Fast Isosurface Extraction Methods for Large Imaging Data Sets

Yarden Livnat, Steven G. Parker, and Christopher R. Johnson
Center for Scientific Computing and Imaging
Department of Computer Science
University of Utah
Salt Lake City, UT 84112
Email: {yarden,sparker,crj}@cs.utah.edu
WWW: http://www.cs.utah.edu/∼sci

August 7, 1999

Abstract

We present fast isosurface extraction methods for large imaging data sets. Specifically, we discuss three different isosurface extraction techniques, which are appropriate for different visualization scenarios.

The first technique is the Near Optimal IsoSurface Extraction (NOISE) algorithm for rapidly extracting isosurfaces. Using a new representation, termed the *span space*, of the underlying domain, we develop an isosurface extraction algorithm with a worst case complexity of $O(\sqrt{n} + k)$ for the *search phase*, where n is the size of the data set and k is the number of cells in the isosurface. The memory requirement is kept at $O(n)$ while the preprocessing step is $O(n \log n)$.

While algorithms such as NOISE have effectively eliminated the search phase bottleneck, the cost of constructing and rendering the isosurface remains high. Many of today's large imaging data sets contain very large and complex isosurfaces that can easily overwhelm even state-of-the-art graphics hardware. As such, we discuss an output sensitive algorithm that is based on extracting only the visible portion of the isosurface.

The previous two algorithms aim to extract explicit geometric primitives for the underlying isosurface. However, it is sometimes unnecessary (and costly) to extract explicit geometry. We introduce an interactive isosurfacing method for very large imaging data sets with brute-force ray tracing on a conventional (distributed) shared-memory multiprocessor machine. Rather than generate geometry representing the isosurface and render with a z-buffer, for each pixel we trace a ray through a volume and do an analytic isosurface intersection computation. Although this method has a high intrinsic computational cost, its simplicity and scalability facilitate interactive exploration of large data sets on current high-end systems.

1 Introduction

Isosurface extraction is a powerful tool for investigating volumetric scalar fields and has been used extensively in medical imaging ever since the seminal paper by Lorensen and Kline on Marching Cubes [1, 2]. In medical imaging applications, isosurfaces permit the extraction of anatomical structures and tissues.

Since the inception of medical imaging, scanners have continually increased in their resolution capability. This increased image resolution has been instrumental in the use of 3D images for diagnosis, surgical planning, and with the advent of the GE Open MRI system, for surgery itself. Such increased resolution, however, has caused researchers to look beyond Marching Cubes in order to obtain near interactive rates for such large-scale imaging data sets. As such, there has been a renewed interested in creating isosurface algorithms that have optimal complexity and can take advantage of advanced computer architectures.

In this chapter, we discuss three techniques developed by the authors (and colleagues) for fast isosurface extraction for large-scale imaging data sets. The first technique is the Near Optimal IsoSurface Extraction (NOISE) algorithm for rapidly extracting isosurfaces. Using a new representation, termed the *span space*, of the underlying domain, we develop an isosurface extraction algorithm with a worst case complexity of $O(\sqrt{n} + k)$ for the *search phase*, where n is the size of the data set and k is the number of cells in the isosurface. The memory requirement is kept at $O(n)$ while the preprocessing step is $O(n \log n)$. We note that we can utilize the span space representation as a tool for comparing other isosurface extraction methods on structured (and unstructred) grids.

While algorithms such as NOISE have effectively eliminated the search phase bottleneck, the cost of constructing and rendering the isosurface remains high. Many of today's large imaging data sets contain very large and complex isosurfaces that can easily overwhelm even state-of-the-art graphics hardware. As such, we discuss an output sensitive algorithm that is based on extracting only the visible portion of the isosurface. This output sesitive algorithm is based on extracting only the visible portion of the isosurface. The visibility tests are done in two phases. First, coarse visibility tests are performed in software to determine the visible cells. These tests are based on hierarchical tiles and shear-warp factorization. The second phase resolves the visible portions of the extracted triangles and is accomplished by the graphics hardware.

When an isosurface is extracted from a large imaging data set by the previous two techniques (or by other Marching Cube-like methods) an explicit polygonal representation for the surface is created. This surface is subsequently rendered with attached graphics hardware accelerators. Such explicit geometry-based isosurface extraction methods can generate an extraordinary number of polygons, which take time to construct and to render. For very large (i.e., greater than several million polygons) surfaces the isosurface extraction and rendering times limit the interactivity.

In the third technique we describe, we generate images of isosurfaces directly with no intermediate surface representation through the use of ray tracing. Using parallel processing and incorporating simple optimizations enables interactive rendering (i.e. 10 frames per second) of the 1GByte full resolution Visible Woman CT data set on an SGI Origin 2000.

2 Accelerated Search

2.1 The Span Space

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

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

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

Definition 1 (Isosurface Extraction) *Given a set of samples $\mathcal{D}$ over a field $\varphi : \mathbf{G} \to \mathbf{V}$, and given a single value $v \in \mathbf{V}$, find,*

$$S = \{g_i\} \quad g_i \in \mathbf{G} \quad \textit{such that,} \quad \varphi(g_i) = v. \tag{1}$$

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

Approximating an isosurface, S, as a global solution to Eq. 1 can be a difficult task because of the sheer size, d, of a large imaging data set.

In thinking about imaging data sets, one can decompose the geometric space, $\mathbf{G}$, into a set of polyhedral cells (voxels), 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 [1, 2]. 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 $\mathbf{G}$. Because this representation does not require explicit adjacency information between cells, isosurface extraction methods on structured grids conduct searches over the geometric space, $\mathbf{G}$. The problem as stated by these methods [3, 4, 5, 6] is defined as follows:

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

Another approach is to forgo the geometric location of a cell and examine only the values at the cell vertices. The advantage of this 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 that are based on this value space approach [7, 8, 9] view the isosurface extraction problem in the following way:

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

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

find the subset I_s such that,

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

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

The method presented in this section addresses the search over the value space. Our approach is not to view the problem as a search over intervals in $\mathbf{V}$ but rather as a search over points in $\mathbf{V}^2$. We start with an augmented definition of the search space.

