Multiresolution Techniques for Interactive Texture-Based Volume Visualization

Eric LaMar, Bernd Hamann, and Kenneth I. Joy

Center for Image Processing and Integrated Computing

Department of Computer Science

University of California, Davis, CA 95616-8562

ABSTRACT

We present a multiresolution technique for interactive texture-based volume visualization. This method uses an adaptive scheme that renders the volume in a region-of-interest at a high resolution and the volume further away from this region at progressively lower resolutions. We use indexed texture maps, which allow for interactive modification of the opacity transfer function. Our algorithm is based on the segmentation of texture space into an octree, where the leaves of the tree define the original data and the internal nodes define lower-resolution approximations. Rendering is done adaptively by selecting high-resolution cells close to a center of attention and low-resolution cells away from this area. We limit the artifacts introduced by this method by modifying the transfer functions in the lower-resolution data sets and utilizing spherical shells as a proxy geometry. It is possible to use this technique for viewpoint-dependent renderings.

Keywords: multiresolution rendering, volume visualization, hardware texture, transfer function

1. INTRODUCTION

The capabilities of computing technology have steadily increased for more than four decades and continue to increase rapidly. These increased computing capabilities have enabled applications to scale accordingly in overall throughput and resulting data set sizes. However, current visualization techniques break down when operating in this environment due to the massive size of today's data sets. New techniques are necessary to provide exploration of very large and often multidimensional data sets.

In this paper, we combine hardware-assisted texture mapping and color table lookup with multiresolution methods for rendering large volumetric data sets. Texture mapping is substantially faster than software-based approaches, and color lookup tables allow for easy manipulation of transfer functions. The multiresolution principle assigns priorities to different regions of the volume and renders "high-priority regions" with highest accuracy, while "low-priority" regions are rendered with progressively less accuracy and progressively faster.

We use an octree to decompose texture space and produce several coarser levels of an original data set. Each level is associated with a level in the octree, and each level is half the resolution of the next level. The leaf nodes are associated with the original resolution and the root node with the coarsest resolution. Interior nodes are created by subsampling a node's eight child nodes. Each value in the texture map is an index, not an RGBA tuple; the transfer function is applied as the texture map is transferred to texture memory.

Rendering a volume involves traversing the octree and applying a selection filter to each node. Three results are possible: (1) The node (and its children) are skipped entirely; (2) the node is skipped, but its children are visited; or (3) the node is rendered, and the children are skipped. The selected nodes are then sorted in back-to-front order. Finally, for each node, the algorithm builds and transfers the color lookup table, transfer the texture map, and render the proxy geometry.

Section 2 contains a survey of related work; section 3 discusses data issues for the multiresolution representation of textures, and Section 4 addresses the rendering of these textures. Section 5 discusses issues of static and indexed texture maps. Section 6 shows results for two data sets and lists performance results. Conclusions and future work are presented in Section 7.

Further author information: (Send correspondence to {lamar,hamann,joy}@cs.ucdavis.edu)

2. RELATED WORK

High-performance computer graphics systems are evolving rapidly. Silicon Graphics, Inc. (SGI) has been a primary developer of this rendering technology, introducing the RealityEngine[1] graphics system in 1994 and the InfiniteReality[12] graphics system in 1998. SGI has also extended the graphics library OpenGL[11,14] to take advantage of this hardware. These systems were among the first to support hardware-based rendering using solid textures.

Cabral et al.[2] show that volume rendering and reconstruction integrals are generalizations of the Radon and inverse Radon transforms. They show that the Radon and inverse Radon transforms have similar mathematical forms, and, by developing this relationship, show that both volume rendering and volume reconstruction can be implemented with hardware-accelerated textures. Their algorithms execute faster than traditional software-based approaches. Cullip and Neumann[3] discuss general implementation issues for hardware textures. Their work illustrates the superiority of viewport- versus object-aligned sampling planes.

Wilson et al.[18] and Van Gelder and Kim[16] develop the mathematical foundation for generating texture coordinates. Van Gelder and Kim[16] also introduce a quantized gradient method for shading. Here, a triangulated sphere describes quantized normals which, when coupled with a quantized set of material values, allows the construction of a lookup table. For each new scene and texture block, the current viewing and lighting parameters are applied to the lookup table, and the lookup table is applied to the texture map as it is transferred to the texture subsystem. They report interactive rates, both for orthographic and perspective projections. However, low-gradient regions show traditional quantization artifacts.

