frac{[VM]_x}{[VM]_w} = \frac{[PM]_x + [DM]_x}{[PM]_w + [DM]_w}$$

To find the bounding box of the projected meta-cell we need to find the maximum and minimum of these projections over the eight vertices in both x and y. Alternatively, we can overestimate these extrema values such that we may classify a nonvisible cell as visible but not the opposite. Overestimating can thus lead to more work but will not introduce additional errors.

The maximum x screen coordinate can be estimated as follows,

$$\begin{aligned}
\max(\frac{[VM]_x}{[VM]_w}) &\leq \frac{\max([PM]_x + [DM]_x)}{\min([PM]_w + [DM]_w)} \\
&\leq \frac{[PM]_x + \max([DM]_x)}{\min([PM]_w + [DM]_w)} \\
&\leq \frac{[PM]_x + [D^+M^+]_x}{\min([PM]_w + [DM]_w)}
\end{aligned}$$

where we define the $^+$ operator to mean to use the absolute value of the vector or matrix elements.

Assuming that the meta-cells are always in front of the screen we have

$$V_z > 0 \Rightarrow P_z - D_z^+ > 0 \Rightarrow [PM]_z - [D^+M^+]_z > 0$$

thus,

$$\max \frac{[VM]_x}{[VM]_w} = \begin{cases} \frac{[PM]_x + [D^+M^+]_x}{[PM]_w - [D^+M^+]_w} & \text{if numerator} \geq 0 \\ \frac{[PM]_x + [D^+M^+]_x}{[PM]_w + [D^+M^+]_w} & \text{otherwise} \end{cases}$$

Similarly, the minimum x screen coordinate can be overestimated as,

$$\min \frac{[VM]_x}{[VM]_w} \leq \begin{cases} \frac{[PM]_x - [D^+M^+]_x}{[PM]_w + [D^+M^+]_w} & \text{if numerator} \geq 0 \\ \frac{[PM]_x - [D^+M^+]_x}{[PM]_w - [D^+M^+]_w} & \text{otherwise} \end{cases}$$

The top and bottom of the bounding box are computed similarly.

The complete procedure for estimating the bounding box (see Figure 6.17) requires only two vector matrix multiplication, two division, four multiplications four comparison and six additions.

6.4.1 Scan Conversion of Concave Polygons

One of the disadvantages of the top-down approach, as discussed in Section 6.3.1, is that it is restricted to only convex polygons. In the WISE algorithm, this restriction has forced the projection of triangles only one triangle at a time.

To alleviate this restriction, the SAGE algorithm employs the scan convert algorithm, which simultaneously project a collection of triangles and concave polygons. The use of the scan conversion algorithm is made particularly simple in SAGE due to the bottom-up update approach. The projected triangles and polygons are scan-converted at screen resolution at the bottom level of the tiles hierarchy before the changes are propagated up the hierarchy. Applying the scan line in a top-down fashion would have made the algorithm unnecessarily complex.

Additional acceleration can be achieved by eliminating redundant edges, projecting each vertex only once per cell and using triangles strips or fans. To achieve these goals, the marching cubes lookup table is first converted into a triangles fans format. The usual marching cubes lookup table contains a list of the triangles (three vertices) per case.

PrecomputeM^+:
$M = ViewMatrix$
$M^+_{i,j} = \|M_{i,j}\|$

Compute($t,\ f_1,\ f_2$ **)**
return $t > 0\ ?\ t * f_1\ :\ t * f_2$

FindBouningBox(cell)
$P =$ center of cell
$D = (\frac{dx}{2}, \frac{dy}{2}, \frac{dz}{2}, 1)$
$PM = p * M$
$DM = d * M^+$

$f_{far} = \frac{1}{PM_w + DM_w}$

$f_{near} = \frac{1}{PM_w - DM_w}$

$right$ = Compute($PM_x + DM_x,\ f_{near},\ f_{far}$)
$left$ = Compute($PM_x - DM_x,\ f_{far},\ f_{near}$)
top = Compute($PM_y + DM_y,\ f_{near},\ f_{far}$)
$bottom$ = Compute($PM_y - DM_y,\ f_{far},\ f_{near}$)

Figure 6.17. Procedure for (over)estimating the bounding box of a projected cell.

Appendix C lists the new lookup table based on triangle fans and which lists each vertex only once per case.

Figure 6.18 depicts four configurations of a pair of triangles. The shared edge in the left two cases is redundant and should be removed from the list of edges to be scan lined. However, the shared edge in the two cases on the right should be included twice. Rather than including two records of the same edge, the shared edge can be marked as such and will be counted twice whenever it is accounted for in the scan line algorithm.

A comparison of the WISE and the SAGE algorithms with respect to the number of polygons and edges that are projected onto the hierarchical tiles is shown in Figure 6.19.

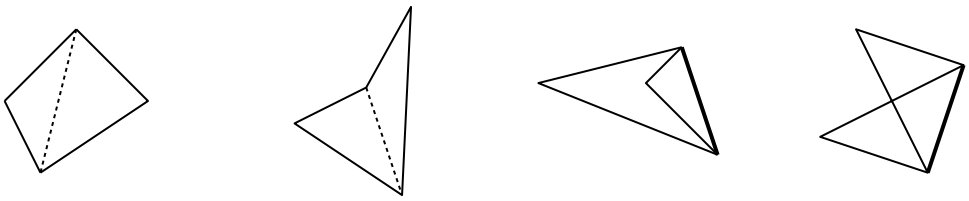


Figure 6.18. Redundant shared edges. The two examples on the left show a shared that should be eliminated. The two examples on the right show a shared edge that should be treated as a single double edge.

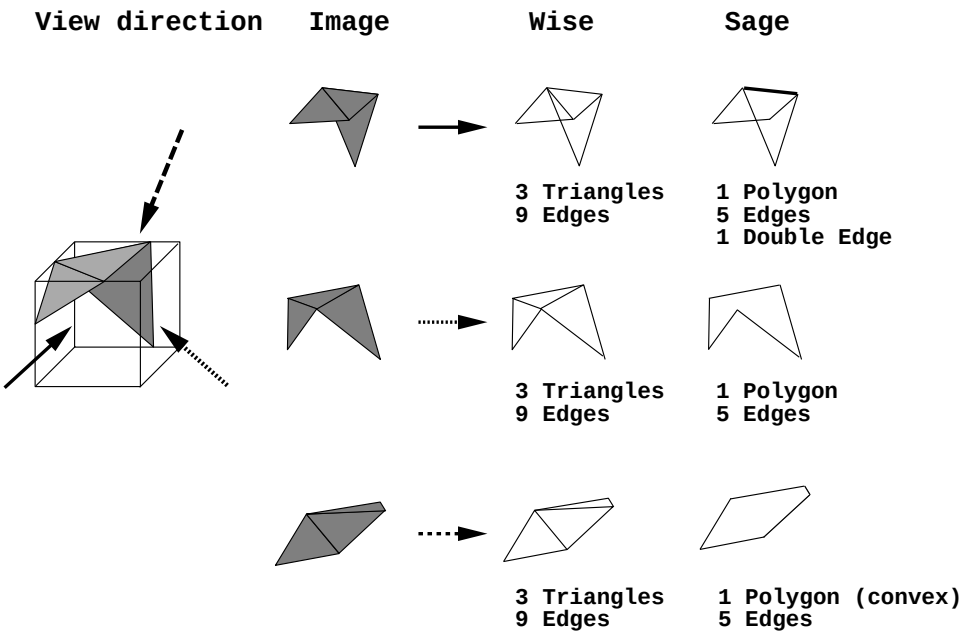


Figure 6.19. Comparison between WISE and SAGE.

6.4.2 Rendering Points

Another potential saving is achieved by using points with normals to represent triangles or [meta-] cells which are smaller than a single pixel. This is an improvement over the WISE algorithm as the exact location of the each screen pixel center is known during the scan line and the visibility tests. Whenever a nonempty [meta-] cell is determined to have a size less then a single pixel and its projection covers the center of a pixel, it is represented by a single point. Figure 6.20 shows an example in which some of the cells