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

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

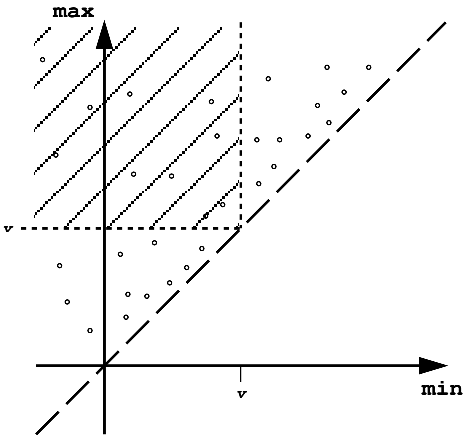

Figure 1: Search over the span space.

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.

Though conceptually not much different than the interval space, the span space will, nevertheless, lead to a simple and near optimal search algorithm.

A key point is that points in 2D exhibit no explicit relations between themselves, while 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 we shall show 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* $v \in \mathbf{V}$*, find the subset* $P_s \subseteq P$*, such that*

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

We note that $\forall (x_i, y_i) \in P_s, \quad x_i \leq y_i$ and thus the associated points will lie on or above the line $y_i = x_i$. A geometric perspective of the span search is given in Fig. 1.

2.2 The NOISE Algorithm

A common obstacle for all the interval methods was that the intervals were ordered according to *either* their maximum or their minimum value. The sweeping simplicies algorithm [9] attempted to tackle this issue by maintaining two lists of the intervals, ordered by the maximum and minimum values. What was missing, however, was 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 employ a kd-tree as a means for simultaneously ordering the cells according to their maximum and minimum values.

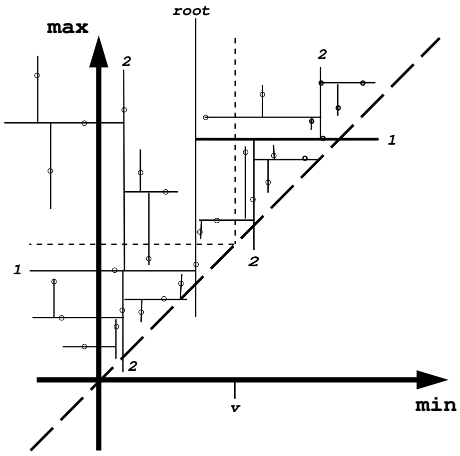

Figure 2: Kd Tree

2.2.1 Kd-Trees

Kd-trees were designed by Bentley in 1975 [10] as a data structure for efficient associative searching. In essence, kd-trees are a multi-dimensional version of binary search trees. Each node in the tree holds one of the data values and has two sub-trees as children. The sub-trees are constructed so that all the nodes in one sub-tree, the *left* one for example, hold values that are less than the parent node's value, while the values in the *right* sub-tree are greater than the parent node's value.

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.

2.2.2 Search over the Span Space Using Kd-Tree

Given a data set, a kd-tree that contains pointers to the data cells is constructed. Using this kd-tree as an index to the data set, the algorithm can now rapidly answer isosurface queries. Fig. 2 depicts a typical decomposition of a span space by a kd-tree.

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 one dimension and store it at the root node. The data is then partitioned according to the median and recursively stored in the two sub-trees. The partition at each level alternates between the min and max coordinates.

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.* [11], 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. We then use these marks to construct the two sub groups, 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. 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 data sets.

A simple (and we think elegant) solution is to use a Quicksort-based selection [12]. 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.

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 $O(n)$.

Query

Given an iso-value, v, we seek to locate all the points in Fig. 1 that are to the *left* of the vertical line at v and are *above* the horizontal line at v. We note that we do not need to locate points that are *on* these horizontal or vertical lines if we assume non-degenerate cells, for which minimum or maximum values are not unique. We will remove this restriction later.

The kd-tree is traversed recursively when the iso-value is compared to the value stored at the current node alternating between the minimum and maximum values at each level. If the node is to the left (above) of the iso-value line, then only the left (right) sub-tree should be traversed. Otherwise, *both* sub-trees should be traversed recursively. For efficiency we define two search routines, *SearchMinMax* and *SearchMaxMin*. The dimension we currently checking is the first named, and the dimension we still need to search is named second. 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 straight-forward. Indeed, the analysis of the worst case was developed by Lee and Wong [13] only several years after Bentley introduced kd-trees. 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 $O(\sqrt{n}+k)$ [12, 14]. 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.

Degenerate Cells

A degenerate cell is defined as a cell having more then one vertex with a minimum or maximum value. When a given iso-value 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. We note that in all the data sets we tested there was not a single case of such a degeneracy.

2.3 Optimization

The algorithm presented in the previous section is not optimal with regards to both the memory requirement and search time. We now present several strategies to optimize the algorithm.

2.3.1 Pointerless Kd-Tree

A kd-tree node, as presented previously, must maintain links to its two sub-trees. This introduces a high cost in terms of memory requirements. To overcome this, we note that in our 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 sub-trees. We can, therefore, represent 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 sub-tree and the last $(n-1)/2$ nodes the right sub-tree.

When we use a pointerless kd-tree, the memory requirements for our kd-tree, per node, reduce to two real numbers, for minimum and maximum values, and one pointer back to the original cell for later usage. Considering that each cell, for a 3D 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 preprocess 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 preprocess 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 data sets.

2.3.2 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 iso-value is less then the minimum value of the node, then we know we can trim the right sub-tree. Consider the case where the iso-value is greater then the node's minimum coordinate. In this case, we need to traverse *both* sub-trees. We have no new information with respect to the search in the right sub-tree, but, for the search in the left sub-tree we *know* that the minimum condition is satisfied. We can take advantage of this fact by skipping over the odd levels from that point on. To achieve this, we define two new routines, *search-min* and *search-max*. 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 sub-tree. If the iso-value is greater than the node's minimum, we recursively traverse the right sub-tree, but with regard to the left sub-tree, 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 Sec 2.3.1, any sub-tree is represented as a *contiguous* block of the tree's nodes. Collecting all the nodes of a sub-tree requires only sequentially traversing this contiguous block.

We remark that with the current performance of the algorithm and current available hardware, the bottle neck is no longer in finding the isosurface or even computing it, but rather in the actual time it takes to display it. As such, we look next at a new view dependent algorithm that constructs and displays only the part of the isosurface that is visible to the user.