Westerman et al.[17] show how to visualize isosurfaces resulting from rectilinear and unstructured grids. They use fragment testing to draw only those pixels that have a density value above a given threshold. Rectilinear grids are rendered by solid-texturing, which is known to be much faster than the unstructured grid method. They also demonstrate how to shade the texture-based isosurfaces with a technique that performs the shading as the texture map is transferred to the texturing subsystem.

Grzeszczuk et al.[5] enumerate most methods for using hardware-accelerated texturing to provide interactive volume visualization. They also introduce a library for texture-based rendering called *Volumizer*.[4]

Massively parallel computers have been used to provide interactive volume visualization and isosurface extraction.[6,7,8,13] Both ray-tracers and marching-cubes algorithms have been implemented and both are parallelizable. The overhead of data distribution and image composition is very high and requires careful partitioning and tuning.

Shen et al.[15] discuss a temporally-based multiresolution scheme for volume visualization of unsteady data sets. They use image caching that allows fast rendering but the caching is not viewpoint independent so images must be recomputed for new viewpoints. LaMar et al.[9] introduce multiresolution techniques on which this work is based. They show that multiresolution techniques, applied to large data sets and volume rendering, are a reasonable approach to reducing both rendering time and amount of data rendered.

Our method differs from these prior approaches in the sense that we allow adaptive rendering of a volume and interactive manipulation of the transfer function. Prior algorithms assume that the data is "uniformly complex" and "uniformly important"; they also assume that a "standard" transfer function is sufficient or that memory is available to store several transformed data sets. This is not the case, for example, in an immersive environment, where a user is exploring unknown data: first, the data closer to the viewer has more visual importance than data far away; second, a viewer cannot know in advance what transfer function will work, so it must be possible to experiment. Also, quality should be a "tunable" parameter: if a graphics supercomputer is not available or a user just wishes to quickly browse a data set, then the user will be satisfied with a poorer rendering quality.

3. GENERATING THE TEXTURE HIERARCHY

In hardware texture algorithms, linear interpolation is used to interpolate the values at the centers of adjacent pixels. A larger texture can be broken into a set of smaller textures or tiles, where interior edge pixels are duplicated between adjacent tiles. This technique is called "bricking".[5]

Figure 1 shows a one-dimensional texture hierarchy of four levels. The top level, level 0, is the original texture, broken into eight tiles, level 1 contains four tiles at half of the original resolution, and so on. The dashed vertical lines on either side show the domain of the texture function over the hierarchy. Arrows show the parent-child relationship of the hierarchy, defining a binary tree, rooted at the coarsest tile, level 3. The bold vertical line denotes a point of

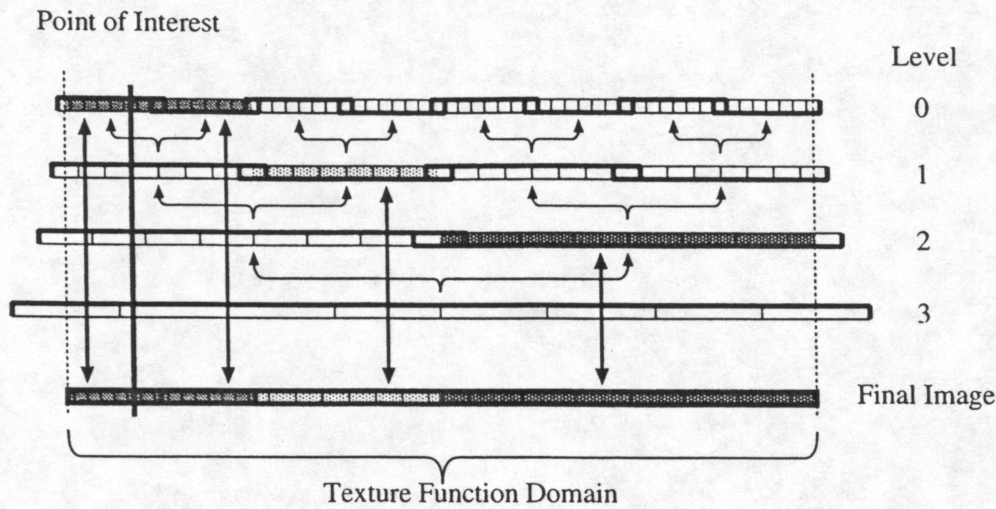

Figure 1. Selecting from a texture hierarchy of four levels. Level 0 is the original texture, broken into eight tiles. The dashed lines show the domain of the texture function over the hierarchy. The bold vertical line represents a point of interest p. Tile selection depends on the width of the tile and the distance from the point.

interest, p, and tiles are selected when the distance from p to the center of the tile is greater than the width of the tile. One starts with the root tile and performs this selection until all tiles meet this criterion, or no smaller tiles exist*. The bold vertical arrows show tiles selected and correspondence in the final image.

