Interactive Ray Tracing for Volume Visualization

Steven Parker, *Member, IEEE Computer Society*,
Michael Parker, *Student Member, IEEE Computer Society*,
Yarden Livnat, *Student Member, IEEE Computer Society*,
Peter-Pike Sloan, *Member, IEEE Computer Society*,
Charles Hansen, *Member, IEEE Computer Society*, and Peter Shirley

Abstract—We present a brute-force ray tracing system for interactive volume visualization. The system runs on a conventional (distributed) shared-memory multiprocessor machine. For each pixel we trace a ray through a volume to compute the color for that pixel. Although this method has high intrinsic computational cost, its simplicity and scalability make it ideal for large datasets on current high-end parallel systems. To gain efficiency several optimizations are used including a volume bricking scheme and a shallow data hierarchy. These optimizations are used in three separate visualization algorithms: isosurfacing of rectilinear data, isosurfacing of unstructured data, and maximum-intensity projection on rectilinear data. The system runs interactively (i.e., several frames per second) on an SGI Reality Monster. The graphics capabilities of the Reality Monster are used only for display of the final color image.

Index Terms—Ray tracing, visualization, isosurface, maximum-intensity projection.

1 Introduction

Many applications generate scalar fields $\rho(x, y, z)$ which can be visualized by a variety of methods. These fields are often defined by a set of point samples and an interpolation rule. The point samples are typically in either a rectilinear grid, a curvilinear grid, or an unstructured grid (simplical complex). The two main visualization techniques used on such fields are to display *isosurfaces* where $\rho(x, y, z) = \rho_{\text{iso}}$, and *direct volume rendering*, where there is some type of opacity/emission integration along the line of sight. The key difference between these techniques is that isosurfacing displays actual surfaces, while direct volume rendering displays some function of all the values seen along a ray throughout the pixel. Ideally, the display parameters for each technique are interactively controlled by the user. In this paper, we present interactive volume visualization schemes that use ray tracing as their basic computation method.

The basic ray-volume traversal method used in this paper is shown in Fig. 1. This framework allows us to implement volume visualization methods that find exactly one value along a ray. Two such methods described in this paper are isosurfacing and maximum-intensity projection. Maximum-intensity projection is a direct volume rendering technique where the opacity is a function of the maximum intensity seen along a ray. The isosurfacing of rectilinear grids has appeared previously [1], while the isosurfacing of unstructured grids and the maximum-intensity projection are described for the first time in this paper. More general forms of direct volume rendering are not discussed in this paper.

The methods are implemented in a parallel ray tracing system that runs on an SGI Reality Monster, which is a conventional (distributed) shared-memory multiprocessor machine. The only graphics hardware that is used is the high-speed framebuffer. This overall system is described in a previous paper [2]. Conventional wisdom holds that ray tracing is too slow to be competitive with hardware z-buffers. However, when rendering a sufficiently large dataset, ray tracing should be competitive because its low time complexity ultimately overcomes its large time constant [3]. This crossover will happen sooner on a multiple CPU computer because of ray tracing's high degree of intrinsic parallelism. The same arguments apply to the volume traversal problem.

In Section 2, we review previous work, describe several volume visualization techniques, and give an overview of the parallel ray tracing code that provides the backbone of our system. Section 3 describes the data organizational optimizations that allow us to achieve interactivity. In Section 4, we describe our memory optimizations for various types of volume visualization. In Section 5, we show our methods applied to several datasets. We discuss the implications of our results in Section 6, and point to some future directions in Section 7. Some material that is not research-oriented but is helpful for implementors is presented in the appendices.

• *The authors are with the Computer Science Department, University of Utah, Salt Lake City, UT 84112.*
E-mail: {sparker, map, yarden, ppsloan, hansen, shirley}@cs.utah.edu.

For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number 109337.

2 Background

Ray tracing has been used for volume visualization in many works (e.g., [4], [5], [6]). Typically, the ray tracing of a pixel is a kernel operation that could take place within any

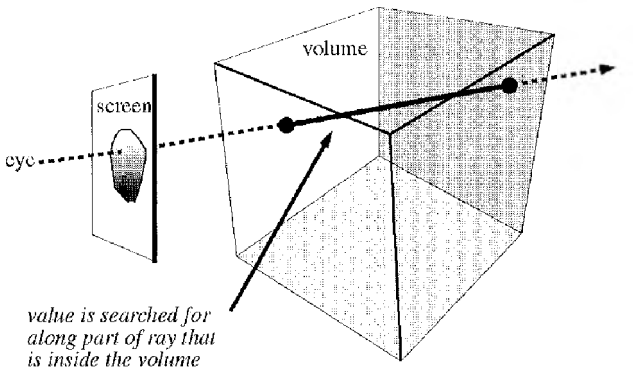


Fig. 1. A ray traverses a volume looking for a specific or maximum value. No explicit surface or volume is computed.

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.

2.1 Efficient Ray Tracing

It is well understood that ray tracing is accelerated through two main techniques [7]: accelerating or eliminating ray/voxel intersection tests and parallelization. Acceleration is usually accomplished by a combination of spatial subdivision and early ray termination [4], [8], [9].

Ray tracing for volume visualization naturally lends itself towards parallel implementations [10], [11]. The computation for each pixel is independent of all other pixels and the data structures used for casting rays are usually read-only. These properties have resulted in many parallel implementations. A variety of techniques have been used to make such systems parallel, and many successful systems have been built (e.g., [10], [12], [13], [14]). These techniques are surveyed by Whitman [15].

2.2 Methods of Volume Visualization

There are several ways that scalar volumes can be made into images. The most popular simple volume visualization techniques that are not based on cutting planes are *isosurfacing*, *maximum-intensity projection*, and *direct volume rendering*.

In isosurfacing, a surface is displayed that is the locus of points where the scalar field equals a certain value. There are several methods for computing images of such surfaces, including constructive approaches such as marching cubes [16], [17] and ray tracing [18], [19], [20].

In maximum-intensity projection (MIP), each value in the scalar field is associated with an intensity and the maximum intensity seen through a pixel is projected onto that pixel [21]. This is a "winner-takes-all" algorithm and, thus, looks more like a search algorithm than a traditional volume color/opacity accumulation algorithm.

More traditional direct volume rendering algorithms accumulate color and opacity along a line of sight [4], [5], [6], [8], [22]. This requires more intrinsic computation than MIP and we will not deal with it in this paper.

2.3 Traversals of Volume Data

Traversal algorithms for volume data are usually customized to the details of the volume data characteristics. The three most common types [23] of volume data used in applications are shown in Fig. 2.

To traverse a line through rectilinear data some type of incremental traversal is used (e.g., [24], [25]). Because there are many cells, a hierarchy can be used that skips "uninteresting" parameter intervals, which increases performance [26], [27], [28], [29].

For curvilinear volumes, the ray can be intersected against a polygonal approximation to the boundary and, then, a more complex cell-to-cell traversal can be used [30].

For unstructured volumes, a similar technique can be used [31], [32]. Once the ray is intersected with a volume, it can be tracked from cell-to-cell using the connectivity information present in the mesh.

Another possibility for both curvilinear and unstructured grids is to resample to a rectilinear grid [33], although resampling artifacts and data explosion are both issues.

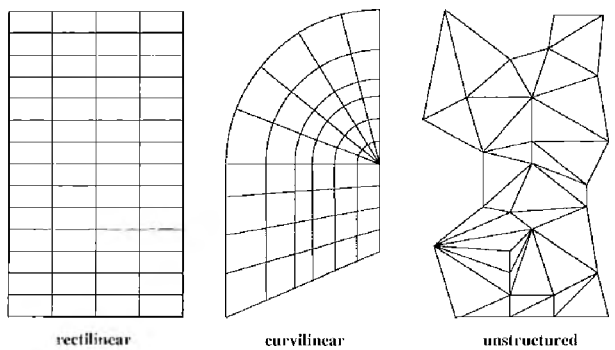


Fig. 2. The three most common types of point-samples volume data.

3 Traversal Optimizations

Our system organizes the data into a shallow rectilinear hierarchy for ray tracing. For unstructured or curvilinear grids, a rectilinear hierarchy is imposed over the data domain. Within a given level of the hierarchy we use the incremental method described by Amanatides and Woo [24].

3.1 Memory Bricking

The first optimization is to improve data 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 [21], [34] (Fig. 3). Our use of lookup tables is particularly similar to that of Sakas et al. [21].