2.4 Other Span Space Algorithms

The span space representation has recently been used by Cignoni et al. [15] to reduce the complexity of the search phase to $O(\log n + k)$ at the expense of higher memory requirements. Shen et al. [16] used a lattice decomposition of the span space for a parallel version on a massive parallel machine.

3 View Dependent Algorithm

The proposed method is based on the observation that isosurfaces extracted from very large data sets often exhibit high depth complexity for two reasons. First, since the data sets are very large, the projection of individual cells tend to be sub-pixel. This leads to a large number of polygons, possibly non-overlapping, projecting onto individual pixels. Secondly, for some data sets, large sections of an isosurface are internal and thus, are occluded by other sections of the isosurface, as illustrated in Figure 3. These internal sections, common in medical data sets, 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 4 depicts a two dimensional scenario. In a view dependent methods only the solid lines are extracted whereas in non view dependent isocontouring both solid and dotted are extracted.

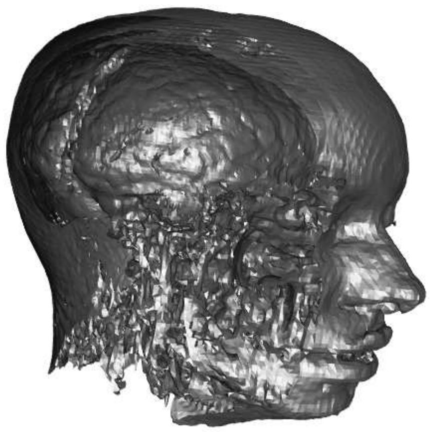

Figure 3: A slice through an isosurface reveal the internal sections which can not contribute to the final image.

The proposed algorithm, which is based on a hierarchical traversal of the data and a marching cubes triangulation, exploit 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 5. First, we augment Wilhelms' and Van Gelder's algorithm [4] by traversing down the octree in a front-to-back order in addition to pruning empty sub-trees based on the min-max values stored at the octree nodes. The second step employs coarse software visibility tests for each [meta-] cell which intersect the isosurface. The aim of these tests is to determine whether the [meta-] cell is hidden from the view point by previously extracted sections

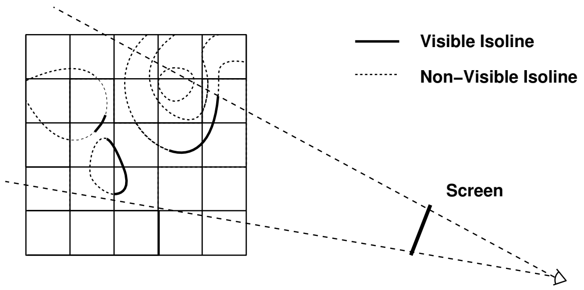

Figure 4: A two-dimensional scenario.

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 dataflow diagram is depicted in Figure 6.

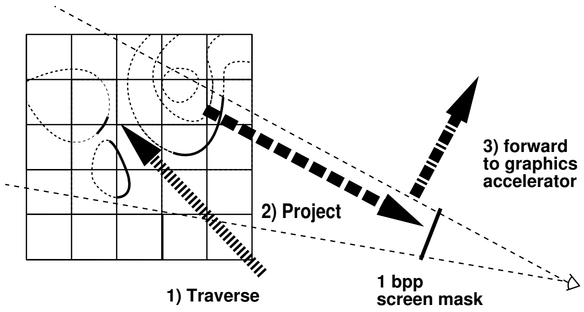

Figure 5: The three step algorithm.

3.1 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.

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.

We 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 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.

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.

In the next sections, we address these costs in two ways. First, we employ a hierarchical tiling for the virtual screen. Secondly, to reduce the cost of the projection we use a variation of the shear-warp

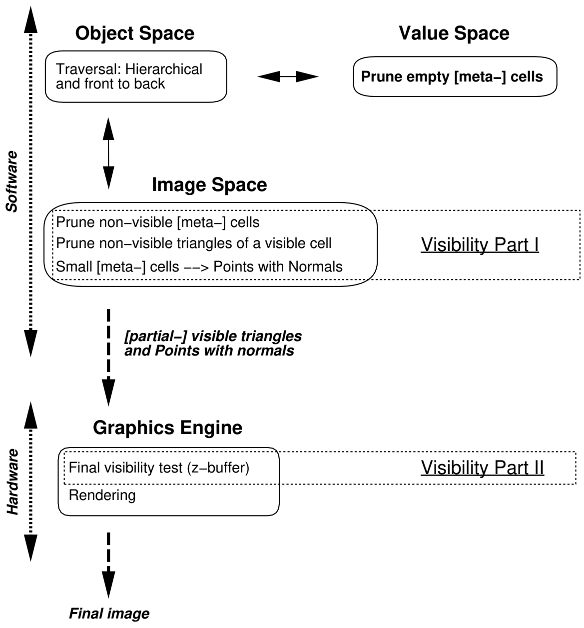

Figure 6: The algorithm data flow.

factorization.

3.2 Image Space Culling

We employ hierarchical tiles [17] as a mean 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.

3.2.1 Hierarchical Tiles

A coverage map (a tile) is a rectangular bitmap (we use 8x8) in which each bit represents a pixel in the final image. The algorithms 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 intersect the tile boarder, Figure 7. The coverage pattern of a convex polygon for a particular tile of the image is computed by combining 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 cover, Figure 8. We refer the reader to the work by Greene [17] 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.

Rendering a polygon amounts to computing the coverage map of the polygon for each tile in the image and isolating only those pixels that are covered by the polygon but where not already covered. 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 sub-tiles in which the polygon might be visible are checked recursively, leading to a logarithmic search.

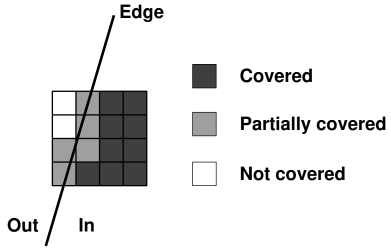

Figure 7: An edge tile.

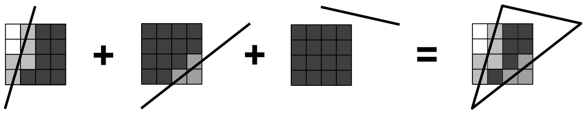

Figure 8: A triangle tile coverage map.