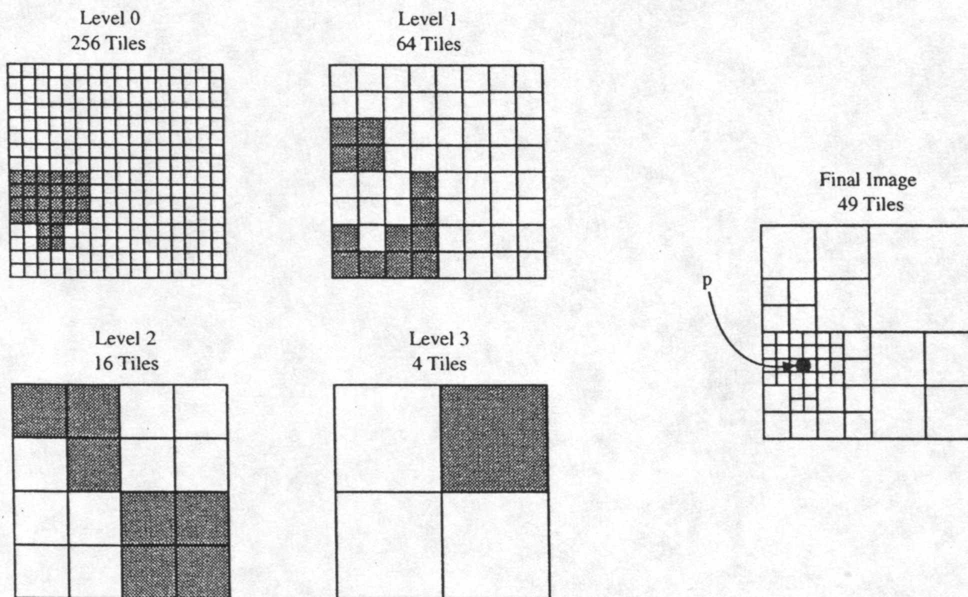

Figure 2. Selecting tiles in two dimensions from a texture hierarchy of five levels (Level 4 is not shown). Given the point p, tiles are selected when the distance from the center of the tile to p is greater than the length of the diagonal of the tile. The selected tiles are shaded.

Figure 2 shows a two-dimensional quadtree example. The original texture, level 0, has 256 tiles. The darker regions in each level show the portion of that level used to approximate the full image. The selection method is similar to the one-dimensional case: a node is selected when the distance from the center of the node to the point p is greater than the length of the diagonal of the node. The original texture, divided into 256 tiles, requires 256 time units to render. The adaptive rendering scheme requires 49 time units, which translates to a speed-up factor of approximately five. This scheme can easily be used for three-dimensional textures using an octree.

How much memory is "wasted" by breaking a volume into tiles? The waste is generated by the outer layer of

*This is the case on the left side of Figure 1.

voxels, which is shared by adjacent tiles. If a brick has size n (n^3 voxels) and is surrounded by a half-voxel layer of duplicate voxels, the effective size of a brick is $n-1$, and there are $n^3-(n-1)^3$ "waste voxels". The waste relative to the tile size is $O(n^2/n^3)=O(n^{-1})$, which means that the relative waste decreases as the tile size increases†. For example, if we choose a tile size of 64^3, the tile contains 262, 144 voxels, 250,047 effective voxels and 12,097 waste voxels.

4. RENDERING

The rendering phase is divided into three steps: (1) selecting tiles to render; (2) sorting the tiles; and (3) rendering the tiles using a proxy geometry.

4.1. Selecting Tiles

The first rendering step determines which tiles will be rendered. The general filtering logic starts at the root tile and performs a depth-first traversal of the octree. For each tile, we evaluate a selection filter, which returns one of three possible responses:

- Ignore this tile and all of its children. This response is used to cull the tree. For example, if a tile is not in the view frustum, then we can ignore the tile and its children.
- The tile satisfies all criteria. In this case, the tile is rendered, and its children ignored.
- The tile does not satisfy the criteria. In this case, the children of the tile are checked.