Effectively utilizing the cache hierarchy is a crucial task in designing algorithms for modern architectures. Bricking or 3D tiling has been a popular method for increasing locality for ray cast volume rendering. The dataset is reordered into $n \times n \times n$ cells which then fill the entire volume. On a machine with 128 byte cache lines, and using 16 bit data values, n is exactly 4. However, using float (32 bit) datasets, n is closer to 3.

Effective translation lookaside buffer (TLB) utilization is also becoming a crucial factor in algorithm performance. The same technique can be used to improve TLB hit rates by creating $m \times m \times m$ bricks of $n \times n \times n$ cells. For example, a

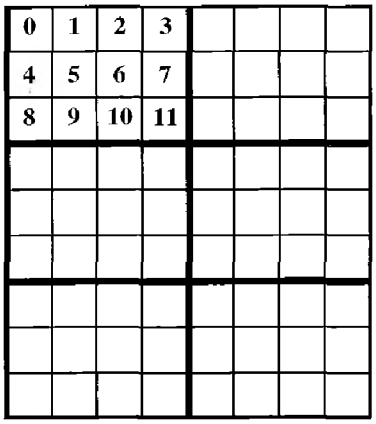


Fig. 3. Cells can be organized into "tiles" or "bricks" in memory to improve locality. The numbers in the first brick represent layout in memory. Neither the number of atomic voxels nor the number of bricks need be a power of two.

$40 \times 20 \times 19$ volume could be decomposed into $4 \times 2 \times 2$ macrobricks of $2 \times 2 \times 2$ bricks of $5 \times 5 \times 5$ cells. This corresponds to $m = 2$ and $n = 5$. Because 19 cannot be factored by $mn = 10$, one level of padding is needed. We use $m = 5$ for 16 bit datasets and $m = 6$ for 32 bit datasets.

The resulting offset q into the data array can be computed for any x, y, z triple with the expression:

$$\begin{aligned} q = &((x \div n) \div m)n^3m^3((N_z \div n) \div m)((N_y \div n) \div m)+ \\ &((y \div n) \div m)n^3m^3((N_z \div n) \div m)+ \\ &((z \div n) \div m)n^3m^3+ \\ &((x \div n) \bmod m)n^3m^2+ \\ &((y \div n) \bmod m)n^3m+ \\ &((z \div n) \bmod m)n^3+ \\ &(x \bmod n \times n)n^2+ \\ &(y \bmod n) \times n+ \\ &(z \bmod n), \end{aligned}$$

where N_x, N_y, and N_z are the respective sizes of the dataset.

This expression contains many integer multiplication, divide and modulus operations. On modern processors, these operations are extremely costly (32+ cycles for the MIPS R10000). Where n and m are powers of two, these operations can be converted to bitshifts and bitwise logical operations. However, the ideal size is rarely a power of two, thus, a method that addresses arbitrary sizes is needed. Some of the multiplications can be converted to shift/add operations, but the divide and modulus operations are more problematic. The indices could be computed incrementally, but this would require tracking nine counters, with numerous comparisons and poor branch prediction performance.

Note that this expression can be written as:

$$q = F_x(x) + F_y(y) + F_z(z)$$

where

$$\begin{aligned} F_x(x) &= ((x \div n) \div m)n^3m^3((N_z \div n) \div m)((N_y \div n) \div m) \\ &\quad + ((x \div n) \bmod m)n^3m^2+ \\ &\quad (x \bmod n \times n)n^2 \\ F_y(y) &= ((y \div n) \div m)n^3m^3((N_z \div n) \div m)+ \\ &\quad ((y \div n) \bmod m)n^3m+ \\ &\quad (y \bmod n) \times n \\ F_z(z) &= ((z \div n) \div m)n^3m^3+ \\ &\quad ((z \div n) \bmod m)n^3+ \\ &\quad (z \bmod n). \end{aligned}$$

We tabulate F_x, F_y, and F_z and use x, y, and z, respectively, to find three offsets in the array. These three values are summed to compute the index into the data array. These tables will consist of N_x, N_y, and N_z elements, respectively. The total sizes of the tables will fit in the primary data cache of the processor even for very large data set sizes. Using this technique, we note that one could produce mappings which are much more complex than the two level bricking described here, although it is not at all obvious which of these mappings would achieve the highest cache utilization.

For many algorithms, each iteration through the loop examines the eight corners of a cell. In order to find these eight values, we need to only lookup $F_x(x)$, $F_x(x+1)$, $F_y(y)$, $F_y(y+1)$, $F_z(z)$, and $F_z(z+1)$. This consists of six index table lookups for each eight data value lookups.

3.2 Multilevel Grid

The other basic optimization we use is a multilevel spatial hierarchy to accelerate the traversal of empty cells, as is shown in Fig. 4. Cells are grouped divided into equal portions and, then, a "macrocell" is created which contains the minimum and maximum data value for its children cells. This is a common variant of standard ray-grid techniques [35] and is especially similar to previous multilevel grids [36], [37]. The use of minimum/maximum caching has been shown to be useful [28], [29], [38]. 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 typical complexity of this search will be $O(\sqrt[3]{n})$ for a three level hierarchy [36] on an $n \times n \times n$ dataset. While the worst case complexity is still $O(n)$, it is difficult to imagine an isosurface occurring 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 of intermediate levels. We have experimented with modifying the number of levels in the hierarchy and empirically determined that a trilevel 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 trilevel hierarchy, the storage overhead is negligible (< 0.5 percent 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.

Macrocells can be indexed with the same approach as used for memory bricking of the data values. However, in this case, there will be three table lookups for each macrocell. This, combined with the significantly smaller memory footprint of the macrocells made the effect of bricking the macrocells negligible.

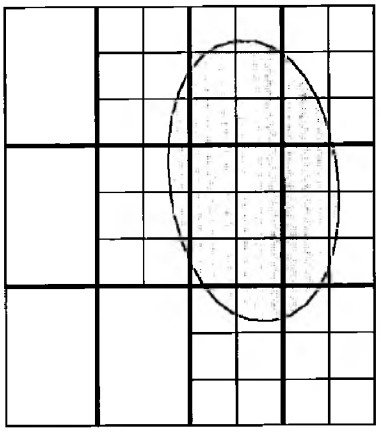


Fig. 4. With a two-level hierarchy, rays can skip empty space by traversing larger cells. A three-level hierarchy is used for most of the examples in this paper.

4 Algorithms

This section describes three types of volume visualization that use ray tracing:

- isosurfacing on rectilinear grids,
- isosurfacing on unstructured meshes,
- maximum-intensity projection on rectilinear grids.

The first two require an operation of the form: Find a specific scalar value along a ray. The third asks: What is the maximum value along a ray. All of these are searches that can benefit from the hierarchical data representations described in the previous section.

4.1 Rectilinear Isosurfacing

Our 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. A 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 physical texture memory, which is currently available up to 64 MBytes. However, newer architectures that use main memory for textures eliminate this issue.

If we assume a regular volume with even grid point spacing arranged in a rectilinear array, then ray-isosurface intersection is straightforward. Analogous simple schemes exist for intersection of tetrahedral cells as described below.

To find an intersection (Fig. 5), the ray $\vec{a}+t\vec{b}$ 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:

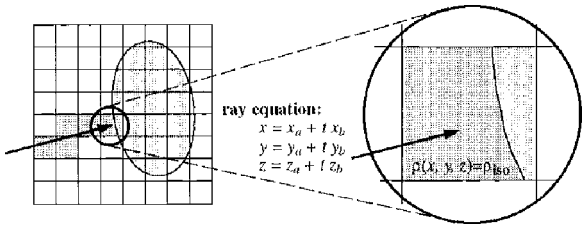


Fig. 5. The ray traverses each cell (left) and, when a cell is encountered that has an isosurface in it (right), an analytic ray-isosurface intersection computation is performed.