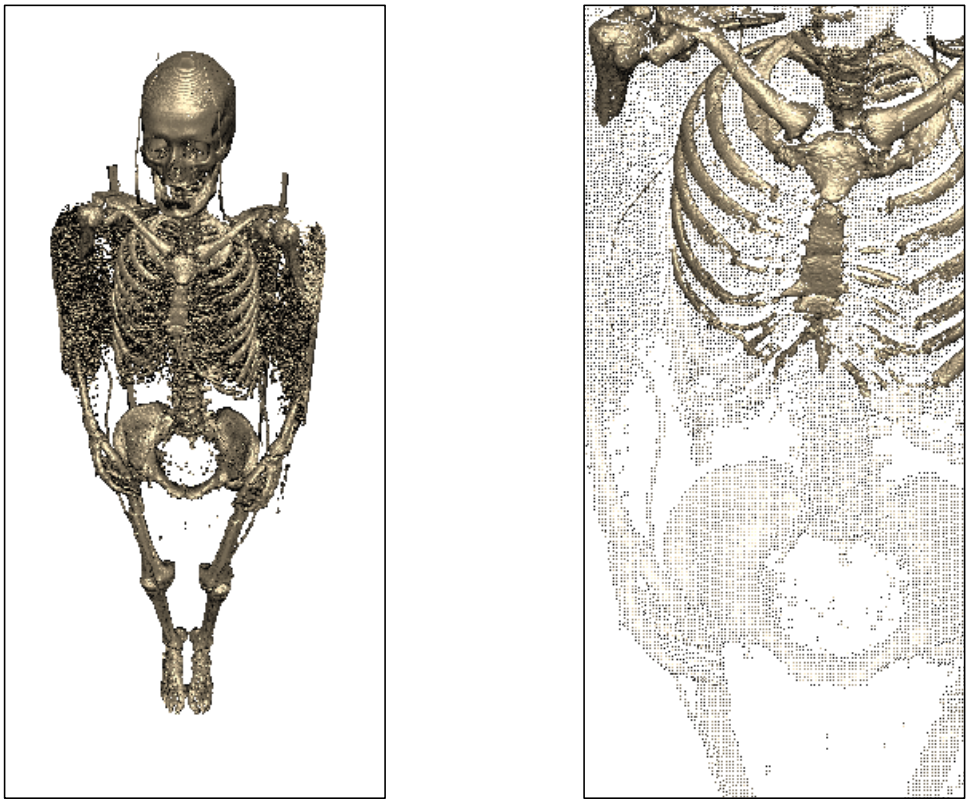


Figure 6.20. Rendering points. The left image was extracted based on the current view point. The right image show a closeup of the same extracted geometry.

are far enough such that they can be rendered as point. On the left is the image as seen by the user while on the right is a close up view of the same extracted geometry (i.e.the user zoomed in but did not extract the geometry based on the new view point).

6.4.3 Results

To evaluate the performance of the SAGE method we compared it to the performance of the Octree, NOISE and WISE methods. We used three datasets from three different views using two isovalues and both local and remove visualization. For the third dataset we used only one isovalue (for the skin). All in all, 80 different test cases were performed for each on the four extraction methods.

The characteristics of the three datasets are shown in Table 6.4. The first two are sections from the visible woman dataset [26]. The third dataset has a large number of

Table 6.4. Datasets

name	dimensions	size	type	comments
v-head	512x512x208	104MB	CT	Visible woman
v-legs	512x512x617	308MB	CT	Visible women
head	256x256x256	64MB	CT	High depth complexity

internal structures with similar data values as the skin. This, in turn, results in a very large and complex isosurfaces for an isovalue of the skin. Most of this complex isosurface is hidden from the user behind the relatively small surface that represents the skin.

The tests cases included a normal view of each dataset, a closeup of a small section, and a distant view. Each test on the visible woman sections was performed twice using the isovalues corresponding to the skin (600.5) and bone (1224.5). Figures 6.21, 6.22 and 6.23 show the extracted isosurfaces for each test for each of the three datasets respectively. The tests were selected such that each will reflect different characteristics. In the visible woman case, the skin cases had large contiguous areas of coverage where as the isosurfaces for the bone exhibits complex structures with many holes and cavities. The closeup test demonstrates one of the benefits of view dependent isosurface extraction when only a small section of the isosurface in needed. In contrast, the distant tests shows examples where not even the *visible* isosurface should be extracted, rather only the visible portion with regard to the resolution of the screen. The size of the objects on the screen, in the case of the distance view, also corresponds to their size when the full dataset is display, see Figure 6.24.

The experiments were done using two scenarios. In one scenario, the isosurfaces were extracted and rendered on the same high-end machine, an SGI $Onyx2$ (using a single CPU). In the second scenario, we used a lower-end machine (SGI O2 with R5000 180Mhz, 192MB of memory) connected with a 100-BaseT switched Ethernet) for the rendering phase and the same SGI $Onyx2$ for the isosurface extraction. Table 6.5, Table 6.6 and Table 6.7 illustrate the results for these tests on the three datasets.

The results can be compared based on the size of the extracted isosurface, the extraction time and rendering both local and on a remote machine.

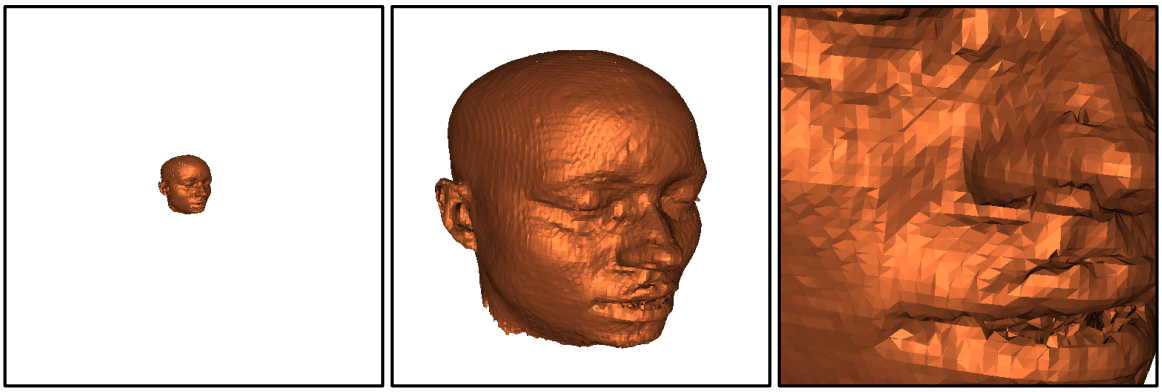
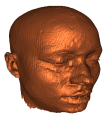
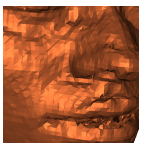


Figure 6.21. Three isosurfaces from the CT head dataset.

Table 6.5. Head Dataset

view	method	extraction time (sec)	number of polygons	local view time (sec)	remote view time (sec)	image
any	Octree NOISE	1.9 2.0	333,616	0.6	10.7	
distant	 WISE SAGE	 2.2 0.7	 34,008 11,153	 <0.1 <0.1	 1.1 0.4	
normal	 WISE SAGE	 2.6 0.7	 34,7024 31,9109	 0.9 <0.1	 1.0 1.0	
closeup	 WISE SAGE	 16.6 5.7	 7,480 5,847	 <0.1 <0.1	 0.3 0.2	

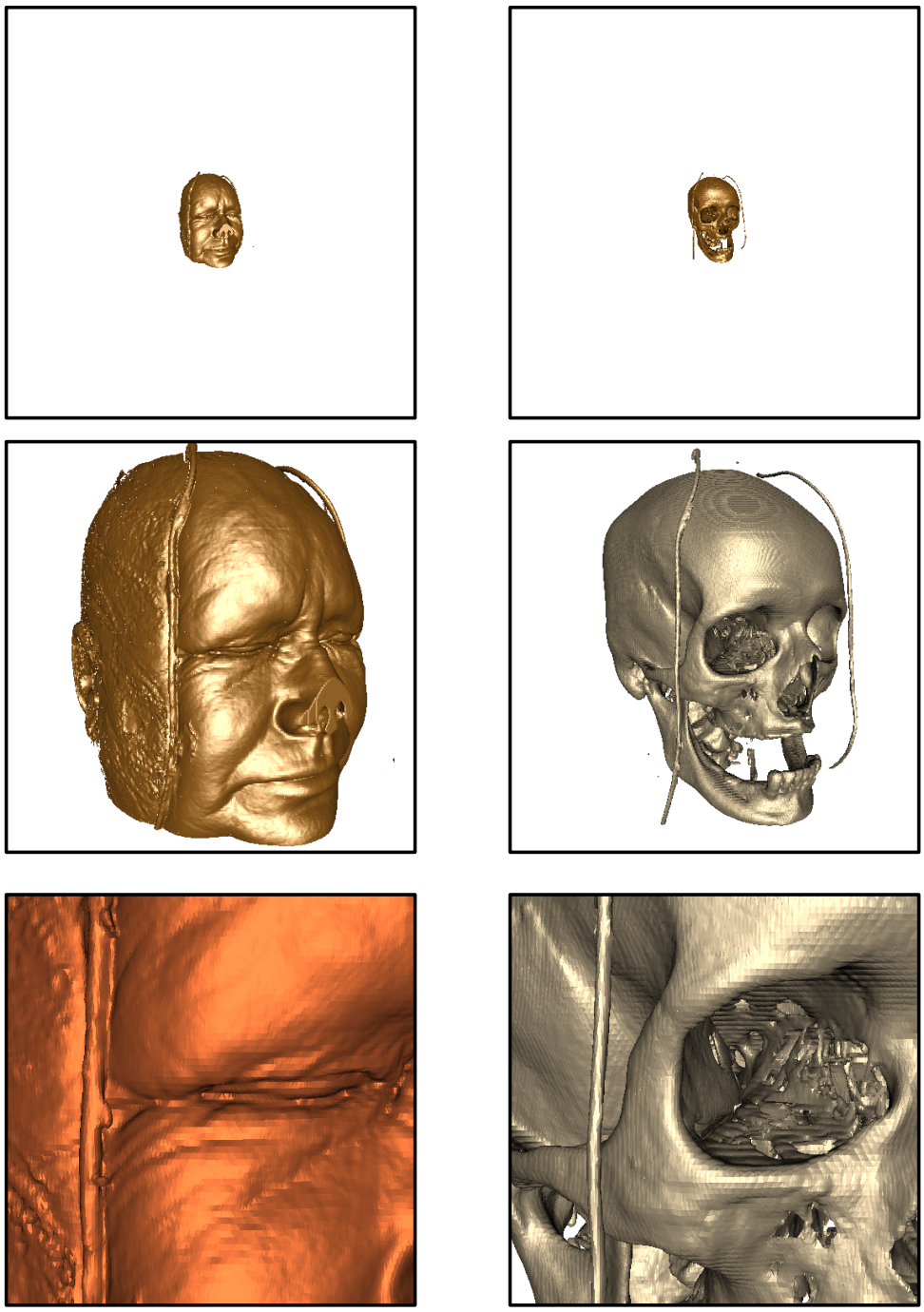


Figure 6.22. Six isosurface from the visible woman's head.