Our primary selection filter is based on one of these two criteria:

- **Field-of-View (FOV) criterion.** Selects a tile if it intersects the view frustum and the projected angle of the tile is less than half the view frustum's field-of-view angle.
- **Cone criterion.** Selects the highest-resolution tiles within the viewing frustum. Its primarily use is to determine the speedup factor of the Field-Of-View criterion.

Tiles are sorted and composited in back-to-front order. We order tiles with respect to a view direction such that, when drawn in this order, no tile is drawn behind a rendered tile. The order is fixed for the entire tree for orthogonal projections and has to be computed just once for each new rendering.[5] For perspective projections, the order must be computed at each node.

4.2. Proxy Geometries

Texture-based volume visualization requires proxy geometries on which the texture is rendered: we use *viewpoint-centered spherical shells* (VCSSs) – finely tessellated concentric spheres surrounding the viewpoint and culled to the view frustum. VCSSs are implemented with three-dimensional textures and using concentric spherical shells centered at the viewpoint, which are culled to the view frustum. This technique does not produce artifacts under perspective projections, but it is slower than traditional techniques.[9] Figure 3(a) illustrates multiresolution VCSSs. In Figure 3(b), the viewpoint is on the left-hand side, almost touching the volume. The sample interval is exaggerated to show the structure – one shell every two voxels. Using this approach, one can achieve continuity across tile faces.

4.3. Preserving Visual Properties

When rendering tiles at different levels of the hierarchy, the opacity properties of the tiles are different. Classical rendering algorithms[10] depend on using the same sampling‡ along rays for each pixel. But in the context of a multiresolution format, the volume is sampled in different ways, and at varying resolutions. To preserve lighting and opacity properties between tiles of different resolutions, we must modify the transfer functions for those tiles generated by subsampling .

†The total waste still increases.

‡Same number of samples and same spacing between samples.

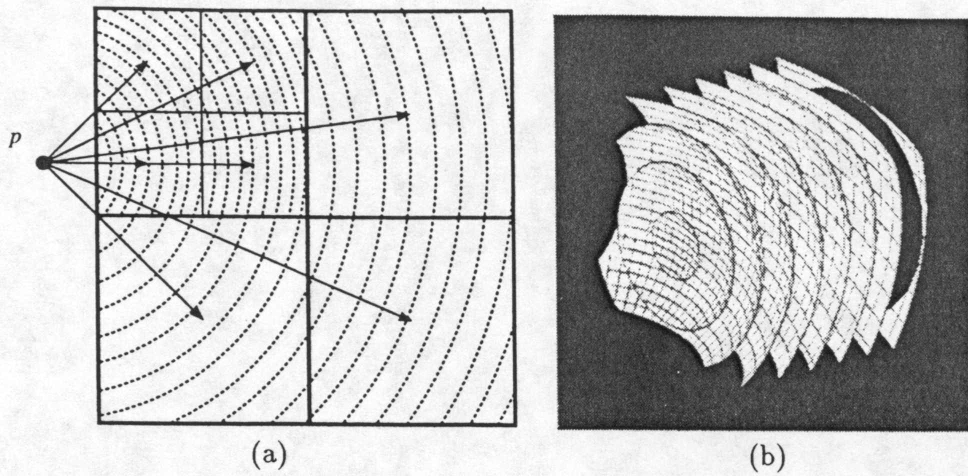

Figure 3. Viewpoint-Centered Spherical Shells (VCSS)

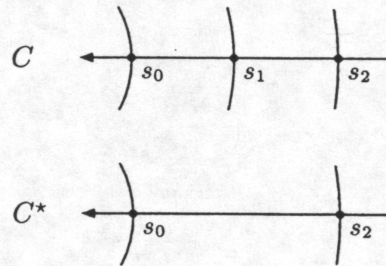

Figure 4. When space is sampled at two different resolutions, the colors C and $C^\star$ should be the same.

Figure 4 shows an example where a texture is sampled with spherical shells at two different resolutions – one is half the resolution of the other. Each sample s_i has an associated color c_i and an opacity value α_i. The light emitted by s_i is a function of the incoming light and the color and opacity properties of the sample itself. According to Levoy,[10] the color C resulting from the high-resolution samples $s_0, s_1, s_2, \ldots$ is

$$C = \alpha_0 c_0 + (1 - \alpha_0) C_1, \tag{1}$$

where C_1 is the incoming color from samples $s_1, s_2, \ldots$, *i.e.*,

$$C_1 = \alpha_1 c_1 + (1 - \alpha_1) C_2, \tag{2}$$

where C_2 is the incoming color from samples $s_2, s_3, \ldots$. For the low-resolution samples $s_0, s_2, s_4, \ldots$, the color $C^\star$ is given by

$$C^\star = \alpha_0^\star c_0 + (1 - \alpha_0^\star) C_2^\star, \tag{3}$$

where $C_2^\star$ is the color calculated as a result of the samples $s_4, s_6, \ldots$