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

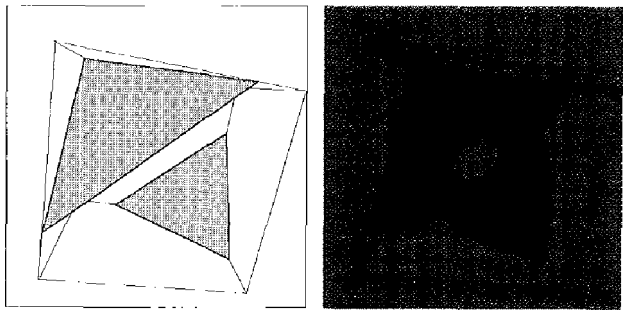
When approximating ρ with a trilinear interpolation between discrete grid points, this equation will expand to a cubic polynomial in t. This cubic can then be solved in closed form to find the intersections of the ray with the isosurface in that cell. We use the closed form solution for convenience since its stability and efficiency have not proven to be major issues for the data we have used in our tests. 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 Appendix A. Note that using trilinear interpolation directly will produce more complex isosurfaces than is possible with a marching cubes algorithm. An example of this is shown in Fig. 6, which illustrates case 4 from Lorensen and Cline's paper [17]. Techniques such as the Asymptotic Decider [39] could disambiguate such cases, but they would still miss the correct topology due to the isosurface interpolation scheme.

4.2 Unstructured Isosurfacing

For unstructured meshes, the same memory hierarchy is used as is used in the rectilinear case. However, we can control the resolution of the cell size at the finest level. We chose a resolution which uses approximately the same number of leaf nodes as there are tetrahedral elements. At the leaf nodes a list of references to overlapping tetrahedra is stored (Fig. 7). For efficiency, we store these lists as integer indices into an array of all tetrahedra.

Rays traverse the cell hierarchy in a manner identical to the rectilinear case. However, when a cell is detected that might contain an isosurface for the current isovalue, each of the tetrahedra in that cell are tested for intersection. No connectivity information is used for the tetrahedra; instead, they are treated as independent items, just as in a traditional surface-based ray tracer.

The 4D isosurface for a tetrahedron is computed implicitly using barycentric coordinates. The intersection of the parameterized ray and the isoplane is computed directly, using the implicit equations for the plane and the parametric equation for the ray. The intersection point is checked to see if it is still within the bounds of the tetrahedron by making sure the barycentric coordinates are all positive. Details of this intersection code are described in Appendix B.



(a) (b)

Fig. 6. (a) The isosurface from the marching cubes algorithm. (b) The isosurface resulting the true cubic behavior inside the cell.

4.3 Maximum-Intensity Projection

The maximum-intensity projection (MIP) algorithm seeks the largest data value that intersects a particular ray. It utilizes the same shallow spatial hierarchy described above for isosurface extraction. In addition, a priority queue is used to track the cells or macrocells with the maximal values. For each ray, the priority queue is first initialized with single top level macrocell. The maximum data value for the dataset is used as the priority value for this entry in the priority queue. The algorithm repeatedly pulls the largest entry from the priority queue and breaks it into smaller (lower level) macrocells. Each of these cells are inserted into the priority queue with the precomputed maximum data value for that region of space. When the lowest-level cells are pulled from the priority queue, the algorithm traverses the segment of the ray which intersects the macrocell. Bilinear interpolation is used at the intersection of the ray with cell faces since these are the extremal values of the ray-cell intersection in a linear interpolation scheme. For each data cell face which intersects the ray, a bilinear interpolation of the data values is computed, and the maximum of these values in stored again in the priority queue. Finally, when one of these data maxima appears at the head of the priority queue, the algorithm has found the maximum data value for the entire ray.

To reduce the average length of the priority queue, the algorithm performs a single trilinear interpolation of the data at one point to establish a lower-bound for the maximum value of the ray. Macrocells and datacells which do not exceed this lower-bound are not entered into the priority queue. To obtain this value, we perform the trilinear interpolation using the t corresponding to the maximum value from whatever previous ray a particular processor has computed. Typically, this will be a value within the same block of pixels and exploits image-space coherence. If not, it still provides a bound on the maximum along the ray. If this t value is unavailable (due to program startup, or a ray missing the data volume), we choose the midpoint of the ray segment which intersects the data volume. This is a simple heuristic which improves the performance for many datasets.

Similar to the isosurface extraction algorithm, the MIP algorithm uses the 3D bricking memory layout for efficient cache utilization when traversing the data values. Since each processor will be using a different priority queue as it processes each ray, an efficient implementation of a priority queue which does not perform dynamic memory allocation is essential for performance of the algorithm.

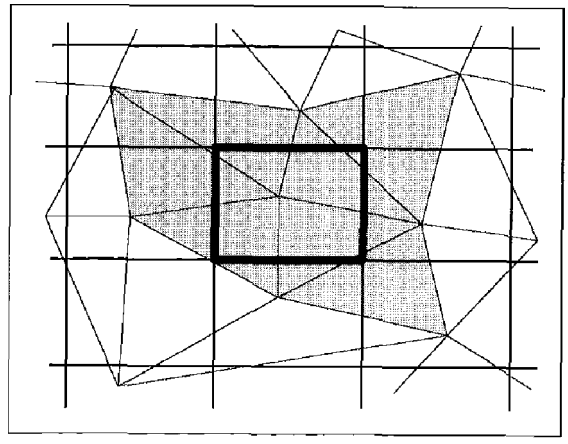


Fig. 7. For a given leaf cell in the rectilinear grid, indices to the shaded elements of the unstructured mesh are stored.

5 Results

We applied ray tracing isosurface extraction to interactively visualize the Visible Woman dataset. The Visible Woman dataset is available through the National Library of Medicine as part of its Visible Human Project [40]. We used the computed tomography (CT) data which was acquired in 1mm slices with varying in-slice resolution. This rectilinear data is composed of 1,734 slices of 512×512 images at 16 bits. The complete dataset is 910 MBytes. Rather than down-sample the data with a loss of resolution, we utilize the full resolution data in our experiments. As previously described, our 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, and shading the resulting intersection point.

Fig. 8 shows a ray tracing for two isosurface values. Fig. 9 illustrates how shadows can improves the accuracy of our geometric perception. Fig. 10 shows a transparent skin isosurface over a bone isosurface. Table 1 shows the percentages of time spent in each of these phases, as obtained through the cycle hardware counter in Silicon Graphics' Speedshop.[1] As can be seen, we achieve about 10 frames per second (FPS) interactive rates while rendering the full, nearly 1 GByte, dataset.

Table 2 shows the scalability of the algorithm from 1 to 128 processors. View 2 uses a zoomed out viewpoint with approximately 75 percent pixel coverage whereas view 1 has nearly 100 percent 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 costs and load imbalance. 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

1. Speedshop is the vendor provided performance analysis environment for the SGI IRIX operating system.

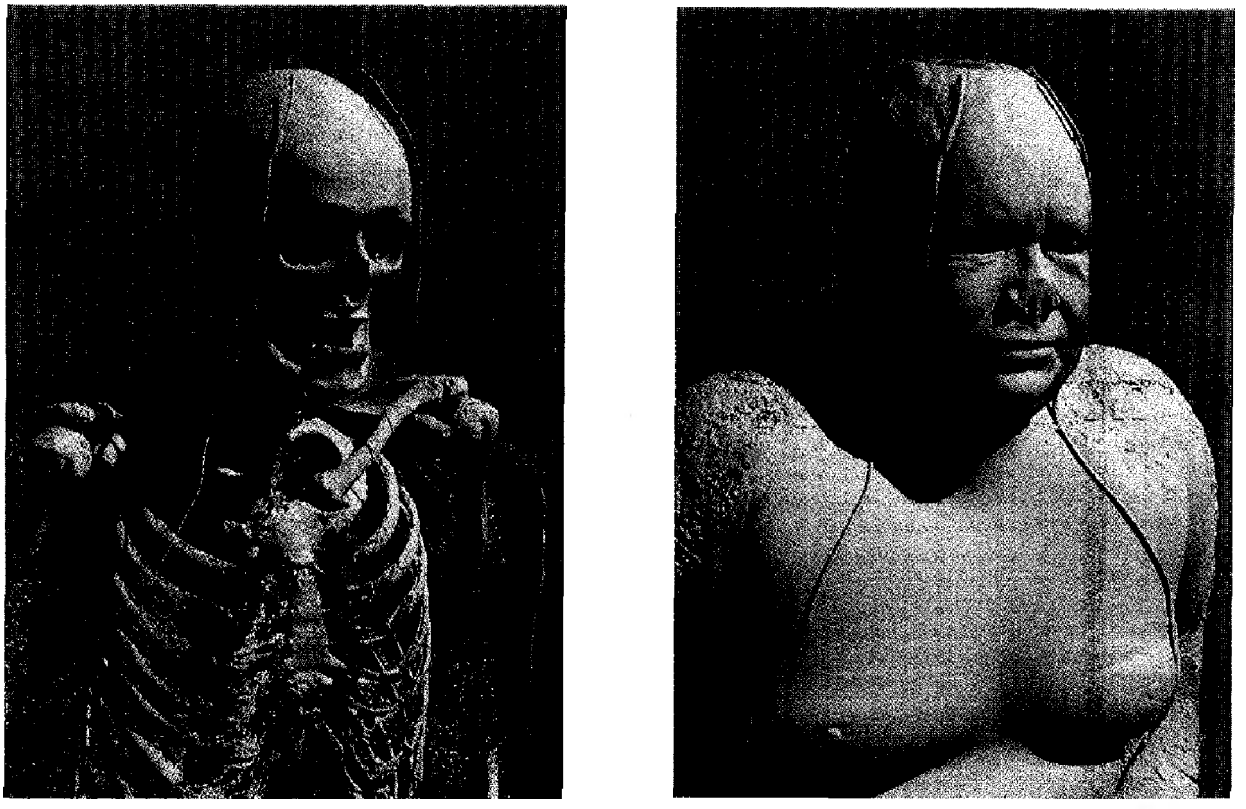


Fig. 8. Ray tracings of the bone and skin isosurfaces of the Visible Woman.