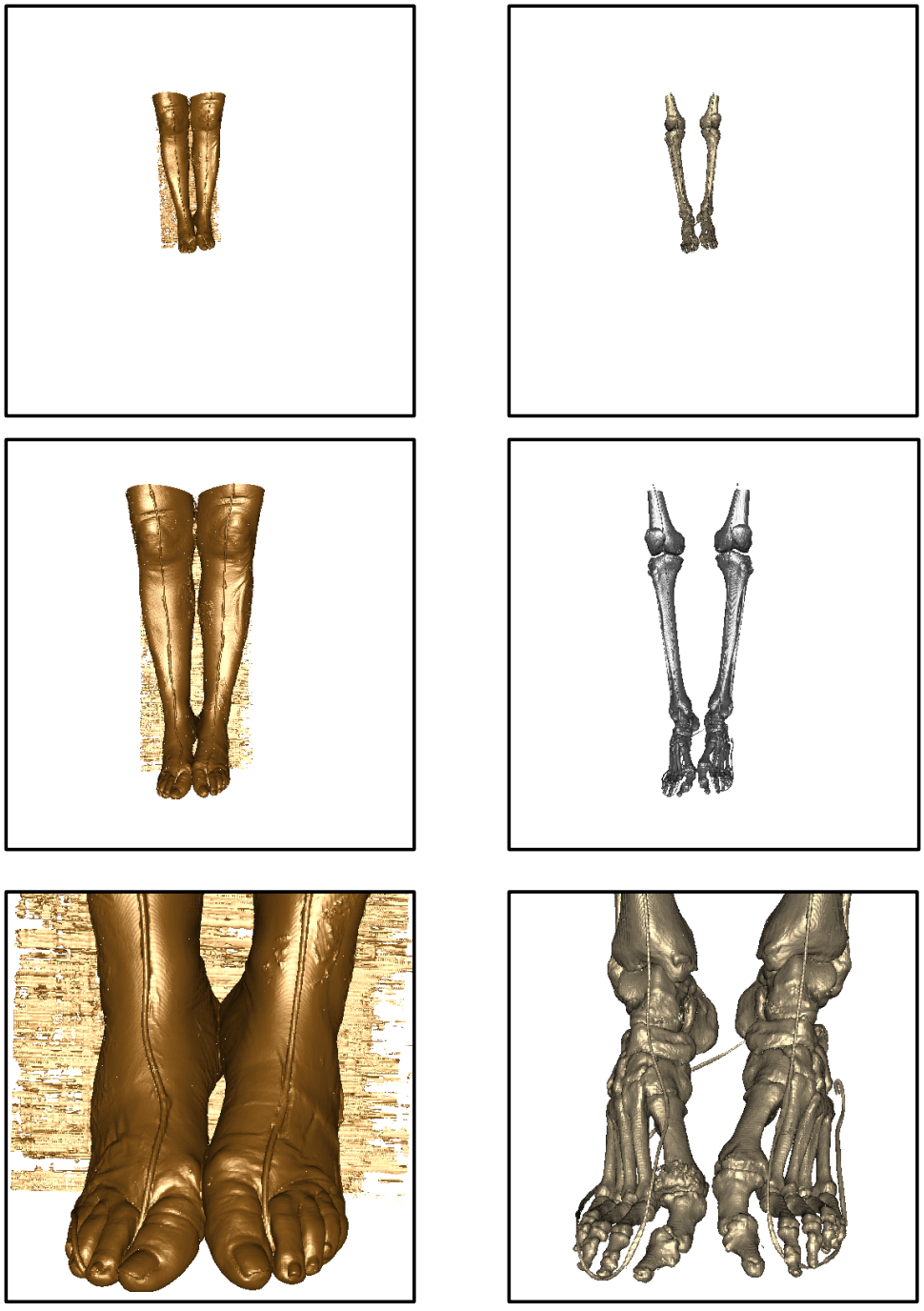


Figure 6.23. Six isosurface from the visible woman's legs.

Table 6.6. Visible Woman Head Dataset

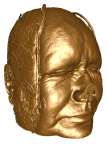
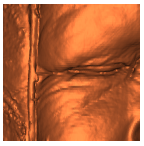
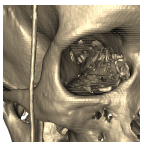
iso	view	method value value	extraction time (sec) time (sec)	number of polygons polygons	local view time (sec)	remote view time (sec)	image
skin	any	Octree NOISE	10.9 10.2	1,430,824	2.6	41.6	
	distant	WISE SAGE	35.1 3.4	292,242 18,645	0.6 <0.1	9.0 0.6	
	normal	WISE SAGE	35.8 4.4	344,628 195,408	0.6 0.3	9.2 5.4	
	closeup	WISE SAGE	4.6 0.6	43,222 36,939	0.1 <0.1	2.8 2.1	
bone	any	Octree NOISE	17.0 14.6	2,207,592	4.6	65.4	
	distant	WISE SAGE	13.9 4.1	271,075 12,747	0.5 <0.1	8.5 0.4	
	normal	WISE SAGE	32.7 4.5	278,735 153,617	0.7 0.2	8.5 5.0	
	closeup	WISE SAGE	10.6 1.5	84,599 67,808	0.2 0.2	1.3 1.1	

Table 6.7. Visible Woman Legs Dataset

iso	view	method value value	extraction time (sec) time (sec)	number of polygons polygons	local view time (sec)	remote view time (sec)	image
skin	any	Octree NOISE	33.4 27.1	3,264,755	6.2	117.6	
	distant	WISE SAGE	37.5 0.7	968,073 14,917	2.0 <0.1	28.4 0.2	
	normal	WISE SAGE	70.4 12.1	935,784 122,229	1.8 0.2	27.8 3.4	
	closeup	WISE SAGE	16.6 5.7	364,394 234,607	0.80 0.51	10.3 6.7	
bone	any	Octree NOISE	18.9 16.3	2,328,940	4.7	69.5	
	distant	WISE SAGE	18.0 0.5	412,530 6,099	0.9 <0.1	12.3 0.1	
	normal	WISE SAGE	33.3 6.8	406,262 503,545	0.9 0.1	12.1 1.5	
	closeup	WISE SAGE	9.6 3.8	180,270 122,262	0.5 0.3	5.8 3.5	

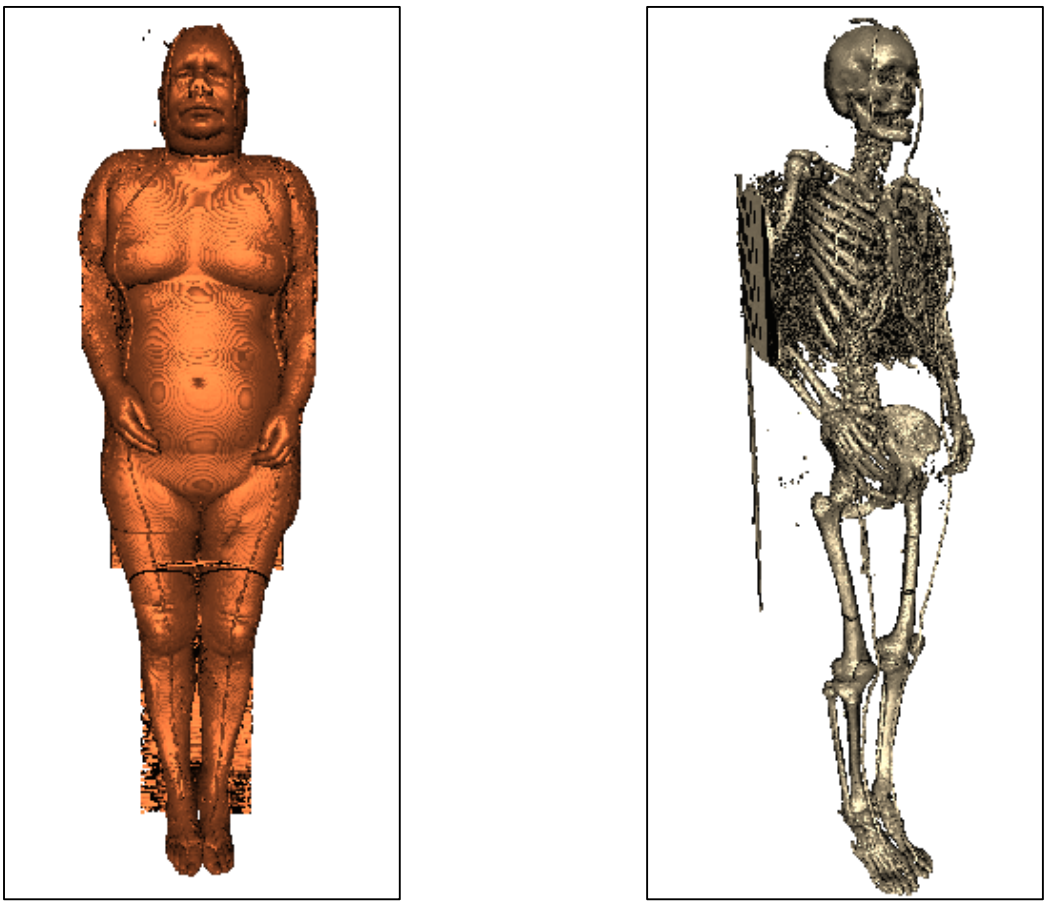


Figure 6.24. A view of the full visible woman dataset using the SAGE algorithm.