3.2.2 Hierarchical Visibility Mask

Our implementation differs from the one proposed by Greene in that we do not actually render the visible portion of a visible 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 render them.

One should note that it is not possible to determine *a priori* the 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 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 non-visible portions will be later removed by the graphics accelerator.

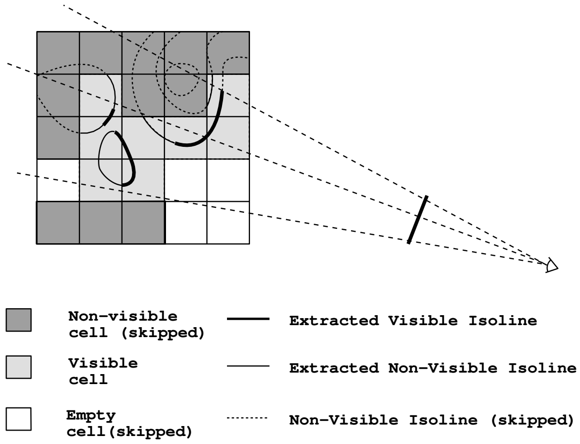

Figure 9: Cells and isolines visibility.

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 with an associated normal to the graphics hardware, similar to the dividing cubes method [18]. 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.

3.3 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 division 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, the non-linearity of the perspective transformation prohibits the use of pre-computed transformation table. To accelerate this critical step, we take advantage of the shear-warp factorization of the viewing transformation.

3.3.1 Shear-Warp Factorization

In 1994, Lacroute [19, 20] 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 is first projected into a sheared object space that is used to create an intermediate, albeit warped, image. Once this image is complete a warping transformation is applied to create the correct final image. Figure 10 illustrates the shear-warp transformation for both orthographic and perspective projections.

The advantage of this method is that the intermediate image is aligned with one of the data set faces. This alignment enables the use of a parallel projection of the 3D data set. The warp stage is then applied to a 2D image rather than to each data point.

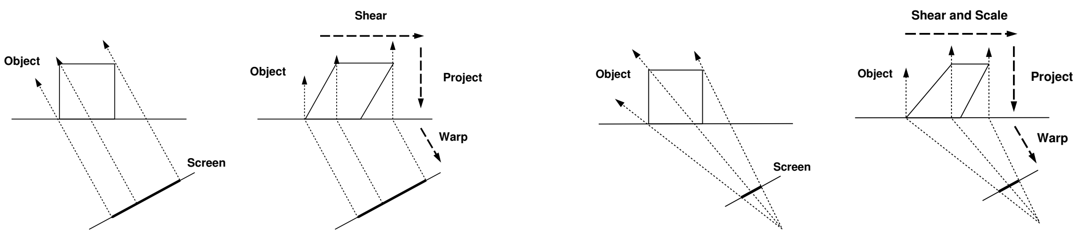

Figure 10: Shear-warp in orthographic and perspective projections.

3.3.2 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 11. In other words, any point in the data set 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. It is therefore sufficient to perform the visibility tests on the warped projection plane. The advantage of this approach is two fold. 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 pre-compute them once for each new view point.

Let $[X, Y, Z]$ be the coordinate system of the data set and let $[s_x, s_y, s_z]$ 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 pre-compute the shear and scale coefficients,

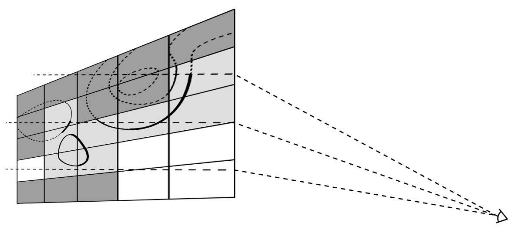

Figure 11: Warped space.

foreach Z

$s = Z * s_z/(Z * s_z - eye_z)$

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

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

$shear_x[Z] = s * eye_x$

$shear_y[Z] = s * eye_y$

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

$Project(p) \equiv$

$x = p_x * scale_x[p_z] + shear_x[p_z]$

$y = p_y * scale_y[p_z] + shear_y[p_z]$

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 pre-computed 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 that a vertex projection is,

$Project(p) \equiv$

$s = p_z/(z - eye_z)$

$x = p_x + s * (eye_x - p_x)$

$y = p_y + s * (eye_y - p_y)$

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

4 Real-time Ray Tracing

Many applications, including most medical imaging techniques, generate scalar fields $\rho(x, y, z)$ which can be viewed by displaying *isosurfaces* where $\rho(x, y, z) = \rho_{\text{iso}}$. Ideally, the value for ρ_{iso} is interactively controlled by the user. When the scalar field is stored as a structured set of point samples, the most common technique for generating a given isosurface is to create an explicit polygonal representation for the surface using a technique such as *Marching Cubes* [1, 2]. This surface is subsequently rendered with attached graphics hardware accelerators such as the SGI Infinite Reality. Marching Cubes can generate an extraordinary number of polygons, which take time to construct and to render. For very large (i.e., greater than several million polygons) surfaces the isosurface extraction and rendering times limit the interactivity. In this paper, we generate images of isosurfaces directly with no intermediate surface representation through the use of ray tracing. Ray tracing for isosurfaces has been used in the past (e.g. [21, 22, 23]), but we apply it to very large data sets in an interactive setting for the first time.

The basic ray-isosurface intersection method used in this paper is shown in Figure 12. Conventional wisdom holds that ray tracing is too slow to be competitive with hardware z-buffers. However, when rendering a surface from a sufficiently large data set, ray tracing should become competitive as its low time complexity overcomes its large time constant [24]. The same arguments apply to the isosurfacing problem. Suppose we have an $n \times n \times n$ rectilinear volume which for a given isosurface value has $O(n^2)$ polygons generated using Marching Cubes. Given intelligent preprocessing, the rendering time will be $O(n^2)$. Since it is hard to improve performance using multiple graphics engines, this seems a hard limit when using commercially available graphics accelerators unless a large fraction of the polygons are not visible [25]. If a ray tracing algorithm is used to traverse the volume until a surface is reached, we would expect each ray to do $O(n)$ work. If the rays are traced on p processors, then we expect the runtime for an isosurface image to be $O(n/p)$, albeit with a very large time constant and a limit that p is significantly lower than the number of pixels. For sufficiently large n, ray tracing will be faster than a z-buffer algorithm for generating and rendering isosurfaces. The question is whether it can occur on an n that occurs in practice (e.g., $n = 500$ to $n = 1000$) with a p that exists on a real machine (e.g., $p = 8$ to $p = 128$). The following demonstrates that with a few optimizations, ray tracing is *already* attractive for at least some isosurface applications, including high resolution medical imaging applications.