could reduce the image size in order to restore the interactive rates. Efficiencies are 91 percent and 80 percent 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 3 shows the improvements which were obtained through the data bricking and spatial hierarchy optimizations.

Using a ray tracing architecture, it is simple to map each isosurface with an arbitrary texture map. The Visible Man dataset includes both CT data and photographic data. Using a texture mapping technique during the rendering phase allows us to add realism to the resultant isosurface. The photographic cross section data which was acquired in 0.33mm slices can be registered with the CT data. This combined data can be used as a texture mapped model to add realism to the resulting isosurface. The size of the photographic dataset is approximately 13 GBytes, which clearly is too large to fit into texture memory. When using texture mapping hardware, it is up to the user to implement intelligent texture memory management. This makes achieving effective texture performance nontrivial. In our implementation, we down-sampled this texture by a factor of 0.6 in two of the dimensions so that it occupied only 5.1 GBytes. The frame rates for this volume with and without shadows and texture are shown in Table 4. A sample image is shown in Fig. 11. We can achieve interactive rates when applying the full resolution photographic cross sections to the full resolution CT data. We know of no other work which achieves these rates.

Fig. 12 shows an isosurface from an unstructured mesh made up of 1.08 million elements which contains adaptively refined tetrahedral elements. The heart and lungs shown are polygonal meshes that serve as landmarks. The rendering times for this data, rendered without the polygonal landmarks at 512×512 pixel resolution, is shown in Table 5. As would be expected, the FPS is lower than for structured data, but the method scales well. We make the number of lowest-level cells proportional to the number of tetrahedral elements, and the bottleneck is intersection with individual tetrahedral elements. This dataset composed of adaptively refined tetrahedra with volume differences of two orders of magnitude.

Fig. 13 shows a maximum-intensity projection of the Visible Female dataset. This dataset runs in approximately 0.5 to 2 FPS on 16 processors. Using the "use last t" optimization saves approximately 15 percent of runtime. Generating such a frame rate using conventional graphics hardware would require approximately a 1.8 GPixel/second pixel fill rate and 900 Mbytes of texture memory.

6 Discussion

We contrast applying our algorithm to explicitly extracting polygonal isosurfaces from the Visible Woman data set. For the skin isosurface, we generated 18,068,534 polygons. For the bone isosurface, we generated 12,922,628 polygons. These numbers are consistent with those reported by Lorensen given that he was using a cropped version of

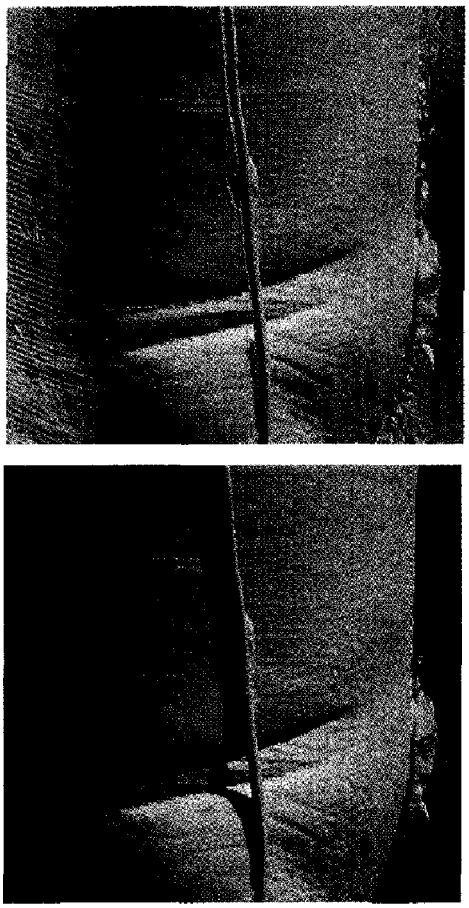


Fig. 9. A ray tracing with and without shadows.

the volume [41]. With this number of polygons, it would be challenging to achieve interactive rendering rates on conventional high-end graphics hardware. Our method can render a ray-traced isosurface of this data at roughly ten frames per second using a 512×512 image on 64 processors. Table 6 shows the extraction time for the bone isosurface using both NOISE [42] and marching cubes [17]. Note that because we are using static load balancing, these numbers would improve with a dynamic load balancing scheme. However, this would still not allow interactive modification of the isovalue while displaying the isosurface, although using a downsampled or simplified detail volume would allow interaction at the cost of some resolution. Simplified, precomputed isosurfaces could also yield interaction, but storage and precomputation time would be significant. Triangle stripping could improve display rates by up to a factor of three because isosurface meshes are usually transform bound. Note that we gain efficiency for both the extraction and rendering components by not explicitly extracting the geometry. Our algorithm is therefore not well-suited for applications that will use the geometry for nongraphics purposes.

The interactivity of our system allows exploration of both the data by interactively changing the isovalue or viewpoint. For example, one could view the entire skeleton and interactively zoom in and modify the isovalue to examine

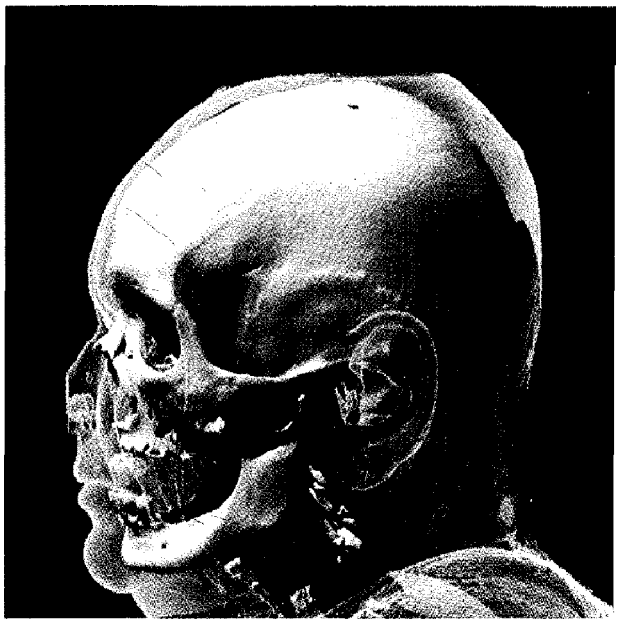


Fig. 10. Ray tracings of the skin and bone isosurfaces with transparency.

the detail in the toes all at about 10 FPS. The variation in framerate is shown in Fig. 14.

Brady et al. [43] describe a system which allows, on a Pentium workstation with accelerated graphics, interactive navigation through the Visible Human data set. Their technique is two-fold:

1. Combine frustum culling with intelligent paging from disk of the volume data, and
2. Utilize a two-phase perspective volume rendering method which exploits coherence in adjacent frames.

Their work differs from ours in that they are using incremental direct volume rendering while we are exploiting isosurface or MIP rendering. This is evidenced by their incremental rendering times of about 2 seconds per frame for a 480×480 image. A full (nonincremental) rendering is nearly 20 seconds using their technique. For a single CPU, our isosurface rendering time is several seconds per frame (see Table 2) depending on viewpoint. While it is difficult to directly compare these techniques due to their differing application focus, our method allows for the entire data set to reside within the view frustum without severe performance penalties since we are exploiting parallelism.