In all the tests the view dependent approaches, WISE and SAGE, consistently reduced the number of extracted polygons down to less than one percent in close up views and no worst than 28% in the worst case. These results depend on the depth complexity of the full isosurface as well as the visible portion of the isosurface and its footprint on the screen.

The SAGE method also shows improvement over the WISE method in extraction time. The WISE method extraction time exhibits large changes between different views, many time longer than the extraction of a full isosurface. In contrast the SAGE method consistently outperformed the full isosurface extraction methods.

The aim of the view-dependent approach are to reduce the extraction time and the

rendering time, especially for remote visulaization. In this context, both WISE and SAGE outperformed, on average, the display of a full isosurface by an order of magnitude in the case of local rendering. The SAGE method, again, consistently achieved better results that WISE, and outperformed a full isosurface rendering by up to two orders of magnitude.

With respect to remote rendering the view-dependent approach performance was the most noticeable. For all practical purposes, full isosurface rendering over the network is impossible as seen in these tests (40,60 seconds in Table 6.6 and even 117 seconds in Table 6.7). On the other hand, the SAGE consistently provided fast extraction and rapid respond to changes in the view parameters.

These results suggest that a view-dependent method such as SAGE can provide an investigator with a practical exploration tool. Furthermore, it is well suitable for remote investigating data that can not fit on a low-end machine and must be kept (or computed) on a separate supercomputer.

CHAPTER 7

CONCLUSIONS

Isosurface extraction is a powerful tool for investigating volumetric scalar fields. The position of an isosurface, as well as its relation to other neighboring isosurfaces, can provide clues to the underlying structure of the scalar field. In medical imaging applications, isosurfaces permit the extraction of particular anatomical structures and tissues. These isosurfaces are static in nature. A more dynamic use of isosurfaces is called for in many scientific computing applications, such as simulation of physical phenomena. In these applications scientists need to be able to change dynamically the isovalue in order to gain better insight into simulation results.

We presented isosurface generation algorithms that enable rapid exploration of large datasets both for local and for remote visualization. We analyzed the isosurface generation process as a whole and the components that make up that process and identified potential bottlenecks. In particular. we address the issues of isolating an isosurface and its geometric representation.

We suggested viewing isosurface extraction as a search problem based on mapping dataset cells onto a two-dimensional span space. It was shown that previous extraction methods, which were not based on the datasets geometry, could also be mapped onto this new space. Analysis of these mappings demonstrated the shortcomings of the previous extraction methods. We also proposed a new decomposition of the span space using a kd-tree and proved its near optimal worst-case time complexity. An additional method was proposed and implemented that enables an estimate of the size of an isosurface within a few milliseconds. This count mode was shown to depend only on the square root of the size of the data and does not depend of the actual size of the extracted isosurface.

The generation of a geometric representation of isosurfaces was also singled out as a potentially significant bottleneck. We addressed this issue by proposing the use of a view-

dependent isosurface extraction approach. Two methods were developed based on a three step paradigm: a) traversing the data in a hierarchical front-to-back order, b) perform visibility culling base on the projection of the data onto the screen and c) forwarding a geometry representation of the isosurface to a graphics hardware accelerator.

The first view dependent method is based on a combination of three components. First, a modified version of the hierarchical tiles, proposed by Greene [11], was used to to provide visibility culling. This method takes advantage of image space and object space coherency. Second, the coherency in the value space was exploited using Wilhems's and Van Gelder's augmented octree [27]. Finally, we employed a shear-warp factorization to perform the visibility test in the warped space. The use of the warped space provides large reduction in the number of operations that were required for vertices projection. This method, Warped IsoSurface Extraction (WISE), was shown to drastically reduce the extraction time as well as the rendering time, making it attractive for remote visualization.

The second view-dependent method, SAGE, provides further improvements over the WISE method. The SAGE is based on the observation that isosurfaces from large datasets contain many small triangles, many of which are smaller the a single pixel. Two improvements over WISE were investigated. First, the top-down approach used in WISE was augmented with a bottom-up traversal of the hierarchical tiles. By projecting the small triangles at the bottom level we were able to reduce the amount of computation required per triangle. The second improvement replaced the use of coverage masks, during the projections of triangles, with a scan line algorithm. This change removed the restriction on projecting only convex polygons and enabled the projection of multiple triangles and concave polygons simultaneously. The SAGE method was shown to perform well for both local and remote visualization providing fast isosurface extraction and rapid respond to changes in the view parameters.

7.1 Future Work

None of the methods presented in this work were implemented as a parallel program. We intend to investigate the use of parallel computers as a mean for accelerating the

extraction and especially the construction of the isosurfaces. Both WISE and SAGE demonstrated the potential of view dependent isosurface extraction, yet both were design for structured grids, for which front-to-back traversal is simple. Though many datasets, such as medical images and finite difference simulations, are based on structured grid, there is a great need for rapid exploration of unstructured grid based datasets. The NOISE method provides part of the solution of these datasets as it reduces the search time to practically a few milliseconds. The octree method was shown to be comparable to NOISE on structured grids yet it is not appropriate for unstructured grids.

Remote visualization is becoming an important area in its own right, yet the combination of remote visualization of isosurfaces has not been explored. This combination is especially hard as the geometry depends on an isovalue provided by the user. The isovalue is not known in advance and it changes during the exploration of the dataset. The dynamic nature of this type of exploration renders most of the work on remote visualization inappropriate. The WISE method provides a first solution to this issue, while the SAGE method provides the first practical solution. One should note that the remote visualization used in our tests consisted of opening a window on a remote display (the user machine) and rendering to that window over the network. A better approach that should be investigated is the use of a separate viewer that resides on the remote (user) machine and that can communicate with the extraction process on the large machine. The remote viewer can then enable the user to locally (and thus much faster) view and manipulate the view-dependent isosurface. Further research should target the integration, by the remote viewer, of several views of an isosurface into one more complete and coherent surface. Additional work can investigate the latency introduced by the intermediate network. A network aware isosurface extraction system can switch among different modes of extraction, transmission and local rendering, providing a dynamic system that could handle the size and complexity of the next generation datasets.

APPENDIX A

WORST-CASE ANALYSIS

A.1 Octree Isosurface Extraction

Wilhelms and Van Gelder [27] did not analyze the time complexity of their octree-based isosurface extraction algorithm, see Section 2.1.2. We now present a worst-case analysis of their method.

The octree used by Wilhelms and Van Gelder is derived from the geometry of the dataset and is *augmented* only by the minimum and maximum values of the cells in the tree. As such, the octree relies solely on geometry to group cells with close field values. On the other hand, the octree is guaranteed to be balanced. Also note that the data cells occur only on the leaves of the tree.

For simplicity, consider first the 1D case of a binary tree with n leaves. For a given k, we seek one of the groups of k leaves with the highest cost to locate. For $k = 1$, the cost is $\log n$; this suggests an estimate of $O(k \log n)$ for the worst-case. This is clearly an overestimate as many segments of the paths to these k cells are shared. When $k = 2$, the two paths from the root must share several intermediate nodes. The maximum cost will occur when only the root node is shared. Therefore,

$$\begin{aligned} T(n,1) &= \log(n) \\ T(n,k) &= 1 + 2T(n/2, k/2), \end{aligned}$$

which, for $k = 2^m$, leads to

$$T(n,k) = k - 1 + k \log(\frac{n}{k}).$$

As an example, $T(n,n) = 2n - 1$, since the a binary tree with n leaves has $n - 1$ internal nodes.

The general case for a d-dimensional tree follows immediately from the binary case. Let $p = 2^d$,

$$\begin{aligned} T_d(n,1) &= \log_p n \\ T_d(n,k) &= 1 + pT(n/p, k/p) \end{aligned}$$

Let $q = \log_p k$. The solution to the recursive formula is

$$\begin{aligned} T_d(n,k) &= \frac{p^q - 1}{p-1} + p^q \log_p (\frac{n}{k}) \\ &= \frac{k-1}{p-1} + \frac{k}{d} \log (\frac{n}{k}). \end{aligned}$$

For the special case of octree, $d = 3$, we have

$$T_3(n,k) = \frac{k-1}{7} + \frac{k}{3} \log (\frac{n}{k})$$

and a complexity of $O(k + k \log (\frac{n}{k}))$.