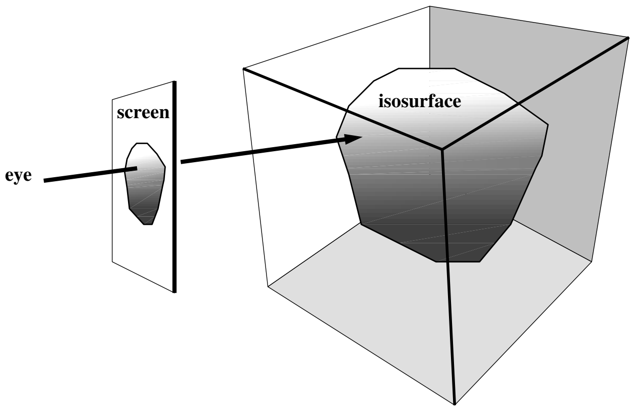

Figure 12: *A ray is intersected directly with the isosurface. No explicit surface is computed.*

Ray tracing has been used for volume visualization in many works (e.g., [26, 27, 28]). Typically, the ray tracing of a pixel is a kernel operation that could take place within any conventional ray tracing system. In this section we review how ray tracers are used in visualization, and how they are implemented efficiently at a systems level.

The algorithm has three phases: traversing a ray through cells which do not contain an isosurface, analytically computing the isosurface when intersecting a voxel containing the isosurface, shading the resulting intersection point. This process is repeated for each pixel on the screen. Since each ray is independent, parallelization is straightforward. An additional benefit is that adding incremental features to the rendering has only incremental cost. For example, if one is visualizing multiple isosurfaces with some of them rendered transparently, the correct compositing order is guaranteed since we traverse the volume in a front-to-back order along the rays. Additional shading techniques, such as shadows and specular reflection, can easily be incorporated for enhanced visual cues. Another benefit is the ability to exploit texture maps which are much larger than texture memory (typically up to 64 MBytes).

In the following subsections, we describe the details of our technique. We first address the ray-isosurface intersection followed by a description of various optimizations we have performed to achieve the interactive rates.

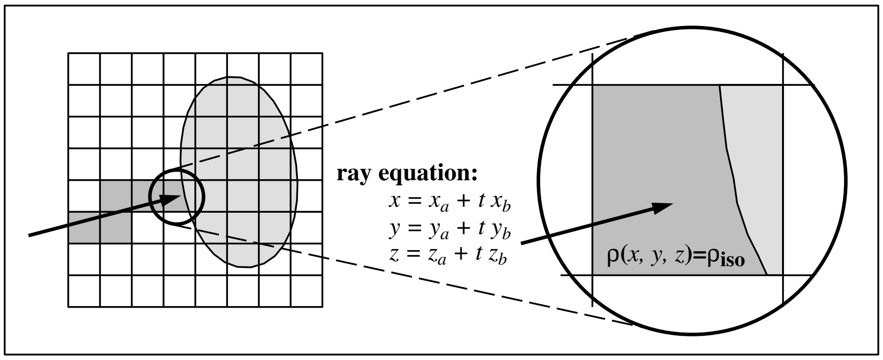

Figure 13: *The ray traverses each cell (left), and when a cell is encountered that has an isosurface in it (right), an analytic ray-isosurface intersection computation is performed.*

4.1 Ray-Isosurface Intersection

If we assume a regular volume with even grid point spacing arranged in a rectilinear array, then the ray-isosurface intersection is straightforward. Analagous simple schemes exist for intersection of tetrahedral cells, but the traversal of such grids is left for future work. This work will focus on rectilinear data.

To find an intersection (Figure 13), the ray $\vec{\mathbf{a}} + t\vec{\mathbf{b}}$ traverses cells in the volume checking each cell to see if its data range bounds an isovalue. If it does, an analytic computation is performed to solve for the ray parameter t at the intersection with the isosurface:

$$\rho(x_a + tx_b, y_a + ty_b, z_a + tz_b) - \rho_{\text{iso}} = 0.$$

When approximating ρ with a trilinear interpolation between discrete grid points, this equation will expand to a cubic polynomial in t. This cubic can then be solved in closed form to find the intersections of the ray with the isosurface in that cell. Only the roots of the polynomial which are contained in the cell are examined. There may be multiple roots, corresponding to multiple intersection points. In this case, the smallest t (closest to the eye) is used. There may also be no roots of the polynomial, in which case the ray misses the isosurface in the cell. The details of this intersection computation are given in [29].

4.2 Optimizations

For the traversal of rays through the data, we use the incremental method described by Amanatides and Woo [30]. We found that traversing the cells is the computational bottleneck for large data sets, so we include optimizations to accelerate performance.

The first optimization is to improve data cache locality by organizing the volume into “bricks” that are analogous to the use of image tiles in image-processing software and other volume rendering programs [31] (Figure 14). The details of our method for efficiently indexing cells is discussed in [29].

The second is to use a multi-level spatial hierarchy to accelerate the traversal of empty cells as is shown in Figure 14. Cells are grouped divided into equal portions, and then a “macrocell” is created which contains the minimum and maximum data value for it’s children cells. This is a common variant of standard ray-grid techniques [32] and the use of minimum/maximum caching has been shown to be useful [33, 3, 4]. The ray-isosurface traversal algorithm examines the min and max at each macrocell before deciding whether to recursively examine a deeper level or to proceed to the next cell. The average complexity of this search will be $O(\sqrt[3]{n})$ for a three level hierarchy. While the worst case complexity is still $O(n)$, it is difficult to imagine an isosurface occuring in practice approaching this worst case. Using a deeper hierarchy can theoretically reduce the average case complexity slightly, but also dramatically increases the storage cost