The architecture of the parallel machine plays an important role in the success of this technique. Since any

TABLE 1
Data from Ray Tracing the Visible Woman

Isosurface	Traversal	Intersec.	Shading	FPS
Skin ($\rho = 600.5$)	55%	22%	23%	7-15
Bone ($\rho = 1224.5$)	66%	21%	13%	6-15

The frames-per-second (FPS) gives the observed range for the interactively generated viewpoints on 64 CPUs.

TABLE 2
Scalability Results for Ray Tracng the Bone Isosurface in the Visible Human

	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

A 512 × 512 image was generated using a single view of the bone isosurface.

Fig. 11. A 3D texture applied to an isosurface from the Visible Man dataset.

processor can randomly access the entire dataset, the dataset must be available to each processor. Nonetheless, there is fairly high locality in the dataset for any particular processor. As a result, a shared memory or distributed shared memory machine, such as the SGI Origin 2000, is ideally suited for this application. The load balancing mechanism also requires a fine-grained low-latency communication mechanism for synchronizing work assignments and returning completed image tiles. With an attached InfiniteReality graphics engine, we can display images at high frame rates without network bottlenecks. We feel that implementing a similar technique on a distributed memory machine would be extraordinarily challenging, and would probably not achieve the same rates without duplicating the dataset on each processor.

7 Future Work and Conclusions

Since all computation is performed in software, there are many avenues which deserve exploration. Ray tracers have a relatively clean software architecture, in which techniques can be added without interfering with existing techniques, without re-unrolling large loops and without complicated state management as are characteristic of a typical polygon renderer.

We believe the following possibilities are worth investigating:

- Exploration of other hierarchical methods in addition to the multilevel hierarchy described above.
- Combination with other scalar and vector visualization tools, such as cutting planes, surface maps, streamlines, etc.
- Using higher-order interpolants. Although numerical root finding would be necessary, the images might look better [19]. Since the intersection routine is not the bottleneck the degradation in performance might be reasonable.

We have shown that ray tracing can be a practical alternative to explicit isosurface extraction for very large datasets. As data sets get larger and as general purpose processing hardware becomes more powerful, we expect this to become a very attractive method for visualizing large scale scalar data both in terms of speed and rendering accuracy.

TABLE 3
Times in Seconds for Optimizations for Ray Tracing the Visible Human

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

A 512 × 512 image was generated on 16 processors using a single view of an isosurface.

Appendix A

Ray-Isosurface Intersection for Trilinear Boxes

This appendix expands on some details of the intersection of a ray and a trilinear surface. It is not new research, but is helpful for implementors.

A rectilinear volume is composed of a three dimensional array of point samples that are aligned to the Cartesian axes and are equally spaced in a given dimension. A single cell from such a volume is shown in Fig. 15. Other cells can be

Fig. 12. Ray tracing of a 1.08 million element unstructured mesh from bioelectric field simulation. The heart and lungs are represented as landmark polygonal meshes and are not part of the isosurface.

generated by exchanging indices (i, j, k) for the zeros and ones in the figure.

The density at a point within the cell is found using *trilinear* interpolation:

$$\begin{aligned}\rho(u,v,w) =&(1-u)(1-v)(1-w)\rho_{000}+\\ &(1-u)(1-v)(w)\rho_{001}+\\ &(1-u)(v)(1-w)\rho_{010}+\\ &(u)(1-v)(1-w)\rho_{100}+\\ &(u)(1-v)(w)\rho_{101}+\\ &(1-u)(v)(w)\rho_{011}+\\ &(u)(v)(1-w)\rho_{110}+\\ &(u)(v)(w)\rho_{111},\end{aligned} \tag{1}$$

where

$$\begin{aligned}u &= x - x_0 x_1 - x_0\\ v &= y - y_0 y_1 - y_0\\ w &= z - z_0 z_1 - z_0.\end{aligned} \tag{2}$$

Note that

TABLE 4
Frame Rates Varying Shadow and Texture for the Visible Male Dataset on 64 CPUs (FPS)

no shadows, no texture	15.9
shadows, no texture	8.7
no shadows, texture	12.6
shadows, texture	7.5

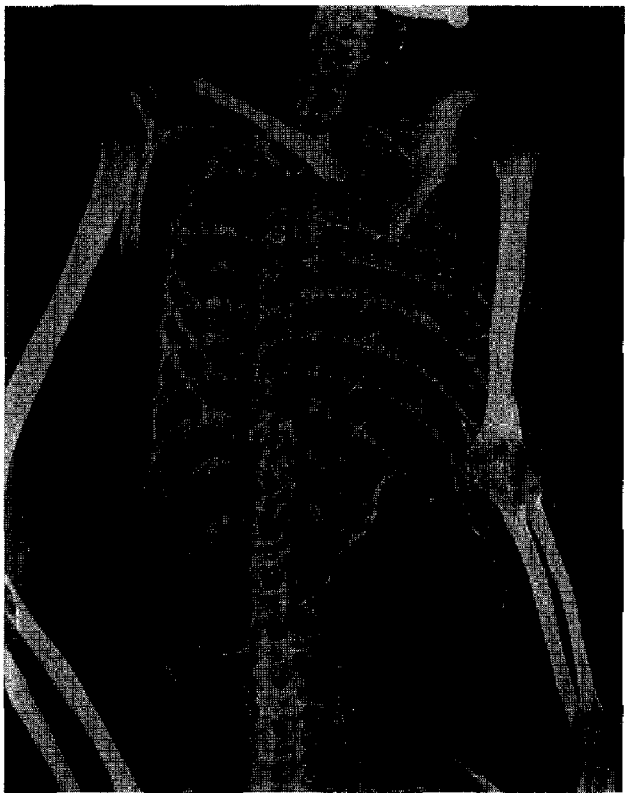


Fig. 13. A maximum-intensity projection of the Visible Female dataset.

$$\begin{aligned}1-u &= x_1 - xx_1 - x_0\\ 1-v &= y_1 - yy_1 - y_0\\ 1-w &= z_1 - zz_1 - z_0.\end{aligned} \tag{3}$$

If we redefine $u_0 = 1 - u$ and $u_1 = u$, and similar definitions for v_0, v_1, w_0, w_1, then we get:

TABLE 5
Data from Ray Tracing Unstructured Grids at 512×512 Pixels on 1 to 124 Processors

# cpus	FPS	speedup
1	0.108	1.00
2	0.21	1.97
3	0.32	2.95
4	0.42	3.91
6	0.63	5.86
8	0.84	7.78
12	1.25	11.56
16	1.64	15.20
24	2.44	22.58
32	3.21	29.68
48	4.76	44.07
64	6.46	59.81
96	9.05	83.80
124	11.13	103.06

The adaptively refined dataset is from a bioelectric field problem.

TABLE 6
Explicit Bone Isosurface Extraction Times in Seconds

# cpus	NOISE build	NOISE extract	Marching cubes
1	4838	110	627
2	2109	81	324
4	1006	56	171
8	885	31	93
16	437	24	49
32	118	14	26
64	59	12	24

$$\rho = \sum_{i,j,k=0,1} u_i v_j w_k \rho_{ijk}$$

For a given point (x, y, z) in the cell, the surface normal is given by the gradient with respect to (x, y, z):

$$\vec{N} = \vec{\nabla}\rho = (\partial\rho\partial x, \partial\rho\partial y, \partial\rho\partial z).$$

So, the normal vector of $(N_x, N_Y, N_z) = \vec{\nabla}\rho$ is

$$N_x = \sum_{i,j,k=0,1} (-1)^{i+1} v_j w_k x_1 - x_0 \rho_{ijk}$$
$$N_y = \sum_{i,j,k=0,1} (-1)^{j+1} u_i w_k y_1 - y_0 \rho_{ijk}$$
$$N_z = \sum_{i,j,k=0,1} (-1)^{k+1} u_i v_j z_1 - z_0 \rho_{ijk}.$$

Lin and Ching [18] described a method for intersecting a ray with a trilinear cell. We derive a similar result that is more tailored to our implementation.