By considering only the first three samples s_0, s_1, and s_2, the colors to be compared are C and $C^\star$, where

$$\begin{aligned} C = &\quad \alpha_0 c_0 + (1 - \alpha_0)\alpha_1 C_1 + (1 - \alpha_0)(1 - \alpha_1) C_2 \text{ and} \\ C^\star = &\quad \alpha_0^\star c_0 + (1 - \alpha_0^\star) C_2^\star. \end{aligned} \tag{4}$$

These two colors, in general, are different. We now describe a method to correct for this.

The accumulated opacities A and $A^\star$, are given by considering only samples s_0, s_1, and s_2,

$$\begin{aligned} A = &\quad \alpha_0 + (1 - \alpha_0)\alpha_1 + (1 - \alpha_0)(1 - \alpha_1) A_2 \text{ and} \\ A^\star = &\quad \alpha_0^\star + (1 - \alpha_0^\star) A_2^\star. \end{aligned} \tag{5}$$

Assuming that the accumulated opacities are equal at the even-indexed samples, it follows that $A_2 = A_2^\star$. The requirement $A = A^\star$ implies that

$$\alpha_0 + (1-\alpha_0)\alpha_1 + (1-\alpha_0)(1-\alpha_1)A_2 = \alpha_0^\star + (1-\alpha_0^\star)A_2. \tag{6}$$

Solving this equation for $\alpha_0^\star$, one obtains

$$\begin{aligned}\alpha_0^\star &= \alpha_0 + (1-\alpha_0)\alpha_1 \\ &= 1-(1-\alpha_0)(1-\alpha_1).\end{aligned} \tag{7}$$

By assuming that $\alpha_1 = \alpha_0 + \epsilon$, where ϵ is a very small number, we obtain the equation

$$\begin{aligned}\alpha_0^\star &= 1-(1-\alpha_0)(1-\alpha_0) + \epsilon(1-\alpha_0), \\ &= 1-(1-\alpha_0)^2 + O(\epsilon).\end{aligned} \tag{8}$$

Therefore, to correct for the unequal sampling, we modify the transfer function of the lower-resolution texture by using the value

$$\alpha^\star = 1-(1-\alpha)^2. \tag{9}$$

This correction value reduces the artifacts between the texture bricks. This formula is used when generating the texture lookup table from the transfer function.

5. STATIC AND INDEX TEXTURE MAPS

Static texture maps are "pre-shaded" texture maps. The data is transformed by the transfer function, from a byte to an RGBA tuple, then stored in a texture map. Opacity correction is also applied at this step. This technique is useful since one can freely choose the transfer function. For example, opacity may be a function of the gradient. Transferring a static texture to the texture subsystem is very fast as no further transformations are necessary. However, it is very slow since the transfer function is implemented in software, which requires that *all* data be touched for each new transfer function.[§] Lastly, static texture maps require two or four times as much memory as an index texture map. The static texture map uses a byte for each RGBA value, or four bytes per texel, while index texture maps use one or two bytes per texel. For these reasons, this technique is inappropriate for situations where one may use a large number of different transfer functions.

Index texture maps are stored directly in the texture maps, interpreted as an index. Opacity correction is deferred until building the lookup table. Though performed in software, this step has very little overhead as it is performed once and mostly involves moving memory. Transferring an index texture map to the texture subsystem is more involved. A texture lookup table is built and transferred to the texture subsystem; the texture lookup table translates the index for a value for each of the RGBA channels. This is performed in hardware and is very fast. The texture lookup table is quite small, 256 to 65536 entries[¶], and can be updated easily and quickly in software. Subsequently, the texture map is transferred and the texture subsystem translates that index to an RGBA tuple for each texel.

6. RESULTS

We have implemented our algorithm and applied it to several complex data sets. All data sets were rendered on an SGI Onyx2 computer system with 512MB of main memory, 2GB of swap, and 16MB of texture memory, using a single 195Mz R10000 processor.