A.2 The Count Mode

A node in a kd-tree holds information regarding only the value used to split the current tree. This always forces a search algorithm to traverse at least one subtree. The best-case performance for the count mode is thus $O(\log n)$.

We now examine the worst-case complexity of the count mode. Referring to the optimized version, section 4.3.1, we find two cases. When the isovalue is less than the value at the root of the tree we need to traverse only one subtree. Otherwise, both subtrees are traversed, yet for one of them we now know that the min or max condition is satisfied. Clearly the worst-case involves the second case,

$$T(1) = 1 \tag{A.1}$$

$$T(n) = 1 + T(n/2) + T_m(n/2). \tag{A.2}$$

For the case where the min or max condition is satisfied there are again two cases. These cases, however, are different from each other only with respect to whether one of the subtrees is completely empty or full. In both these cases, only one subtree is descended. Moreover, the next level of this subtree can be skipped and the algorithm

descends directly to both subsubtrees. Note that the root of the subtree still needs to be checked. Therefore,

$$\begin{aligned} T_m(1) &= 1 \\ T_m(n) &= 1 + 2T_m(n/4) \\ &= \sum_{i=0}^{\log_4 n} 2^i \\ &= 2^{\log_4(n)+1} - 1 \\ &\leq 2\sqrt{n}. \end{aligned} \tag{A.3}$$

Substituting Equation A.3 in Equation A.2 and using Equation A.1 we get,

$$\begin{aligned} T(n) &\leq 1 + 2\sqrt{n} + T(n/2) \\ &= \log n + 2\sqrt{n} \sum_{i=0}^{\log(n)-1} 2^{-i/2} \\ &= \log n + 2\sqrt{n}\frac{1-\sqrt{2/n}}{1-1/\sqrt{2}} \\ &\leq \log n + 6\sqrt{n}. \end{aligned}$$

Hence a complexity of $O(\sqrt{n})$.

APPENDIX B

PROJECTION COSTS IN THE WISE ALGORITHM

B.1 Perspective Projection

Fig. B.1 depicts a 2D example of a point P which is warped into point W based on perspective projection. Assuming that the eye coordinates (x_e, z_e) is given in the object coordinate system and using the triangles similarity, we can derive,

$$\frac{x_e - x_w}{x_w - x_p} = \frac{-z_e}{z_p}$$

.

Note that since the eye is located on the other side of the warping plane, z_e is negative and thus the distance of the eye from the plane is $-z_e$.

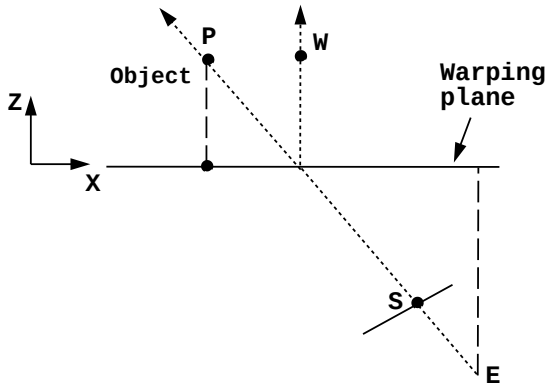


Figure B.1. Warped point location in a perspective projection.

Therefore,

$$x_w = \frac{x_p z_e - z_p x_e}{z_e - z_p}$$

Similarly, for the y coordinates we get,

$$y_w = \frac{y_p z_e - z_p y_e}{z_e - z_p}$$

The above expressions involve four multiplications, four additions and two divisions. However, further factorization leads to,

$$\begin{aligned} x_w &= \frac{x_p z_e + (-x_p z_p + x_p z_p) - z_p x_e}{e_z - z_p} \\ &= \frac{x_p(z_e - z_p) + z_p(x_p - x_e)}{e_z - z_p} \\ &= x_p + \frac{z_p}{e_z - z_p}(x_p - x_e) \\ &= x_p + C(x_p - x_e) \end{aligned}$$

where,

$$C = \frac{z_p}{e_z - z_p}$$

is used for both x_w and y_w.

The cost for a single point warp is thus one addition and one division for C and one multiplication and two additions for each of the x and y coordinates for a total of two multiplications, five additions and one division.

A full computation of the projection point S involves a multiplication of a 4D point with a 4×4 transformation matrix plus two divisions for the perspective projection. Since the z coordinate in screen space is not important one ends up with three sets of three multiplications and three additions plus two divisions for a total of nine multiplications, nine additions and two divisions.