See Fig 16. Given a ray $\vec{p} = \vec{a} + t\vec{b}$, the intersection with the isosurface occurs where $\rho(\vec{p}) = \rho_{\text{iso}}$. We can convert this ray into coordinates defined by (u_0, v_0, w_0): $\vec{p}_0 = \vec{a}_0 + t\vec{b}_0$ and a third ray defined by $\vec{p}_1 = \vec{a}_1 + t\vec{b}_1$. These rays $\vec{p}_0 = \vec{a}_0 + t\vec{b}_0$ and $\vec{p}_1 = \vec{a}_1 + t\vec{b}_1$ are now used for the intersection computation. These two rays are in the two coordinate systems (Fig. 16):

$$\vec{a}_0 = (u_0^a, v_0^a, w_0^a)$$
$$= (x_1 - x_a x_1 - x_0, y_1 - y_a y_1 - y_0, z_1 - z_a z_1 - z_0),$$

and

$$\vec{b}_0 = (u_0^b, v_0^b, w_0^b) = (x_b x_1 - x_0, y_b y_1 - y_0, z_b z_1 - z_0).$$

These equations are different because $\vec{a}_0$ is a location and $\vec{b}_0$ is a direction. The equations are similar for $\vec{a}_1$ and $\vec{b}_1$:

$$\vec{a}_1 = (u_1^a, v_1^a, w_1^a)$$
$$= (x_a - x_0 x_1 - x_0, y_a - y_0 y_1 - y_0, z_a - z_0 z_1 - z_0),$$

and

$$\vec{b}_1 = (u_1^b, v_1^b, w_1^b) = (-x_b x_1 - x_0, -y_b y_1 - y_0, -z_b z_1 - z_0).$$

Note that t is the same for all three rays. This point can be found by traversing the cells and doing a brute-force algebraic solution for t. The intersection with the isosurface $\rho(\vec{p}) = \rho_{iso}$ occurs where:

$$\rho_{\text{iso}} = \sum_{i,j,k=0,1} \left(u_i^a + t u_i^b\right)\left(v_j^a + t v_j^b\right)\left(w_k^a + t w_k^b\right)\rho_{ijk}$$

This can be simplified to a cubic polynomial in t:

$$At^3 + Bt^2 + Ct + D = 0,$$

where

$$A = \sum_{i,j,k=0,1} u_i^b v_j^b w_k^b \rho_{ijk}$$
$$B = \sum_{i,j,k=0,1} \left(u_i^a v_j^b w_k^b + u_i^b v_j^a w_k^b + u_i^b v_j^b w_k^a\right)\rho_{ijk}$$
$$C = \sum_{i,j,k=0,1} \left(u_i^b v_j^a w_k^a + u_i^a v_j^b w_k^a + u_i^a v_j^a w_k^b\right)\rho_{ijk}$$
$$D = -\rho_{iso} + \sum_{i,j,k=0,1} u_i^a v_j^a w_k^a \rho_{ijk}.$$

The solution to a cubic polynomial is discussed the article by Schwarze [44]. We used his code (available on the web in several *Graphics Gems* archive sites) with two modifications: special cases for quadratic or linear solutions (his code assumes A is nonzero), and the EQN_EPS parameter was set to 1.e-30 which provided for maximum stability for large coefficients.

APPENDIX B

RAY-ISOSURFACE INTERSECTION FOR BARYCENTRIC TETRAHEDRA

This appendix is geared toward implementors and discusses the details of intersecting a ray with a barycentric tetrahedral isosurface.

An unstructured mesh is composed of three dimensional point samples arranged into a simplex of tetrahedra. A single cell from such a volume is shown in Fig. 17, where the four vertices are $\mathbf{p}_i = (x_i, y_i, z_i)$.

The density at a point within the cell is found using *barycentric* interpolation:

$$\rho(\alpha_0, \alpha_1, \alpha_2, \alpha_3) = \alpha_0\rho_0 + \alpha_1\rho_1 + \alpha_2\rho_2 + \alpha_3\rho_3,$$

where

$$\alpha_0 + \alpha_1 + \alpha_2 + \alpha_3 = 1.$$

Similar equations apply to points in terms of the vertices. For points inside the tetrahedron, all barycentric coordinates are positive.

One way to compute barycentric coordinates is to measure the distance from the plane that defines each face (Fig. 18). This is accomplished by choosing a plane equation $f_0(\mathbf{p}) = 0$ such that $f_0(\mathbf{p}_0) = 1$. Such equations for all four plane-faces of the tetrahedron allow us to compute barycentric coordinates of a point $\mathbf{p}$ directly: $\alpha_i(\mathbf{p}) = f_i(\mathbf{p})$.

If we take the ray $\mathbf{p}(t) = \mathbf{a} + t\vec{\mathbf{b}}$, then we get an equation for the density along the ray:

$$\rho(t) = \sum_{i=0}^{3} f_i(\mathbf{a} + t\vec{\mathbf{b}})\rho_i.$$

If we solve for $\rho(t) = \rho_{\text{iso}}$, then we get a linear equation in t,

# processors	Frame rate /Speedup				
1	0.427 /1.00	0.304 /1.00	0.084 /1.00	0.155 /1.00	0.568 /1.00
2	0.84 /1.97	0.60 /1.98	0.17 /1.99	0.31 /2.00	1.13 /1.98
3	1.26 /2.94	0.89 /2.93	0.25 /2.95	0.46 /2.96	1.68 /2.96
4	1.67 /3.91	1.19 /3.92	0.33 /3.96	0.62 /3.97	2.24 /3.94
6	2.45 /5.73	1.76 /5.77	0.50 /5.97	0.93 /5.96	3.29 /5.80
8	3.20 /7.50	2.32 /7.61	0.67 /7.94	1.23 /7.93	4.36 /7.67
12	4.81 /11.26	3.44 /11.30	1.00 /11.89	1.84 /11.88	6.51 /11.47
16	6.38 /14.93	4.59 /15.08	1.33 /15.84	2.45 /15.80	8.64 /15.21
24	9.54 /22.33	6.84 /22.48	1.98 /23.54	3.65 /23.49	12.92 /22.76
32	12.65 /29.61	9.12 /29.96	2.63 /31.38	4.88 /31.47	17.09 /30.10
48	18.85 /44.13	13.52 /44.39	3.92 /46.72	7.30 /47.02	25.27 /44.50
64	24.73 /57.90	17.72 /58.19	5.18 /61.78	9.64 /62.14	32.25 /56.80
96	35.38 /82.82	25.04 /82.23	7.67 /91.38	14.28 /92.02	45.50 /80.14
124	43.06 /100.79	30.28 /99.45	9.73 /115.88	18.17 /117.08	57.70 /101.63

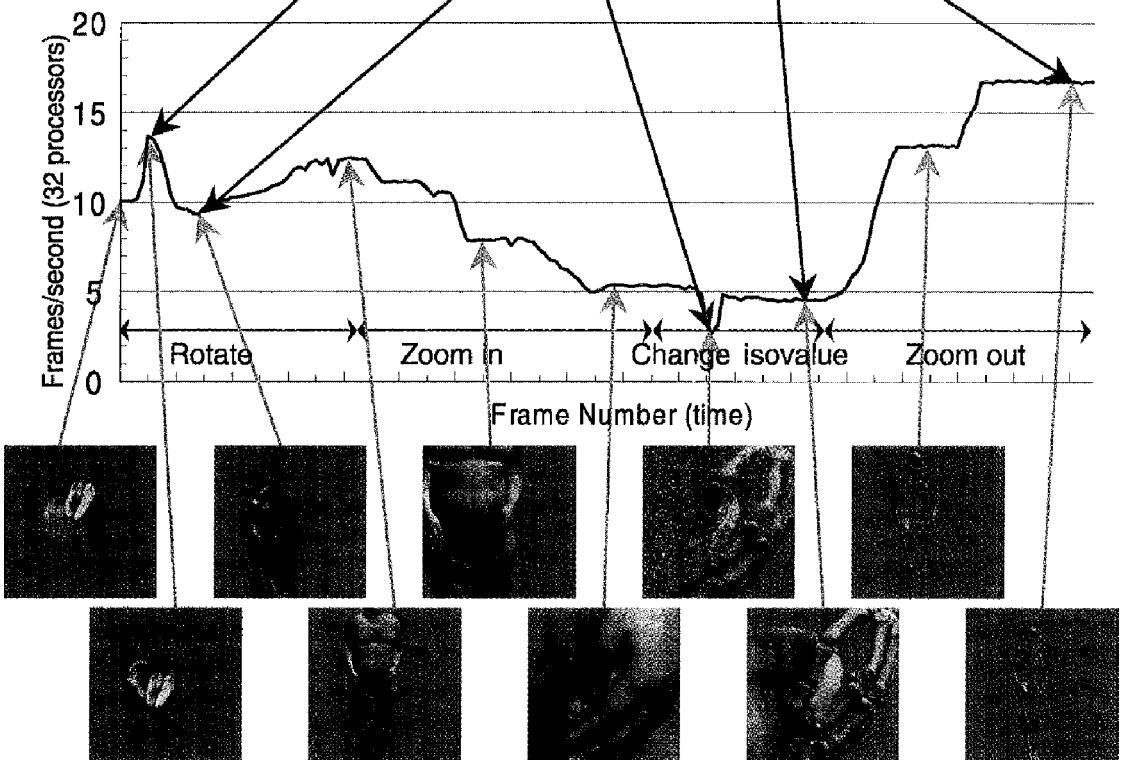


Fig. 14. Variation in framerate as the viewpoint and isovalue changes.

