Interactive Ray Tracing for Isosurface Rendering

Steven Parker   Peter Shirley   Yarden Livnat   Charles Hansen   Peter-Pike Sloan

Computer Science Department
University of Utah
[sparker | shirley | ylivnat | hansen | ppsloan] @cs.utah.edu

Abstract

We show that it is feasible to perform interactive isosurfacing of very large rectilinear datasets 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 make it ideal for large datasets on current high-end systems. Incorporating simple optimizations, such as volume bricking and a shallow hierarchy, enables interactive rendering (i.e. 10 frames per second) of the 1GByte full resolution Visible Woman dataset on an SGI Reality Monster. The graphics capabilities of the reality monster are used only for display of the final color image.

1 Introduction

Many applications 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* [4, 10]. 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. [7, 11, 14]), but we apply it to very large datasets in an interactive setting for the first time. In the appendices, we provide details that are new to the literature that are necessary to implement this algorithm.

The basic ray-isosurface intersection method used in this paper is shown in Figure 1. 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 dataset, ray tracing should become competitive as its low time complexity overcomes its large time constant [6]. 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 [8]. 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
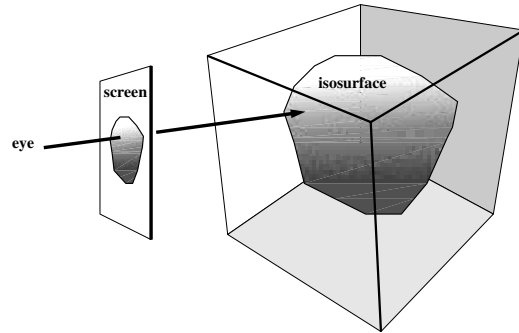

Figure 1: *A ray is intersected directly with the isosurface. No explicit surface is computed.*

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 $p = 128$). This paper demonstrates that with a few optimizations, ray tracing is *already* attractive for at least some isosurface applications.

2 The Algorithm

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. 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.

2.1 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. Analogous 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.
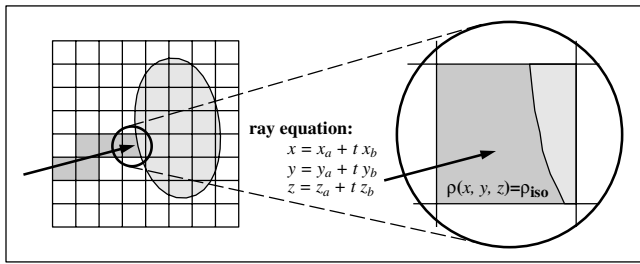
Figure 2: *The ray traverses each cell (left), and when a cell is encountered that has an isosurface in it (right), an analytic ray-isosurface intersection computation is performed.*

To find an intersection (Figure 2), 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 Appendix A.

2.2 Optimizations

For the traversal of rays through the data, we use the incremental method described by Amanatides and Woo [1]. We found that traversing the cells is the computational bottleneck for large datasets, 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 [3] (Figure 3). The details of our method for efficiently indexing cells is discussed in Appendix B.

The second is to use a multi-level spatial hierarchy to accelerate the traversal of empty cells as is shown in Figure 4. 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 [2] and the use of minimum/maximum caching has been shown to be useful [5, 16, 17]. 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 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

Figure 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.*
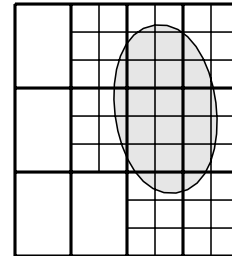

Figure 4: *With a two-level hierarchy, rays can skip empty space by traversing larger cells. A three-level hierarchy is used for the Visible Woman example.*

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 [15].

3 Results

We applied the 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 [12]. We used the computed tomography (CT) data which was acquired in 1mm slices with varying in-slice resolution. This data is composed of 1734 slices of 512x512 images at 16 bits. The complete dataset is 910MBytes. 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.
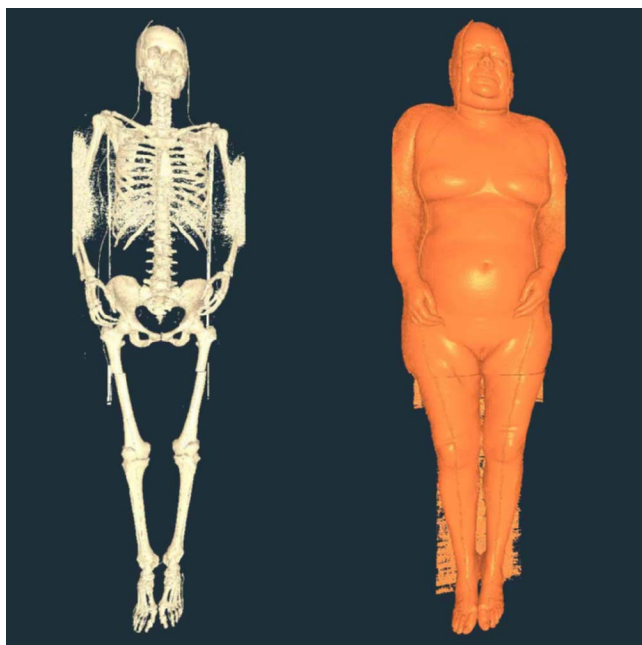
IEEE
COMPUTER
SOCIETY