Figures 5a to 5d show an Equine Metacarpus data set. This is a CT scan data set and consists of $108^2 \times 126$ voxels. Each tile contains 16^3 texels. The number of tiles for levels 0 to 3 are: 448, 64, 8, and 1; for a total of 521 tiles. The memory requirements for static textures, with $16^3 \times 4 = 16384$ bytes per tile, is 8 MB. The memory

[§]The results can be cached, for each transfer function, but at a very high memory cost.

[¶]At 4 bytes per entry, this results in 1024 bytes to 256KB; compared to a 1024^3 static texture map, which requires 4GB.

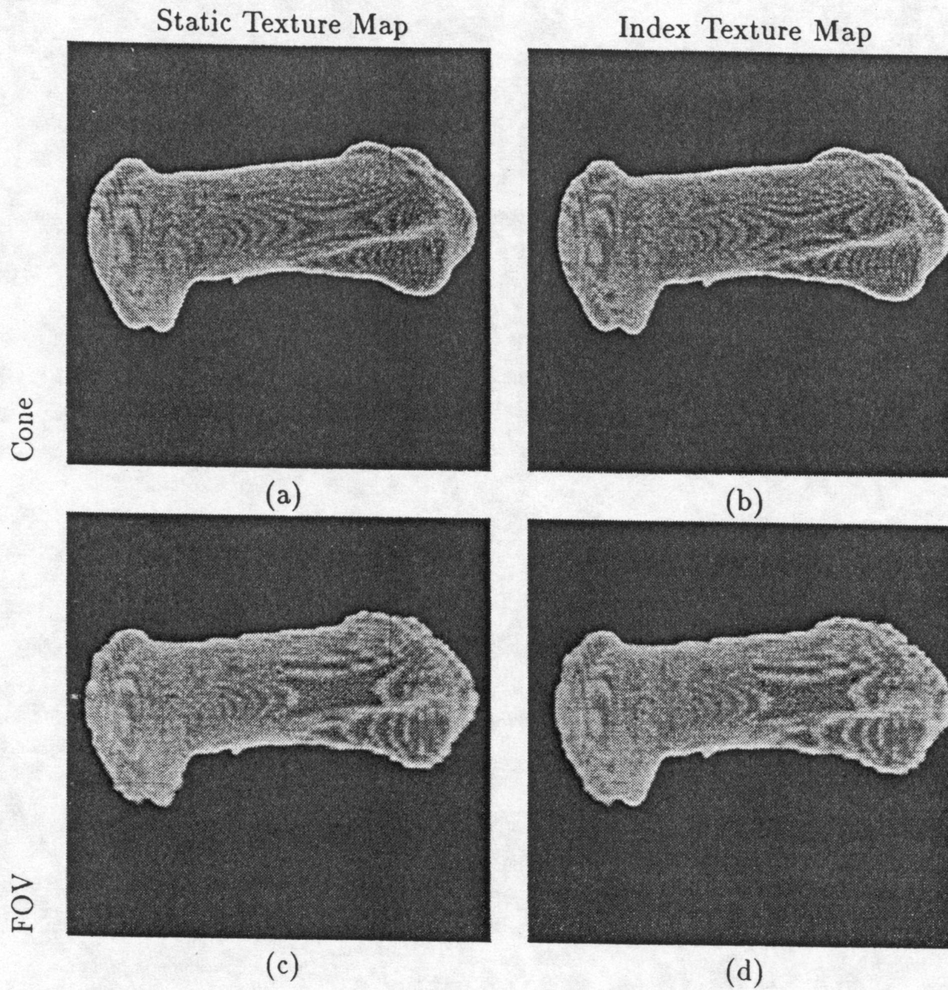

Figure 5. *Equine Metacarpus* data set. Image size is 500^2 pixels.

	Static Texture Maps		Index Texture Maps	
Data Set Size	$108^2 \times 126$			
Tile Size	16^2 texels			
Tiles at Level 0	448			
Tile Time	25.3 sec.		5.1 sec.	
Memory Used	8 MB		4 MB	
Filter	Cone	FOV	Cone	FOV
Number Of Tiles	432	148	433	150
Rendering Time	10.72 sec.	5.1 sec.	9.94 sec.	4.93 sec.

Table 1. Rendering statistics for the *Equine Metacarpus* data set. Times are in seconds.

requirements for index textures, with $16^3 \times 2 = 8192$ bytes per tile, is 4 MB. Images 5(a) and 5(c) were rendered using static texture maps, while images 5(b) and 5(d) were rendered using index texture maps. Images 5(a) and 5(b) use tiles from the Cone filter; images 5(c) and 5(d) use tiles from the FOV filter. Note that while the rendering times are roughly the same, index texture maps require 5.1 seconds to generate all tiles ("Tile Time"), and is done only once. Static texture maps require 25.3 second to generate all tiles, and this must be done every time the transfer function is changed. Statistics concerning the rendering times for this data set are given in Table 1.