so solution is straightforward. If the resulting barycentric coordinates of $\mathbf{p}(\mathbf{t})$ are all positive, the point is in the tetrahedron, and it is accepted. Finding the normal is just a matter of taking the gradient:

$$\vec{\nabla}\rho(\mathbf{p}) = \sum_{i=0}^{3} \rho_i \vec{\nabla} f_i(\mathbf{p}).$$

Because f_i is just a plane equation of the form $\vec{\mathbf{n}}_i \cdot (\mathbf{p} - \mathbf{q}_i)$, where $\mathbf{q}_i$ is a constant point, the normal vector $\vec{\mathbf{N}}$ is simply

$$\vec{\mathbf{N}} = \sum_{i=0}^{3} \rho_i \vec{\mathbf{n}}_i.$$

This is a constant for the cell, but we do not precompute it since it would require extra memory accesses.

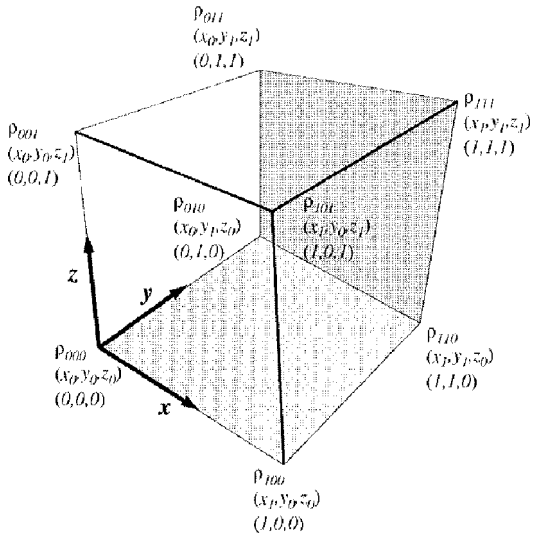


Fig. 15. The geometry for a cell. The bottom coordinates are the (u, v, w) values for the intermediate point.

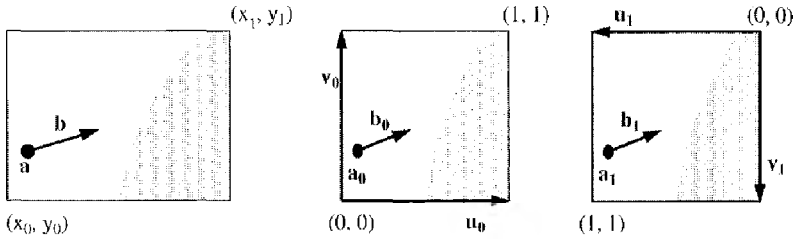


Fig. 16. Various coordinate systems used for interpolation and intersection.

ACKNOWLEDGMENTS

Thanks to Matthew Bane and Michelle Miller for comments on the paper. Thanks to Chris Johnson for providing the open collaborative research environment that allowed this work to happen. Special thanks to Steve Modica and Robert Cummins at SGI for crucial bug fixes in support code. This work was supported by the SGI Visual Supercomputing Center, the Utah State Centers of Excellence, the U.S. Department of Energy, and the U.S. National Science Foundation. Special thanks to Jamie Painter and the Advanced Computing Laboratory at Los Alamos National Laboratory for access to a 128 processor machine for final benchmarks. Ruth Klepfer provide assistance in obtaining the various unstructured data sets.

REFERENCES

[1] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P.-P. Sloan, "Interactive Ray Tracing for Isosurface Rendering," *Proc. Visualization '98*, Oct. 1998.
[2] S. Parker, W. Martin, P.-P. Sloan, P. Shirley, B. Smits, and C. Hansen, "Interactive Ray Tracing," *Proc. Symp. Interactive 3D Graphics*, Apr. 1999.
[3] J.T. Kajiya, "An Overview and Comparison of Rendering Methods," *A Consumer's and Developer's Guide to Image Synthesis*, pp. 259-263, 1988.
[4] M. Levoy, "Display of Surfaces from Volume Data," *IEEE Computer Graphics and Applications*, vol. 8, no. 3, pp. 29-37, 1988.
[5] P. Sabella, "A Rendering Algorithm for Visualizing 3D Scalar Fields," *Computer Graphics*, vol. 22, no. 4, pp. 51-58, July 1988.

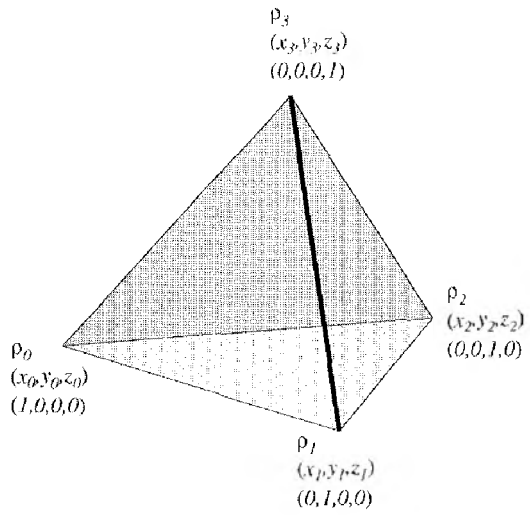


Fig. 17. The geometry for a barycentric tetrahedron. The bottom barycentric coordinates are the $(\alpha_0, \alpha_1, \alpha_2, \alpha_3)$ values for the vertex.