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

Table 1: *Data From Ray Tracing the Visible Woman. The frames-per-second (FPS) gives the observed range for the interactively generated viewpoints on 64 CPUs.*

	View 1		View 2	
# of processors	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: *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.*

Figure 5: *Ray tracings of the skin and bone isosurfaces of the Visible Woman (see color page).*

Figure 5 shows a ray tracing for two isosurface values. Figure 6 illustrates how shadows can improve our the accuracy of our geometric perception. Table 1 shows the percentages of time spent in each of these phases, as obtained through the cycle hardware counter in SGI's speedshop. As can be seen, we achieve about 10 frames per second (FPS) interactive rates while rendering the full, nearly 1GByte, dataset.

Table 2 shows the scalability of the algorithm from 1 to 64 processors. View 2 is simpler than view 1, and thus achieves higher frame rates. 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. These efficiencies would be higher for a larger image.

Table 3 shows the improvements which were obtained through the data bricking and spatial hierarchy optimizations.

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

Table 3: *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.*

Figure 6: *A ray tracing with and without shadows (see color page).*

Data Res.	Isosurface	No. of Triangles	Time in Secs.
Full	Skin	6,650,596	1335
Full	Bone	9,914,122	1218
4x4x4	Skin	503,064	10.04
4x4x4	Bone	494,802	10.06

Table 4: *Data From GE Work on the Visible Human.*

4 Discussion

We contrast applying our algorithm to the Visible Woman data with previous work done by GE Corporate Research and Development Center [9]. Table 4 shows the time required by their algorithm. There are two data resolutions reported: the full resolution and the data strided by 4 in all dimensions. In that work, the isosurfaces from the Visible Woman data were extracted using the Marching Cubes algorithm. When the data was strided by 4 in all dimensions, a total of 494,802 polygons make up the bone surface. If the Marching Cubes was applied to the full resolution data, the isosurface requires almost 10 million polygons. On a MIPS R10000 running at 190Mhz, the isosurface extraction for the down-sampled data took a full 10.07 seconds. This does not include rendering time. Thus, the time to generate an isosurface is dominated by the geometry extraction algorithm. Our algorithm can render 64 times more data (the Visible Woman at full resolution) at roughly 10 frames per second. For the full resolution isosurfaces, one might expect a straightforward implementation on the SGI Infinite Reality graphics accelerator to render the 10M polygons in about ten seconds. With heavily optimized code it might be possible to reduce this time to one second at best. 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 non-graphics 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 the detail in the toes all at about 10 FPS.

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

5 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:
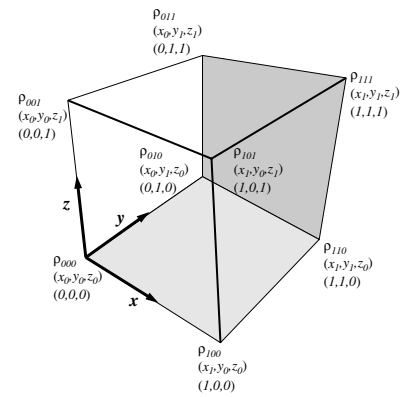

Figure 7: *The geometry for a cell. The bottom coordinates are the* (u, v, w) *values for the intermediate point.*

- Using an associated color volume as a 3D texture map for an isosurface.

- Exploration of other hierarchical methods in addition to the multilevel hierarchy described above.

- Isosurfacing of tetrahedral and hexahedral element grids.

- 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 be better [11] and the intersection routine is not the bottleneck so the degradation in performance could be acceptable.

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.