Data Set Size	$500^2 \times 768$	
Tile Size	64^3 texels	
Tiles at Level 0	1053	
Tile Time	1049 sec.	
Memory Used	637 MB	
Filter	Cone	FOV
Number of Tiles	593	47
Rendering Time	25.4 sec.	2.47 sec.

Table 2. Rendering statistics for the *Raleigh-Taylor instability* data set. Times are in seconds.

Figures 6a to 6h show a Raleigh-Taylor instability data set. This data set is a single time step from a simulation and consists of $500^2 \times 768$ voxels. Each tile contains 64^3 texels. The number of tiles for levels 0 to 4 are: 1053, 175, 36, 8, and 1; for a total of 1273 tiles. For index textures, this requires $1273 \times 64^3 \times 2 = 637$ MB of memory. Images 6(a), (c), (e), and (g) are rendered using tiles selected by the Cone filter, requiring 593 tiles. Images 6(b), (d), (f), and (h) are rendered using tiles selected by the FOV filter, requiring only 47 tiles. The while grid lines show the tile boundaries. Images 6(c), (e), and (g) and images 6(d), (f), and (h) are rendered in perspective. Images 6(a) and 6(b) are rendered orthographically. The yellow pyramid is the viewing frustum of the perspective projections, and the white and blue boxes show the tiles boundaries. The fixed resolution images, 6(c), (e), and (g), each require approximately 25 second to render, while the multiresolution images, 6(d), (f), and (h), each require about 2.5 seconds to render. Statistics concerning rendering times for this data set are given in Table 2.

We were not able to render a static texture map version of the *Raleigh-Taylor instability* data set. However, the Equine Metacarpus data shows the timing issues. We should also note that the *Raleigh-Taylor instability* data set "Tile Time" also includes a reasonable amount of time lost to swapping. However, the render time includes the time to change the transfer function. If we extrapolate the time (from the Equine Metacarpus Static vs. Index "Tile Time") to regenerate the tiles for a static texture map version, the "Tile Time" should be about 5245 seconds (~88 minutes); this is clearly not a reasonable amount of time if one wishes to experiment with the transfer function.

7. CONCLUSIONS

We have described a new method for building and rendering a multiresolution texture hierarchy approximation for very large data sets where the transfer function can be modified interactively. The approach utilizes a "bricking" strategy, where the displayed bricks are selected from an octree representation, and index texture maps, where the transfer function applied with a color lookup table. The color lookup table is significantly faster than prior methods and provides a better basis for exploring very large data sets. Despite the fact that our overall system is limited by the amount of available texture memory, the algorithm produces very good results, and we expect that this approach will have a major impact on the huge volumetric data sets that are currently encountered in numerous applications. Future work involves removing the artifacts between tiles of different resolution and rendering pre-segmented biological data sets.

ACKNOWLEDGMENTS

This work was supported by: the National Science Foundation under contract ACI 9624034 (CAREER Award) and through the National Partnership for Advanced Computational Infrastructure (NPACI), and through a Lage Scientific and Software Data Set Visualization (LSSDV) award; the Office of Naval Research under contract N00014-97-1-0222; the Army Research Office under contract ARO 36598-MA-RIP; the NASA Ames Research Center through an NRA award under contract NAG2-1216; the Lawrence Livermore National Laboratory under ASCI ASAP Level-2