[6] C. Upson and M. Keeler, "V-Buffer: Visible Volume Rendering," *Computer Graphics*, vol. 22, no. 4, pp. 59-64, July 1988.
[7] E. Reinhard, A.G. Chalmers, and F.W. Jansen, "Overview of Parallel Photo-Realistic Graphics," *Proc. Eurographics '98*, 1998.
[8] A. Kaufman, *Volume Visualization*, IEEE CS Press, 1991.
[9] L. Sobierajski and A. Kaufman, "Volumetric Ray Tracing," *Proc. 1994 Workshop Volume Visualization*, pp. 11-18, Oct. 1994.
[10] K.L. Ma, J.S. Painter, C.D. Hansen, and M.F. Krogh, "Parallel Volume Rendering using Binary-Swap Compositing," *IEEE Computer Graphics and Applications*, vol. 14, no. 4, pp. 59-68, July 1993.
[11] M.J. Muuss, "RT and REMRT-Shared Memory Parallel and Network Distributed Ray-Tracing Programs," *USENIX: Proc. Fourth Computer Graphics Workshop*, Oct. 1987.
[12] G. Vézina, P.A. Fletcher, and P.K. Robertson, "Volume Rendering on the MasPar MP-1," *Proc. 1992 Workshop Volume Visualization*, pp. 3-8, Boston, 19-20 Oct. 1992.
[13] P. Schröder and G. Stoll, "Data Parallel Volume Rendering as Line Drawing," *Proc. 1992 Workshop Volume Visualization*, pp. 25-31, Boston, 19-20 Oct. 1992.
[14] M.J. Muuss, "Towards Real-Time Ray-Tracing of Combinatorial Solid Geometric Models," *Proc. BRL-CAD Symp.*, June 1995.
[15] S. Whitman, "A Survey of Parallel Algorithms for Graphics and Visualization," *Proc. High Performance Computing for Computer Graphics and Visualization*, pp. 3-22, Swansea, 3-4 July 1995.
[16] B. Wyvill, G. Wyvill, C. McPheeters, "Data Structures for Soft Objects," *The Visual Computer*, vol. 2, pp. 227-234, 1986.
[17] W.E. Lorensen and H.E. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," *Computer Graphics*, vol. 21, no. 4, pp. 163-169, July 1987.
[18] C.-C. Lin and Y.-T. Ching, "An Efficient Volume-Rendering Algorithm with an Analytic Approach," *The Visual Computer*, vol. 12, no. 10, pp. 515-526, 1996.
[19] S. Marschner and R. Lobb, "An Evaluation of Reconstruction Filters for Volume Rendering," *Proc. Visualization '94*, pp. 100-107, Oct. 1994.
[20] M. Sramek, "Fast Surface Rendering from Raster Data by Voxel Traversal Using Chessboard Distance," *Proc. Visualization '94*, pp. 188-195, Oct. 1994.
[21] G. Sakas, M. Grimm, and A. Savopoulos, "Optimized Maximum Intensity Projection (MIP)," *Proc. Eurographics Rendering Workshop 1995*, June 1995.
[22] R.A. Drebin, L. Carpenter, and P. Hanrahan, "Volume Rendering," *Computer Graphics*, vol. 22, no. 4, pp. 65-74, July 1988.
[23] D. Speray and S. Kennon, "Volume Probes: Interactive Data Exploration on Arbitrary Grids," *Proc. 1990 Workshop Volume Visualization*, pp. 5-12, San Diego, 1990.
[24] J. Amanatides and A. Woo, "A Fast Voxel Traversal Algorithm for Ray Tracing," *Proc. Eurographics '87*, 1987.
[25] A. Fujimoto, T. Tanaka, and K. Iwata, "Arts: Accelerated Ray-Tracing System," *IEEE Computer Graphics and Applications*, pp. 16-26, Apr. 1986.
[26] J. Danskin and P. Hanrahan, "Fast Algorithms for Volume Ray Tracing," *Proc. 1992 Workshop Volume Visualization*, pp. 91-98, 1992.
[27] M. Levoy, "Efficient Ray Tracing of Volume Data," *ACM Trans. Graphics*, vol. 9, no. 3, pp. 245-261, July 1990.
[28] J. Wilhelms and A. Van Gelder, "Octrees for Faster Isosurface Generation," *Proc. 1990 Workshop Volume Visualization*, pp. 57-62, San Diego, Calif., 1990.
[29] J. Wilhelms and A. Van Gelder, "Octrees for Faster Isosurface Generation," *ACM Trans. Graphics*, vol. 11, no. 3, pp. 201-227, July 1992.

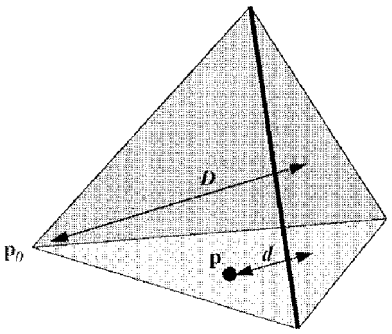


Fig. 18. The barycentric coordinate α_0 is the scaled distance d/D. The distances are d and D are signed distances to the plane containing the triangular face opposite p_0.

[30] J. Wilhelms and J. Challinger, "Direct Volume Rendering of Curvilinear Volumes," *Proc. 1990 Workshop Volume Visualization*, pp. 41-47, San Diego, Calif., 1990.
[31] M. Garrity, "Ray Tracing Irregular Volume Data," *Proc. 1990 Workshop Volume Visualization*, pp. 35-40, San Diego, Calif., 1990.
[32] C. Silva, J.S.B. Mitchell, and A.E. Kaufman, "Fast Rendering of Irregular Grids," *Proc. 1996 Volume Visualization Symp.*, pp. 15-22, Oct. 1996.
[33] C.E. Prakash and S. Manohar, "Volume Rendering of Unstructured Grids—A Voxelization Approach," *Computers & Graphics*, vol. 19, no. 5, pp. 711-726, Sept. 1995.
[34] M.B. Cox and D. Ellsworth, "Application-Controlled Demand Paging for Out-of-Core Visualization," *Proc. Visualization '97*, pp. 235-244, Oct. 1997.
[35] J. Arvo and D. Kirk, "A Survey of Ray Tracing Acceleration Techniques," *An Introduction to Ray Tracing*, A.S. Glassner, ed. San Diego, Calif.: Academic Press, 1989.
[36] D. Jevans and B. Wyvill, "Adaptive Voxel Subdivision for Ray Tracing," *Proc. Graphics Interface '89*, pp. 164-172, June 1989.
[37] K.S. Klimaszewski and T.W. Sederberg, "Faster Ray Tracing Using Adaptive Grids," *IEEE Computer Graphics and Applications*, vol. 17, no. 1, pp. 42-51, Jan.-Feb. 1997.
[38] A. Globus, "Octree Optimization," Technical Report RNR-90-011, NASA Ames Research Center, July 1990.
[39] G. Nielson and B. Hamann, "The Asymptotic Decider: Resolving the Ambiguity in Marching Cubes," *Proc. Visualization '91*, pp. 83-91, Oct. 1991.
[40] Nat'l Library of Medicine (U.S.) Board of Regents, "Electronic Imaging: Report of the Board of Regents, U.S. Dept. of Health and Human Services, Public Health Service, Nat'l Inst. of Health," NIH Publication 90-2197, 1990.
[41] B. Lorensen, "Marching Through the Visible Woman," http://www.crd.ge.com/cgi-bin/vw.pl, 1997.
[42] Y. Livnat, H. Shen, and C.R. Johnson, "A Near Optimal Isosurface Extraction Algorithm Using the Span Space," *IEEE Trans. Visualization and Computer Graphics*, vol. 2, no. 1, pp. 73-84, 1996.
[43] M.L. Brady, K.K. Jung, H.T. Nguyen, and T.P.Q. Nguyen, "Interactive Volume Navigation," *IEEE Trans. Visualization and Computer Graphics*, vol. 4, no. 3, pp. 243-256, July-Sept. 1998.
[44] J. Schwarze, "Cubic and Quartic Roots," *Graphics Gems*, A. Glassner, ed., pp. 404-407, San Diego, Calif.: Academic Press, 1990.

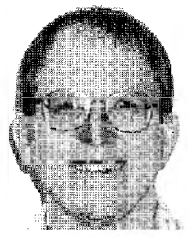
Steven Parker received a BS in electrical engineering from the University of Oklahoma in 1992. He will receive a PhD in computer science from the University of Utah in 1999. He is a research scientist in the Department of Computer Science at the University of Utah. His research focuses on problem solving 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 PhD dissertation. He was a recipient of the Computational Science Graduate Fellowship from the Department of Energy.

Michael Parker is a PhD student in computer science at the University of Utah. He received a BS in electrical engineering from the University of Oklahoma in 1995. He is interested in computer architecture and VLSI design. He has recently concluded his work on a project to reduce communication latency and overhead in clusters of workstations. He is currently involved in the architecture of an adaptable memory controller. His dissertation deals with reducing I/O and communication overhead and latency.

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

Peter-Pike Sloan has recently joined the Graphics Research Group at Microsoft as a research SDE. He was previously a student at the University of Utah and worked in the Scientific Computing and Imaging Group for Chris Johnson. He also previously worked on a 3D painting product at Parametric Technology in Salt Lake City. His interests span the spectrum of computer graphics and, most recently, he has been working/dabbling in the areas of interactive techniques, image-based rendering, surface parameterizations, and nonphotorealistic rendering.

Charles Hansen received a BS in computer science from Memphis State University in 1981 and a PhD in computer science from the University of Utah in 1987. He is an associate professor of computer science at the University of Utah. From 1997 to 1999, he was a research associate professor of computer science at Utah. From 1989 to 1997, he was a technical staff member in the Advanced Computing Laboratory (ACL) located at Los Alamos National Laboratory, where he formed and directed the visualization efforts in the ACL. He was a Bourse de Chateaubriand PostDoc Fellow at INRIA in 1987 and 1988. His research interests include large-scale scientific visualization, massively parallel processing, parallel computer graphics algorithms, 3D shape representation, and computer vision.



Peter Shirley received a BA in physics from Reed College in 1984 and a PhD in computer science from the University of Illinois at Urbana/Champaign in 1991. He is an associate professor of computer science at the University of Utah. From 1994 to 1996, he was a visiting assistant professor at the Cornell Program of Computer Graphics. From 1990 to 1994, he was an assistant professor of computer science at Indiana University. His research interests include visualization, realistic rendering, and application of visual perception research in computer graphics.