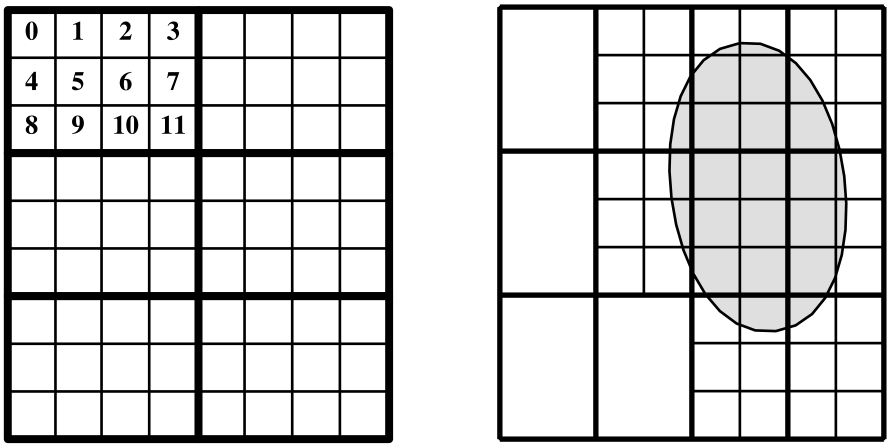

Figure 14: The ray-tracing algorithm uses two different hierarchies simultaneously. On the left, Cells can be organized into "tiles" or "bricks" in memory to improve locality. The numbers in the first brick represent layout in memory. Neither the number of atomic voxels nor the number of bricks need be a power of two. On the right is the hierarchy used to efficiently skip over empty cells. With a two-level hierarchy, rays can skip empty space by traversing larger cells. A three-level hierarchy is used for the Visible Woman example.

of intermediate levels. We have experimented with modifying the number of levels in the hierarchy and empirically determined that a tri-level hierarchy (one top-level cell, two intermediate macrocell levels, and the data cells) is highly efficient. This optimum may be data dependent and is modifiable at program startup. Using a tri-level hierarchy, the storage overhead is negligible ($< 0.5\%$ of the data size). The cell sizes used in the hierarchy are independent of the brick sizes used for cache locality in the first optimization.

Since one cannot predict *a priori* the complexity of extracting an isosurface from a particular screen pixel, we employ a dynamic load balancing scheme to ensure high processor utilization over a wide range of views. The screen space is first split into tiles in the image space. In our implementation, tiles are 32 pixels wide by 4 pixels high. The width of the tile (128 bytes) ensures that tiles will not share a cache line with neighboring tiles. At the beginning of a frame, each tile becomes an assignment in a queue. Each processor pulls a range of assignments from the queue, performs the assigned work, and then returns to the queue for more work. The assignments, which are initially doled out in large chunks, get smaller and smaller as the frame nears completion. The large granularity in the beginning reduces contention for a large portion of the image, and the smaller granularity near the end helps to balance the load efficiently [34].

4.3 Real-time Ray Tracing Results

Table 1 shows the scalability of the algorithm from 1 to 128 processors. View 2 uses a zoomed out viewpoint with approximately 75% pixel coverage whereas view 1 has nearly 100% pixel coverage. We chose to examine both cases since view 2 achieves higher frame rates. The higher frame rates cause less parallel efficiency due to synchronization and load balancing. Of course, maximum interaction is obtained with 128 processors, but reasonable interaction can be achieved with fewer processors. If a smaller number of processors were available, one could reduce the image size in order to restore the interactive rates. Efficiencies are 91% and 80% for view 1 and 2 respectively on 128 processors. The reduced efficiency with larger numbers of processors (> 64) can be explained by load imbalances and the time required to synchronize processors at the required frame rate. The efficiencies would be higher for a larger image.

Table 2 shows the improvements which were obtained through the data bricking and spatial hierarchy optimizations.

Table 1: Scalability results for ray tracing the bone isosurface in the visible human. A 512x512 image was generated using a single view of the bone isosurface.

	View 1		View 2	
# cpus	FPS	speedup	FPS	speedup
1	0.18	1.0	0.39	1.0
2	0.36	2.0	0.79	2.0
4	0.72	4.0	1.58	4.1
8	1.44	8.0	3.16	8.1
12	2.17	12.1	4.73	12.1
16	2.89	16.1	6.31	16.2
24	4.33	24.1	9.47	24.3
32	5.55	30.8	11.34	29.1
48	8.50	47.2	16.96	43.5
64	10.40	57.8	22.14	56.8
96	16.10	89.4	33.34	85.5
128	20.49	113.8	39.98	102.5

Table 2: Times in seconds for optimizations for ray tracing the visible human. A 512x512 image was generated on 16 processors using a single view of an isosurface.

View	Initial	Bricking	Hierarchy+Bricking
skin: front	1.41	1.27	0.53
bone: front	2.35	2.07	0.52
bone: close	3.61	3.52	0.76
bone: from feet	26.1	5.8	0.62

5 Example Applications

In this last section, we give examples of the NOISE, View Dependent, and Real-time Ray Tracing algorithms given in the three previous sections. The examples we chose are from large medical imaging data sets, as well as from a large-scale geoscience imaging data set. Further examples of each of the algorithms along with detailed performance analyses can be found in our recent papers on isosurface extraction [16, 35, 25, 29, 36].

5.1 NOISE Examples

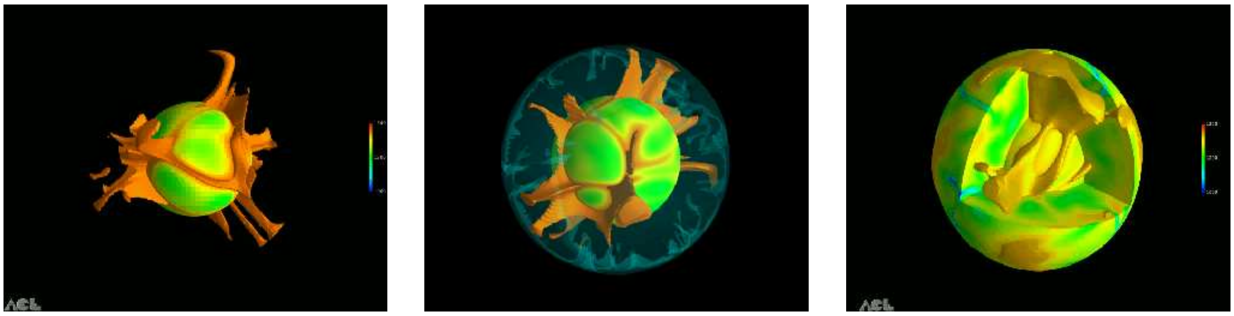

Figure 15: Mantle convection modeling which was done on a Cray T3D at the Advanced Computing Lab at Los Alamos National Lab using over 20 million elements. The left image shows a single, hot, isosurface while the second adds a cold (blue) transparent isosurface. transparency. The third image shows a slice through the mantle with a single isosurface.