APPENDIX C

TRIANGLE FAN LOOKUP TABLE FOR THE MARCHING CUBES

The SAGE method, Section 6.3, uses a triangle fan lookup table instead of the traditional marching cubes lookup table. The following is the modified lookup table which contains 256 entries similarly to the marching cubes . The first number in each row state the number of triangle fans, followed by a list of vertices. The vertices are listed in groups, one group per triangle fan, with a negative one (-1) separating the different groups. The first vertex in each such triangle fan is the base vertex of the fan with the rest of the vertices listed in a clockwise order around this base vertex. The vertices are mapped to the cell edges via the *edge_table* which list the two cell vertices that make up that edge.

```
static int edge_table[12][2] = {{0,1}, {1,2}, {3,2}, {0,3},
                                {4,5}, {5,6}, {7,6}, {4,7},
                                {0,4}, {1,5}, {3,7}, {2,6}};

struct TriangleCase {
  int n;  // number of triangle fans
  int vertex[16];
};


static TriangleCase tri_case[] = {
  /* 00000000 */
  { 0, { -1  -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 00000001 */
  { 1, {  0,  3,  8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 00000010 */
  { 1, {  0,  9,  1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 00000011 */
  { 1, {  1,  3,  8,  9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 00000100 */
  { 1, {  1, 11,  2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 00000101 */
  { 2, {  0,  3,  8, -1,  1, 11,  2, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
```

```
/* 00000110 */
{ 1, {  9, 11,  2,  0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00000111 */
{ 1, {  2,  3,  8,  9, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00001000 */
{ 1, {  3,  2, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00001001 */
{ 1, {  0,  2, 10,  8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00001010 */
{ 2, {  1,  0,  9, -1,  2, 10,  3, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00001011*/
{ 1, {  8,  9,  1,  2, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00001100 */
{ 1, {  3,  1, 11, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00001101 */
{ 1, {  0,  1, 11, 10,  8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00001110 */
{ 1, {  3,  0,  9, 11, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00001111 */
{ 1, {  9, 11, 10,  8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00010000 */
{ 1, {  4,  8,  7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00010001 */
{ 1, {  4,  0,  3,  7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00010010 */
{ 2, {  0,  9,  1, -1,  8,  7,  4, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00010011 */
{ 1, {  4,  9,  1,  3,  7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00010100 */
{ 2, {  1, 11,  2, -1,  8,  7,  4, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00010101 */
{ 2, {  3,  7,  4,  0, -1,  1, 11,  2, -1, -1, -1, -1, -1, -1, -1}},
/* 00010110 */
{ 2, {  9, 11,  2,  0, -1,  8,  7,  4, -1, -1, -1, -1, -1, -1, -1}},
/* 00010111 */
{ 1, {  2,  3,  7,  4,  9, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00011000 */
{ 2, {  8,  7,  4, -1,  3,  2, 10, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00011001 */
{ 1, { 10,  7,  4,  0,  2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00011010 */
{ 3, {  9,  1,  0, -1,  8,  7,  4, -1,  2, 10,  3, -1, -1, -1, -1}},
/* 00011011 */
{ 1, {  4,  9,  1,  2, 10,  7, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00011100 */
{ 2, {  3,  1, 11, 10, -1,  7,  4,  8, -1, -1, -1, -1, -1, -1, -1}},
/* 00011101 */
{ 1, {  1, 11, 10,  7,  4,  0, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00011110 */
{ 2, {  4,  8,  7, -1,  9, 11, 10,  3,  0, -1, -1, -1, -1, -1, -1}},
/* 00011111 */
{ 1, {  4,  9, 11, 10,  7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
```

```
/* 00100000 */
{  1, { 9,  4,  5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00100001 */
{ 2, {  9,  4,  5, -1,  0,  3,  8, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00100010 */
{ 1, {  0,  4,  5,  1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00100011 */
{ 1, {  8,  4,  5,  1,  3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00100100 */
{ 2, {  1, 11,  2, -1,  9,  4,  5, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00100101 */
{ 3, {  3,  8,  0, -1,  1, 11,  2, -1,  4,  5,  9, -1, -1, -1, -1}},
/* 00100110 */
{ 1, {  5, 11,  2,  0,  4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00100111 */
{ 1, {  2,  3,  8,  4,  5, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00101000 */
{ 2, {  9,  4,  5, -1,  2, 10,  3, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00101001 */
{ 2, {  0,  2, 10,  8, -1,  4,  5,  9, -1, -1, -1, -1, -1, -1, -1}},
/* 00101010 */
{ 2, {  0,  4,  5,  1, -1,  2, 10,  3, -1, -1, -1, -1, -1, -1, -1}},
/* 00101011 */
{ 1, {  2,  10, 8,  4,  5,  1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00101100 */
{ 2, { 11, 10,  3,  1, -1,  9,  4,  5, -1, -1, -1, -1, -1, -1, -1}},
/* 00101101 */
{ 2, {  4,  5,  9, -1,  0,  1, 11, 10,  8, -1, -1, -1, -1, -1, -1}},
/* 00101110 */
{ 1, {  5, 11, 10,  3,  0,  4, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00101111 */
{ 1, {  5, 11, 10,  8,  4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00110000 */
{ 1, {  9,  8,  7,  5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00110001 */
{ 1, {  9,  0,  3,  7,  5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00110010 */
{ 1, {  0,  8,  7,  5,  1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00110011 */
{ 1, {  1,  3,  7,  5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00110100 */
{ 2, {  9,  8,  7,  5, -1, 11,  2,  1, -1, -1, -1, -1, -1, -1, -1}},
/* 00110101 */
{ 2, { 11,  2,  1, -1,  9,  0,  3,  7,  5,  3, -1, -1, -1, -1, -1}},
/* 00110110 */
{ 1, {  8,  7,  5, 11,  2,  0, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00110111 */
{ 1, {  2,  3,  7,  5, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00111000 */
{ 2, {  7,  5,  9,  8, -1,  3,  2, 10, -1, -1, -1, -1, -1, -1, -1}},
/* 00111001 */
{ 1, {  9,  0,  2, 10,  7,  5, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
```

```
/* 00111010 */
{ 2, {  2, 10,  3, -1,  0,  8,  7,  5,  1, -1, -1, -1, -1, -1, -1, -1}},
/* 00111011 */
{ 1, { 10,  7,  5,  1,  2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00111100 */
{ 2, {  9,  8,  7,  5, -1, 11, 10,  3,  1, -1, -1, -1, -1, -1, -1, -1}},
/* 00111101 */
{ 1, {  0,  1, 11, 10,  7,  5,  9, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00111110 */
{ 1, {  0,  8,  7,  5, 11, 10,  3, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 00111111 */
{ 1, { 10,  7,  5, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01000000 */
{ 1, { 11,  5,  6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01000001 */
{ 2, {  0,  3,  8, -1,  5,  6, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01000010 */
{ 2, {  9,  1,  0, -1,  5,  6, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01000011 */
{ 2, {  1,  3,  8,  9, -1,  5,  6, 11, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01000100 */
{ 1, {  1,  5,  6,  2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01000101 */
{ 2, {  1,  5,  6,  2, -1,  3,  8,  0, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01000110 */
{ 1, {  9,  5,  6,  2,  0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01000111 */
{ 1, {  5,  6,  2,  3,  8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01001000 */
{ 2, {  2, 10,  3, -1, 11,  5,  6, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01001001 */
{ 2, { 10,  8,  0,  2, -1, 11,  5,  6, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01001010 */
{ 3, {  0,  9,  1, -1,  2, 10,  3, -1,  5,  6, 11, -1, -1, -1, -1, -1}},
/* 01001011 */
{ 2, {  5,  6, 11, -1,  9,  1,  2, 10,  8, -1, -1, -1, -1, -1, -1, -1}},
/* 01001100 */
{ 1, {  3,  1,  5,  6, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01001101 */
{ 1, {  0,  1,  5,  6, 10,  8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01001110 */
{ 1, {  3,  0,  9,  5 , 6, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01001111 */
{ 1, {  9,  5,  6, 10,  8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01010000 */
{ 2, {  5,  6, 11, -1,  4,  8,  7, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01010001 */
{ 2, {  4,  0,  3,  7, -1,  6, 11,  5, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01010010 */
{ 3, {  1,  0,  9, -1,  5,  6, 11, -1,  8,  7,  4, -1, -1, -1, -1, -1}},
/* 01010011 */
{ 2, { 11,  5,  6, -1,  9,  1,  3,  7,  4, -1, -1, -1, -1, -1, -1, -1}},
```

```
/* 01010100 */
{ 2, {  6,  2,  1,  5, -1,  4,  8,  7, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01010101 */
{ 2, {  1,  5,  6,  2, -1,  3,  7,  4,  0, -1, -1, -1, -1, -1, -1, -1}},
/* 01010110 */
{ 2, {  8,  7,  4, -1,  0,  9,  5,  6,  2, -1, -1, -1, -1, -1, -1, -1}},
/* 01010111 */
{ 1, {  9,  5,  6,  2,  3,  7,  4, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01011000 */
{ 3, {  3,  2, 10, -1,  7,  4,  8, -1, 11,  5,  6, -1, -1, -1, -1, -1}},
/* 01011001 */
{ 2, {  5,  6, 11, -1,  4,  0,  2, 10,  7, -1, -1, -1, -1, -1, -1, -1}},
/* 01011010 */
{ 4, {  0,  9,  1, -1,  4,  8,  7, -1,  2, 10,  3, -1,  5,  6, 11, -1}},
/* 01011011 */
{ 2, {  9,  1,  2, 10,  7,  4, -1,  5,  6, 11, -1, -1, -1, -1, -1, -1}},
/* 01011100 */
{ 2, {  8,  7,  4, -1,  5 , 6, 10,  3,  1, -1, -1, -1, -1, -1, -1, -1}},
/* 01011101 */
{ 1, { 10,  7,  4,  0,  1,  5,  6, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01011110 */
{ 2, {  0,  9,  5,  6, 10,  3, -1,  8,  7,  4, -1, -1, -1, -1, -1, -1}},
/* 01011111 */
{ 1, {  9,  5,  6, 10,  7,  4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01100000 */
{ 1, { 11,  9,  4,  6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01100001 */
{ 2, {  4,  6, 11,  9, -1,  0,  3,  8, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01100010 */
{ 1, {  0,  4,  6, 11,  1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01100011 */
{ 1, {  8,  4,  6, 11,  1,  3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01100100 */
{ 1, {  4,  6,  2,  1,  9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01100101 */
{ 2, {  3,  8,  0, -1,  4,  6,  2,  1,  9, -1, -1, -1, -1, -1, -1, -1}},
/* 01100110 */
{ 1, {  0,  4,  6,  2,  4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01100111 */
{ 1, {  2,  3,  8,  4,  6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01101000 */
{ 2, { 11,  9,  4,  6, -1, 10,  3,  2, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01101001 */
{ 2, {  0,  2, 10,  8, -1,  4,  6, 11,  9, -1, -1, -1, -1, -1, -1, -1}},
/* 01101010 */
{ 2, {  3,  2, 10, -1,  6, 11,  1,  0,  4, -1, -1, -1, -1, -1, -1, -1}},
/* 01101011 */
{ 1, {  1, 10,  8,  4,  6, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01101100 */
{ 1, {  6,  9,  1,  3, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01101101 */
{ 1, {  1,  0,  8, 10,  6,  4,  9, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
```

```
/* 01101110 */
{ 1, {  6, 10,  3,  0,  4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01101111 */
{ 1, {  6,  10, 8,  4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01110000 */
{ 1, {  7,  6, 11,  9,  8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01110001 */
{ 1, {  0,  3,  7,  6, 11,  9, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01110010 */
{ 1, {  1,  0,  8,  7,  6, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01110011 */
{ 1, {  1,  3,  7,  6, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01110100 */
{ 1, {  8,  7,  6,  2,  1,  9, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01110101 */
{ 1, {  9,  0,  3,  7,  6,  2, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01110110 */
{ 1, {  0,  8,  7,  6,  2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01110111 */
{ 1, {  7,  6,  2,  3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01111000 */
{ 2, {  2, 10,  3, -1,  8,  7,  6, 11,  9, -1, -1, -1, -1, -1, -1}},
/* 01111001 */
{ 1, {  7,  6, 11,  9,  0,  2, 10, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01111010 */
{ 2, {  7,  1,  0,  8, 11,  6, -1,  2, 10,  3, -1, -1, -1, -1, -1}},
/* 01111011 */