6 Acknowledgements

A Ray-Isosurface Intersection

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 Figure 7. Other cells can be 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:

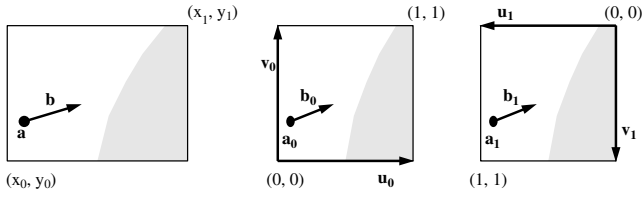$$\rho(u, v, w) \quad = \quad (1 - u)(1 - v)(1 - w)\rho_{000} + \qquad (1)$$

IEEE
COMPUTER
SOCIETY

Figure 8: *Various coordinate systems used for interpolation and intersection.*

$$(1-u)(1-v)(w)\rho_{001} +$$
$$(1-u)(v)(1-w)\rho_{010} +$$
$$(u)(1-v)(1-w)\rho_{100} +$$
$$(u)(1-v)(w)\rho_{101} +$$
$$(1-u)(v)(w)\rho_{011} +$$
$$(u)(v)(1-w)\rho_{110} +$$
$$(u)(v)(w)\rho_{111}$$

where

$$u = \frac{x - x_0}{x_1 - x_0} \tag{2}$$
$$v = \frac{y - y_0}{y_1 - y_0}$$
$$w = \frac{z - z_0}{z_1 - z_0}$$

Note that

$$1 - u = \frac{x_1 - x}{x_1 - x_0} \tag{3}$$
$$1 - v = \frac{y_1 - y}{y_1 - y_0}$$
$$1 - w = \frac{z_1 - z}{z_1 - z_0}$$

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:

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

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

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

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

$$N_x = \sum_{i,j,k=0,1} \frac{(-1)^{i+1} v_j w_k}{x_1 - x_0} \rho_{ijk}$$

$$N_y = \sum_{i,j,k=0,1} \frac{(-1)^{j+1} u_i w_k}{y_1 - y_0} \rho_{ijk}$$

$$N_z = \sum_{i,j,k=0,1} \frac{(-1)^{k+1} u_i v_j}{z_1 - z_0} \rho_{ijk}$$

Lin and Ching [7] described a method for intersecting a ray with a trilinear cell. We derive a similar result that is more tailored to our implementation.

See figure 8. 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 second ray defined by $\vec{p}_1 = \vec{a}_1 + t\vec{b}_1$. Here the rays are in the two coordinate systems (Figure 8):

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

and

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

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 $\vec{b}_1$:

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

and

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

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 $\rho(\vec{p}) = \rho_{\text{iso}}$ occurs where:

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

This can be simplified to a cubic polynomial in t:

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

where

$$A = \sum_{i,j,k=0,1} u_i^b v_i^b w_i^b \rho_{ijk}$$

$$B = \sum_{i,j,k=0,1} \left(u_i^a v_i^b w_i^b + u_i^b v_i^a w_i^b + u_i^b v_i^b w_i^a \right) \rho_{ijk}$$

$$C = \sum_{i,j,k=0,1} \left(u_i^b v_i^a w_i^a + u_i^a v_i^b w_i^a + u_i^a v_i^a w_i^b \right) \rho_{ijk}$$

$$D = -\rho_{\text{iso}} + \sum_{i,j,k=0,1} u_i^a v_i^a w_i^a \rho_{ijk}$$

The solution to a cubic polynomial is discussed the article by Schwarze [13]. 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 non-zero), and the EQN_EPS parameter was set to 1.e-30 which provided for maximum stability for large coefficients.

B   Address Management For Bricks

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 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 $40 \times 20 \times 19$ 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 float datasets.

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

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

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). For n and m which are powers of two, these operations can be converted to bit-shifts and bitwise logical operations. However, as noted above the ideal size is not a power of two. 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 9 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

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

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 added 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 isosurfacing, 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. Lookups for macrocells can be computed in the same way. 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.

References

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

[2] 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.

[3] 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.

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

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

[6] 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.

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

[8] Yarden Livnat and Charles Hansen. View dependent isosurface extraction. In *Proceedings of Visualization '98*, October 1998. This proceedings.

[9] Bill Lorensen. Marching through the visible woman. http://www.crd.ge.com/cgi-bin/vw.pl, 1997.

[10] 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.

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

[12] National Library of Medicine (U.S.) Board of Regents. Electronic imaging: Report of the board of regents. u.s. department of health and human services, public health service, national institutes of health. NIH Publication 90-2197, 1990.

[13] Jochen Schwarze. Cubic and quartic roots. In Andrew Glassner, editor, *Graphics Gems*, pages 404–407. Academic Press, San Diego, 1990.

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

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

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

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