5.2 View Dependent Examples

Figure 16: Full vs. View dependent isosurface extraction. The isosurfaces were computed based on a user point of view which was above and behind the skull. These images illustrate the large portions of the isosurface which the view dependent algorithm was able to avoid.

5.3 Real-time Ray Tracing Examples

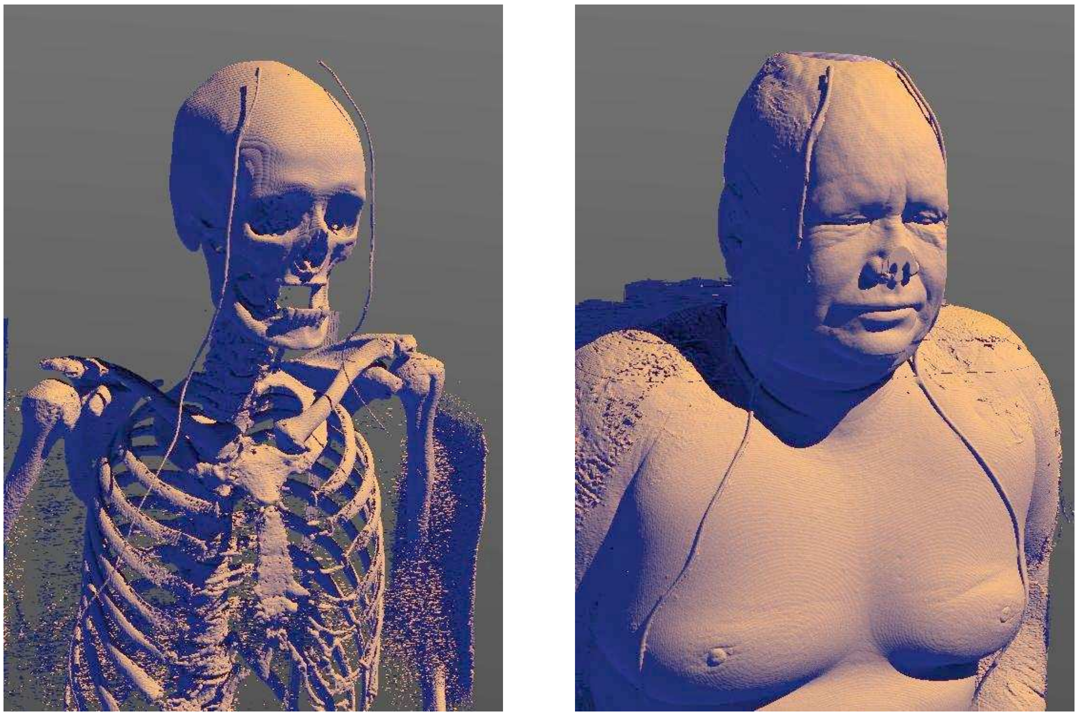

Figure 17: Ray tracings of the bone and skin isosurfaces of the Visible Woman.

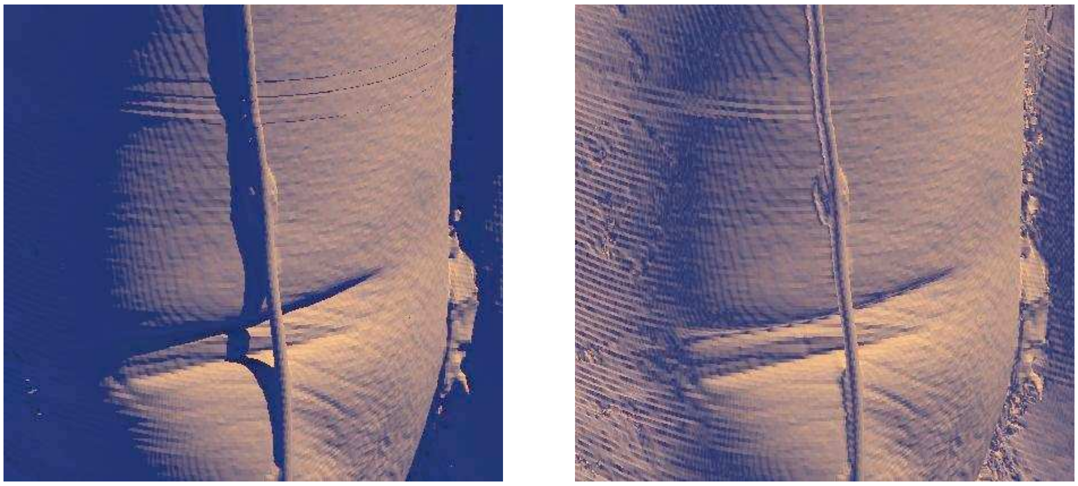

Figure 18: A ray tracing with and without shadows.

6 Acknowledgements

This work was supported in part by awards from the Department of Energy and the National Science Foundation. The authors would like to thank Peter Shirley, Chuck Hansen, Han-Wei Shen, and Peter-Pike Sloan for their significant contributions to the research presented in this chapter. The Visible Woman data set was obtained from the Visible Human Project of the National Library of Medicine.

7 Authors

Yarden Livnat is a Research Associate at the Department of Computer Science at the University of Utah. working with the Scientific Computing and Imaging Research Group. Yarden received a B.Sc. in computer science in 1982 from Ben Gurion University Israel and an M.Sc. cum laude in computer science from the Hebrew University, Israel in 1991. He will receive his Ph.D from the University of Utah in 1999. His research interests include computational geometry, scientific computation and visualization and computer generated holograms.

Steven Parker is a research scientist in the Department of Computer Science at the University of Utah. His research focuses on problem solving environments, which tie together scientific computing, scientific visualization, and computer graphics. He is the principal architect of the SCIRun Software System, which formed the core of his Ph.D. dissertation. He was a recipient of the Computational Science Graduate Fellowship from the Department of Energy. He received a B.S. in Electrical Engineering from the University of Oklahoma in 1992.