{ 1, {  1,  2, 10,  7,  6, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01111100 */
{ 1, {  6, 10,  3,  1,  9,  8,  7, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01111101 */
{ 2, {  0,  1,  9, -1, 10,  7,  6, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01111110 */
{ 1, {  0,  8,  7,  6, 10,  3, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 01111111 */
{ 1, {  7,  6, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 10000000 */
{ 1, {  7, 10,  6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 10000001 */
{ 2, {  3,  8,  0, -1, 10,  6,  7, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 10000010 */
{ 2, {  0,  9,  1, -1, 10,  6,  7, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 10000011 */
{ 2, {  8,  9,  1,  3, 10,  6,  7, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 10000100 */
{ 2, { 11,  2,  1, -1,  6,  7, 10, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 10000101 */
{ 3, {  1, 11,  2, -1,  3,  8,  0, -1,  6,  7, 10, -1, -1, -1, -1}},
/* 10000110 */
{ 2, {  2,  0,  9, 11, -1,  6,  7, 10, -1, -1, -1, -1, -1, -1, -1}},
/* 10000111 */
{ 2, {  6,  7, 10, -1, 11,  2,  3,  8,  9, -1, -1, -1, -1, -1, -1}},
```

```
  /* 10001000 */
  { 1, {  7,  3,  2,  6, -1,  -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -
1}},
  /* 10001001 */
  { 1, {  7,  8,  0,  2,  6, -1, -1, -1, -1, -1, -1, -1, -1 ,-1, -1, -1}},
  /* 10001010 */
  { 2, {  2,  6,  7,  3, -1,  0,  9,  1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 10001011 */
  { 1, {  1,  2,  6,  7,  8,  9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 10001100 */
  { 1, { 11,  6,  7,  3,  1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 10001101 */
  { 1, { 11,  6,  7,  8,  0,  1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 10001110 */
  { 1, {  0,  9, 11,  6,  7,  3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 10001111 */
  { 1, {  7,  8,  9, 11,  6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 10010000 */
  { 1, {  6,  4,  8, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 10010001 */
  { 1, {  3, 10,  6,  4,  0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 10010010 */
  { 2, {  8, 10,  6,  4, -1,  9,  1,  0, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 10010011 */
  { 1, {  9,  1,  3, 10,  6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 10010100 */
  { 2, {  6,  4,  8, 10, -1,  2,  1, 11, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 10010101 */
  { 2, {  1, 11,  2, -1,  3, 10,  6,  4,  0, -1, -1, -1, -1, -1, -1, -1}},
  /* 10010110 */
  { 2, {  4,  8, 10,  6, -1,  0,  9, 11,  2, -1, -1, -1, -1, -1, -1, -1}},
  /* 10010111 */
  { 1, {  3, 10,  6,  4,  9, 11,  2, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 10011000 */
  { 1, {  8,  3,  2,  6,  4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 10011001 */
  { 1, {  0,  2,  6,  4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 10011010 */
  { 2, {  1,  0,  9, -1,  2,  6,  4,  8,  3, -1, -1, -1, -1, -1, -1, -1}},
  /* 10011011 */
  { 1, {  4,  9,  1,  2,  6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 10011100 */
  { 1, {  4,  3,  8,  6,  1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 10011101 */
  { 1, {  0,  1, 11,  6,  4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 10011110 */
  { 1, {  3,  0,  9, 11,  6,  4,  8, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 10011111 */
  { 1, { 11,  6,  4,  9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 10100000 */
  { 2, {  4,  5,  9, -1,  7, 10,  6, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 10100001 */
```

```
{ 3, {  0,  3,  8, -1,  4,  5,  9, -1, 10,  6,  7, -1, -1, -1, -1, -1}},
/* 10100010 */
{ 2, {  5,  1,  0,  4, -1,  7, 10,  6, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 10100011 */
{ 2, { 10,  6,  7, -1,  3,  8,  4,  5,  1, -1, -1, -1, -1, -1, -1, -1}},
/* 10100100 */
{ 3, {  9,  4,  5, -1, 11,  2,  1, -1,  7, 10,  6, -1, -1, -1, -1, -1}},
/* 10100101 */
{ 4, {  6,  7, 10, -1,  1, 11,  2, -1,  0,  3,  8,  -1, 4,  5,  9, -1}},
/* 10100110 */
{ 2, {  7, 10,  6, -1,  4,  5, 11,  2,  0, -1, -1, -1, -1, -1, -1, -1}},
/* 10100111 */
{ 2, {  3,  8,  4,  5, 11,  2, -1, 10,  6,  7, -1, -1, -1, -1, -1, -1}},
/* 10101000 */
{ 2, {  7,  3,  2,  6, -1,  5,  9,  4, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 10101001 */
{ 2, {  9,  4,  5, -1,  6,  7,  8,  0,  2, -1, -1, -1, -1, -1, -1, -1}},
/* 10101010 */
{ 2, {  3,  2,  6,  7, -1,  1,  0,  4,  5, -1, -1, -1, -1, -1, -1, -1}},
/* 10101011 */
{ 1, {  8,  4,  5,  1,  2,  6,  7, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 10101100 */
{ 2, {  9,  4,  5, -1, 11,  6,  7,  3,  1, -1, -1, -1, -1, -1, -1, -1}},
/* 10101101 */
{ 2, {  1, 11,  6,  7,  8,  0, -1, -1, -1, -1, -1, -1,  9,  4,  5, -1}},
/* 10101110 */
{ 1, {  4,  5, 11,  6,  7,  3,  0, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 10101111 */
{ 1, { 11,  6,  7,  8,  4,  5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 10110000 */
{ 1, {  6,  5,  9,  8, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 10110001 */
{ 1, {  0,  3, 10,  6,  5,  9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 10110010 */
{ 1, {  0,  8, 10,  6,  5,  1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 10110011 */
{ 1, {  3, 10,  6,  5,  1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 10110100 */
{ 2, {  1, 11,  2, -1, 10,  6,  5,  9,  8, 10, -1, -1, -1, -1, -1, -1}},
/* 10110101 */
{ 2, {  0,  3, 10,  6,  5,  9, -1,  1, 11,  2, -1, -1, -1, -1, -1, -1}},
/* 10110110 */
{ 1, {  5, 11,  2,  0,  8, 10,  6, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 10110111 */
{ 1, {  3, 10,  6,  5, 11,  2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 10111000 */
{ 1, {  2,  6,  5,  9,  8,  3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 10111001 */
{ 1, {  6,  5,  9,  0,  2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 10111010 */
{ 1, {  8,  3,  2,  6,  5,  1,  0, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 10111011 */
```

```
{ 1,  { 1,  2,  6,  5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 10111100 */
{ 1, {  6,  5,  9,  8,  3,  1, 11, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 10111101 */
{ 1, {  0,  1, 11,  6,  5,  9, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 10111110 */
{ 2, {  0,  8,  3, -1,  5, 11,  6, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 10111111 */
{ 1, { 11,  6,  5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11000000 */
{ 1, { 10, 11,  5,  7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11000001 */
{ 2, { 10, 11,  5,  7, -1,  8,  0,  3, -1, -1, -1, -1, -1, -1, -1}},
/* 11000010 */
{ 2, {  5,  7, 10, 11, -1,  1,  0,  9, -1, -1, -1, -1, -1, -1, -1}},
/* 11000011 */
{ 2, { 11,  5,  7, 10, -1,  9,  1,  3,  8, -1, -1, -1, -1, -1, -1}},
/* 11000100 */
{ 1, { 10,  2,  1,  5,  7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11000101 */
{ 2, {  0,  3,  8, -1,  1,  5,  7, 10,  2, -1, -1, -1, -1, -1, -1}},
/* 11000110 */
{ 1, {  9,  5,  7, 10,  2,  0, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11000111 */
{ 1, {  7, 10,  2,  3,  8,  9,  5, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11001000 */
{ 1, {  5,  7,  3,  2, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11001001 */
{ 1, {  8,  0,  2, 11,  5,  7, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11001010 */
{ 2, {  9,  1,  0, -1,  5,  7,  3,  2, 11, -1, -1, -1, -1, -1, -1}},
/* 11001011 */
{ 1, {  2, 11,  5,  7,  8,  9,  1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11001100 */
{ 1, {  3,  1,  5,  7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11001101 */
{ 1, {  7,  8,  0,  1,  5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11001110 */
{ 1, {  3,  0,  9,  5,  7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11001111 */
{ 1, {  9,  5,  7,  8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11010000 */
{ 1, {  8, 10, 11,  5,  4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11010001 */
{ 1, {  5,  4,  0,  3, 10, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11010010 */
{ 2, {  0,  9,  1, -1,  8, 10, 11,  5,  4, -1, -1, -1, -1, -1, -1}},
/* 11010011 */
{ 1, {  4,  9,  1,  3, 10, 11,  5, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11010100 */
{ 1, {  5,  4,  8, 10,  2,  1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11010101 */
```

```
{ 1, { 10,  2,  1,  5,  4,  0,  3, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11010110 */
{ 1, {  5,  4,  8, 10,  2,  0,  9, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11010111 */
{ 2, {  9,  5,  4, -1,  2,  3, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11011000 */
{ 1, {  5,  4,  8,  3,  2, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11011001 */
{ 1, {  2, 11,  5,  4,  0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11011010 */
{ 2, {  3,  2, 11,  5,  4,  8, -1,  0,  9,  1, -1, -1, -1, -1, -1, -1}},
/* 11011011 */
{ 1, {  2, 11,  5,  4,  9,  1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11011100 */
{ 1, {  5,  4,  8,  3,  1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11011101 */
{ 1, {  0,  1,  5,  4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11011110 */
{ 1, {  5,  4,  8,  3,  0,  9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11011111 */
{ 1, {  9,  5,  4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11100000 */
{ 1,{  10, 11,  9,  4,  7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11100001 */
{ 2, {  0,  3,  8, -1,  9,  7,  4, 10, 11, -1, -1, -1, -1, -1, -1, -1}},
/* 11100010 */
{ 1, { 10, 11,  1,  0,  4,  7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11100011 */
{ 1, {  4,  7, 10, 11,  1,  3,  8, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11100100 */
{ 1, {  4,  7, 10,  2,  1,  9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11100101 */
{ 2, {  9,  4,  7, 10,  2,  1, -1,  0,  3,  8, -1, -1, -1, -1, -1, -1}},
/* 11100110 */
{ 1, {  4,  7, 10,  2,  0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11100111 */
{ 1, {  4,  7, 10,  2,  3,  8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11101000 */
{ 1, {  2, 11,  9,  4,  7,  3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11101001 */
{ 1, {  7,  8,  0,  2, 11,  9,  4, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11101010 */
{ 1, { 11,  1,  0,  4,  7,  3,  2, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11101011 */
{ 2, {  1,  2, 11, -1,  8,  4,  7, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11101100 */
{ 1, {  1,  9,  4,  7,  3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11101101 */
{ 1, {  1,  9,  4,  7,  8,  0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11101110 */
{ 1, {  4,  7,  3,  0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
/* 11101111 */
```

```
  { 1, {  4,  7,  8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 11110000 */
  { 1, {  9,  8, 10, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 11110001 */
  { 1, {  9,  0,  3, 10, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 11110010 */
  { 1, { 11,  1,  0,  8, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 11110011 */
  { 1, {  3, 10, 11,  1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 11110100 */
  { 1, { 10,  2,  1,  9,  8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 11110101 */
  { 1, {  9,  0,  3, 10,  2,  1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 11110110 */
  { 1, {  0,  8, 10,  2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 11110111 */
  { 1, {  3, 10,  2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 11111000 */
  { 1, {  8,  3,  2, 11,  9,  8, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 11111001 */
  { 1, {  9,  0,  2, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 11111010 */
  { 1, {  8,  3,  2, 11,  1,  0, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 11111011 */
  { 1, {  1,  2, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 11111100 */
  { 1, {  1,  9,  8,  3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 11111101 */
  { 1, {  0,  1,  9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 11111110 */
  { 1, {  0,  8,  3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}},
  /* 11111111 */
  { 0, { -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}}
};
```

APPENDIX D

PUBLICATIONS

1. Y. Livnat, S. Parker, and C.R. Johnson. Fast isosurface extraction methods for large imaging datasets. In *Handbook of Medical Image Processing*, Isaac Bankman, Editor-in-chief, 1999 (to appear).

2. C.R. Johnson, C. Hansen, S.G. Parker, G. Kindlmann, and Y. Livnat. Interactive Computing and Visualization. *IEEE Computer*, 1999 (to appear).

3. S. Parker, M. Parker, Y. Livnat, P. Sloan, C. Hanson, P. Shirley, Interactive Ray Tracing for Volume Visualization IEEE Trans. Vis. Comp. Graphics 1999 (to appear).

4. Y. Livnat, C.D. Hansen. View Dependent Isosurface Extraction, *IEEE Visualization '98*, pp. 175-180, 1998.

5. S. Parker, P. Shirley, Y. Livnat, C. Hanson, P. Sloan. Interactive Ray Tracing for Isosurface Extraction IEEE Proceesings Visualization '98, pp. 233-238, Oct 1998.

6. J.S.Painter, P.Bunge, Y. Livnat. Case Study: Mantle Convection Visualization on the Cray t3D IEEE Proceedings Visualization '96, pp. 409-412, 1996.

7. Y. Livnat, H.W. Shen and C.R. Johnson. A Near Optimal Isosurface Extraction Algorithm Using the Span Space. *IEEE Transactions on Visualization and Computer Graphics*, vol. 2, no. 1, pp. 73-84, 1996.

8. H.W. Shen, C.D. Hansen, Y. Livnat, and C.R. Johnson. Isosurfacing in span space with utmost efficiency (ISSUE) , *IEEE Visualization '96*, pp. 287-294, 1996.

REFERENCES

[1] BAUMGARDNER, J. Three dimensional treatment of convective flow in the earth's mantle. *J. Stat. Phys 39*, 5–6 (1985), 501–511.

[2] BAUMGARDNER, J., AND FREDRICKSON, P. Icosahedral discretization of the two sphere. *Siam J. Numer Anal 22*, 6 (1985), 1107–1115.

[3] BENTLEY, J., AND F., S. D. Analysis of range searches in quad trees. *Info. Proc. Lett. 3*, 6 (1975), 170–173.

[4] BENTLEY, J. L. Multidimensional binary search trees used for associative search. *Communications of the ACM 18*, 9 (1975), 509–516.

[5] BLUM, M., FLOYD, R. W., PRATT, V., RIVEST, R. L., AND TARJAN, R. E. Time bounds for selection. *J. of Computer and System Science 7* (1973), 448–461.

[6] BUNGE, H.-P., RICHARDS, M., AND BAUMGARDNER, J. Effect of depth-dependent viscosity on the planform of mantle convection. *Nature*, 279 (1996), 436–438.

[7] CIGNONI, P., MONTANI, C., PUPPO, E., AND SCOPIGNO, R. Optimal isosurface extraction from irregular volume data. In *Proceedings of IEEE 1996 Symposium on Volume Visualization* (1996), ACM Press.

[8] CLINE, H., W.E., L., AND S., L. Two algorithms for the three-dimensional reconstruction of tomograms. *Medical Physics 15*, 3 (1988), 320–327.

[9] GALLAGHER, R. S. Span filter: An optimization scheme for volume visualization of large finite element models. In *Proceedings of Visualization '91* (1991), IEEE Computer Society Press, Los Alamitos, CA, pp. 68–75.

[10] GILES, M., AND HAIMES, R. Advanced interactive visualization for CFD. *Computing Systems in Engineering 1*, 1 (1990), 51–62.

[11] GREENE, N. Hierarchical polygon tiling with coverage masks. In *Computer Graphics* (August 1996), Annual Conference Series, pp. 65–74.

[12] ITOH, T., AND KOYAMADA, K. Isosurface generation by using extrema graphs. In *Visualization '94* (1994), IEEE Computer Society Press, Los Alamitos, CA, pp. 77–83.

[13] ITOH, T., YAMAGUCHI, Y., AND KOYYAMADA, K. Volume thining for automatic isosurface propagation. In *Visualization '96* (1996), IEEE Computer Society Press,

Los Alamitos, CA, pp. 303–310.

[14] LACROUTE, P., AND LEVOY, M. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *ACM SIGGRAPH 94* (July 24-29 1994), Annual Conference Series, pp. 451–458.

[15] LACROUTE, P. G. Fast volume rendering using shear-warp factorization of the viewing transformation. Tech. rep., Stanford University, September 1995.

[16] LEE, D. T., AND WONG, C. K. Worst-case analysis for region and partial region searches in multidimentional binary search trees and balanced quad trees. *Acta Informatica 9*, 23 (1977), 23–29.

[17] LIVNAT, Y., SHEN, H., AND JOHNSON, C. R. A near optimal isosurface extraction algorithm using the span space. *IEEE Trans. Vis. Comp. Graphics 2*, 1 (1996), 73–84.

[18] LORENSEN, W., AND CLINE, H. E. Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics 21*, 4 (July 1987), 163–169.

[19] MATVEYEV, S. V. Approximation of isosurface in the marching cube: Ambiguity problem. In *Visualization '94* (1994), IEEE Computer Society Press, Los Alamitos, CA, pp. 288–292.

[20] MOLNAR, S., COX, M., ELLSWORTH, D., AND FUCHS, H. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications 14*, 4 (1994), 23–32.

[21] NIELSON, G. M., ET AL. Visualizing and modeling scattered multivariate data. *IEEE Computer Graphics & Applications 11*, 3 (May 1991), 47–54.

[22] R., S. *Algorithms in C++*. Addison–Wesley, Massachusetts, 1992.

[23] SHEKHAR, R., FAYYAD, E., YAGEL, R., AND CORNHILL., J. F. Octree-based decimation of marching cubes surfaces. In *Visualization '96* (1996), IEEE Computer Society Press, Los Alamitos, CA, pp. 335–342.

[24] SHEN, H., HANSEN, C. D., LIVNAT, Y., AND JOHNSON, C. R. Isosurfacing in span space with utmost efficiency (ISSUE). In *Proceedings of Visualization '96* (1996), IEEE Computer Society Press, pp. 287–294.

[25] SHEN, H., AND JOHNSON, C. R. Sweeping simplicies: A fast iso-surface extraction algorithm for unstructured grids. *Proceedings of Visualization '95* (1995), 143–150.

[26] National Library Of Medicine. The Visible Human Project (1986). see http://www.nlm.nih.gov/research/visible/visible_human.html.

[27] WILHELMS, J., AND VAN GELDER, A. Octrees for faster isosurface generation. *Computer Graphics 24*, 5 (November 1990), 57–62.

[28] WILHELMS, J., AND VAN GELDER, A. Topological considerations in isosurface generation. *Computer and Graphics 24*, 5 (1990), 79–86.

[29] WILHELMS, J., AND VAN GELDER, A. Octrees for faster isosurface generation. *ACM Transactions on Graphics 11*, 3 (July 1992), 201–227.

[30] WYVILL, G., MCPHEETERS, C., AND WYVILL, B. Data structure for soft objects. *The Visual Computer 2* (1986), 227–234.