Christopher Johnson is an Associate Professor in the Department of Computer Science at the University of Utah. His research interests are in the area of scientific computing. Particular interests include inverse and imaging problems, adaptive methods for partial differential equations, problem solving environments, large scale computational problems in medicine, and scientific visualization. Professor Johnson was awarded a FIRST Award from the NIH in 1992, a National Young Investigator (NYI) Award from the NSF in 1994, and the Presidential Faculty Fellow (PFF) Award in 1995. He directs the Center for Scientific Computing and Imaging (SCI) at the University of Utah (see the SCI web page at http://www.cs.utah.edu/ ~sci for more information). He can be reached via email at crj@cs.utah.edu.

References

[1] B. Wyvill G. Wyvill, C. McPheeters. Data structures for soft objects. *The Visual Computer*, 2:227–234, 1986.

[2] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics*, 21(4):163–169, July 1987. ACM Siggraph '87 Conference Proceedings.

[3] J. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation. *Computer Graphics*, 24(5):57–62, November 1990.

[4] J. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, July 1992.

[5] T. Itoh and K. Koyamada. Isosurface generation by using extrema graphs. In *Visualization '94*, pages 77–83. IEEE Computer Society Press, Los Alamitos, CA, 1994.

[6] T. Itoh, Y. Yamaguchi, and K. Koyyamada. Volume thining for automatic isosurface propagation. In *Visualization '96*, pages 303–310. IEEE Computer Society Press, Los Alamitos, CA, 1996.

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

[8] M. Giles and R. Haimes. Advanced interactive visualization for CFD. *Computing Systems in Engineering*, 1(1):51–62, 1990.

[9] H. Shen and C. R. Johnson. Sweeping simplicies: A fast iso-surface extraction algorithm for unstructured grids. *Proceedings of Visualization '95*, 1995. (to appear).

[10] J. L. Bentley. Multidimentional binary search trees used for associative search. *Communications of the ACM*, 18(9):509–516, 1975.

[11] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *J. of Computer and System Science*, 7:448–461, 1973.

[12] Sedgewick R. *Algorithms in C++*. Addison–Wesley, Massachusetts, 1992.

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

[14] J. L. Bentley and Stanat D. F. Analysis of range searches in quad trees. *Info. Proc. Lett.*, 3(6):170–173, 1975.

[15] P. Cignoni, C. Montani, E. Puppo, and R. Scopigno. Optimal isosurface extraction from irregular volume data. In *Proceedings of IEEE 1996 Symposium on Volume Visualization*. ACM Press, 1996.

[16] H. Shen, C. D. Hansen, Y. Livnat, and C. R. Johnson. Isosurfacing in span space with utmost efficiency (issue). In *Proceedings of Visualization '96*, pages 287–294. IEEE Computer Society Press, 1996.

[17] Ned Green. Hierarchical polygon tiling with coverage masks. In *Computer Graphics*, Annual Conference Series, pages 65–74, August 1996.

[18] H.E. Cline, Lorensen W.E., and Ludke S. Two algorithms for the three-dimensional reconstruction of tomograms. *Medical Physics*, 15(3):320–327, 1988.

[19] Philippe Lacroute and Mark Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *Computer Graphics*, Annual Conference Series, pages 451–458. ACM SIGGRAPH, 1994.

[20] Philippe G. Lacroute. Fast volume rendering using shear-warp factorization of the viewing transformation. Technical report, Stanford university, September 1995.

[21] Chyi-Cheng Lin and Yu-Tai Ching. An efficient volume-rendering algorithm with an analytic approach. *The Visual Computer*, 12(10):515–526, 1996.

[22] Stephen Marschner and Richard Lobb. An evaluation of reconstruction filters for volume rendering. In *Proceedings of Visualization '94*, pages 100–107, October 1994.

[23] Milos Sramek. Fast surface rendering from raster data by voxel traversal using chessboard distance. In *Proceedings of Visualization '94*, pages 188–195, October 1994.

[24] James T. Kajiya. An overview and comparison of rendering methods. *A Consumer's and Developer's Guide to Image Synthesis*, pages 259–263, 1988. ACM Siggraph '88 Course 12 Notes.

[25] Y. Livnat and C.D. Hansen. View dependent isosurface extraction. In *IEEE Visualization '98*, pages 175–180. IEEE Computer Society, Oct. 1998.

[26] Mark Levoy. Display of surfaces from volume data. *IEEE Computer Graphics & Applications*, 8(3):29–37, 1988.

[27] Paolo Sabella. A rendering algorithm for visualizing 3d scalar fields. *Computer Graphics*, 22(4):51–58, July 1988. ACM Siggraph '88 Conference Proceedings.

[28] Craig Upson and Micheal Keeler. V-buffer: Visible volume rendering. *Computer Graphics*, 22(4):59–64, July 1988. ACM Siggraph '88 Conference Proceedings.

[29] Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter-Pike Sloan. Interactive ray tracing for isosurface rendering. In *Proceedings of Visualization '98*, October 1998.

[30] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *Eurographics '87*, 1987.

[31] Michael B. Cox and David Ellsworth. Application-controlled demand paging for Out-of-Core visualization. In *Proceedings of Visualization '97*, pages 235–244, October 1997.

[32] James Arvo and David Kirk. A survey of ray tracing acceleration techniques. In Andrew S. Glassner, editor, *An Introduction to Ray Tracing*. Academic Press, San Diego, CA, 1989.

[33] Al Globus. Octree optimization. Technical Report RNR-90-011, NASA Ames Research Center, July 1990.

[34] Scott Whitman. *Multiprocessor Methods for Computer Graphics Rendering*. Jones and Bartlett Publishers, 1992.

[35] Y Livnat, H. Shen, and C. R. Johnson. A near optimal isosurface extraction algorithm using the span space. *IEEE Trans. Vis. Comp. Graphics*, 2(1):73–84, 1996.

[36] J. Painter, H.P. Bunge, and Y. Livnat. Case study: Mantle convection visualization on the Cray T3D. In *Proceedings of IEEE Visualization '96*. IEEE Press, Oct. 1996.