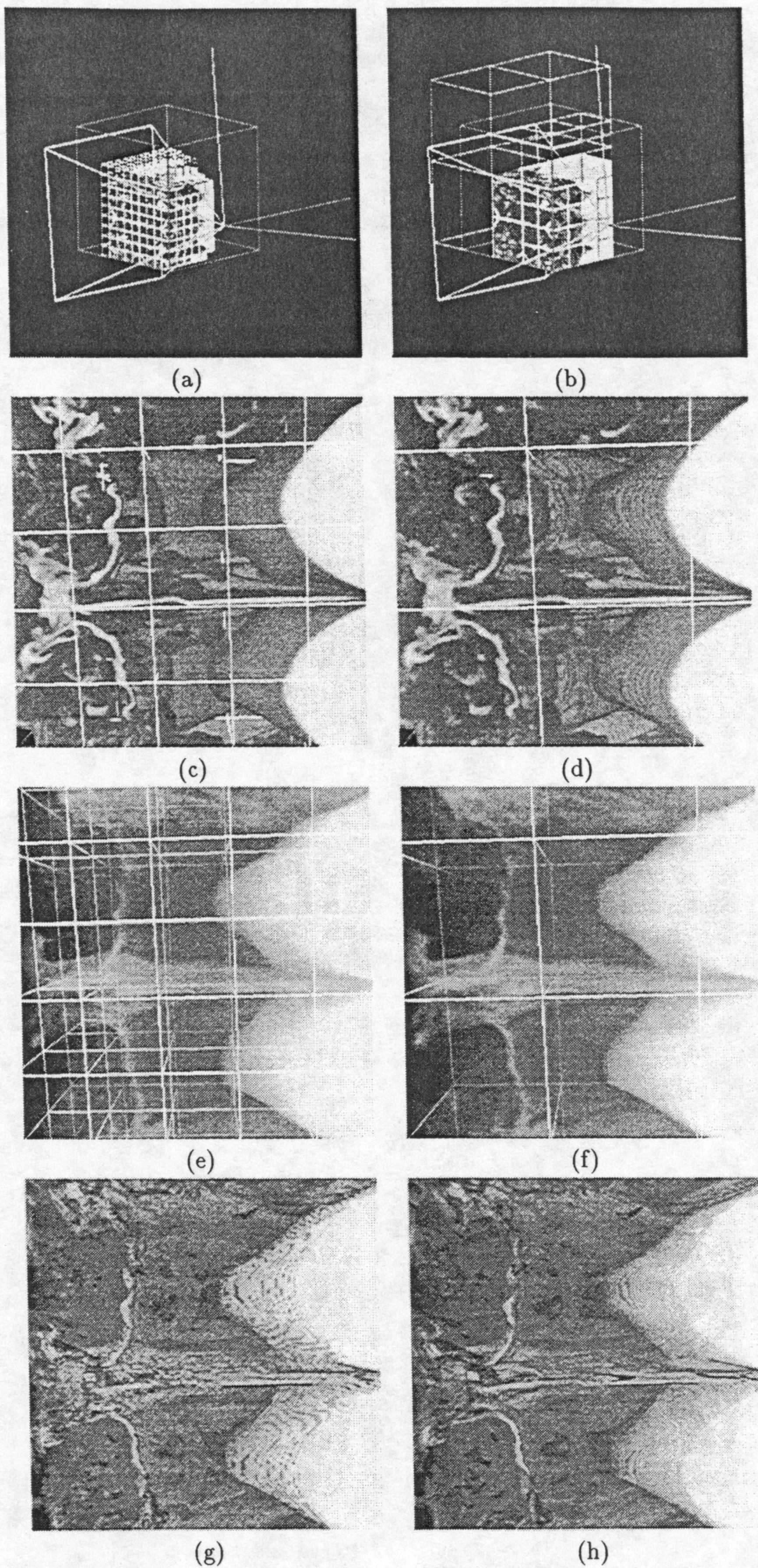

Figure 6. *Raleigh-Taylor instability* data set. Images (a), (c), (e), and (g) rendered with tiles selected by the **Cone** filter. Images (b), (d), (f), and (h) are rendered with tiles selected by the **FOV** filter. Image size is 500^2 pixels. The lower three rows show three different transfer functions. Images(a) and (b) show the viewing frustum, the yellow pyramid, with respect to the data.

Memorandum Agreement B347878 and under Memorandum Agreement B503159; and the North Atlantic Treaty Organization (NATO) under contract CRG.971628 awarded to the University of California, Davis.

We also acknowledge the support of ALSTOM Schilling Robotics, Chevron, General Atomics, Silicon Graphics, Inc., and ST Microelectronics, Inc. We thank the members of the Visualization Thrust at the Center for Image Processing and Integrated Computing (CIPIC) at the University of California, Davis. The *Equine Metacarpus* data set was provided by Dr. C.M. Les of the JD Wheat Veterinary Orthopedic Research Laboratory at the University of California, Davis. The *Raleigh-Taylor instability* data set was provided by Mark Duchaineau of Lawrence Livermore National Laboratory.

REFERENCES

1. Kurt Akeley. RealityEngine Graphics. In *Proceedings of Siggraph 93*, pages 109–116. ACM, August 1993.
2. Brian Cabral, Nancy Cam, and Jim Foran. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In *1994 Symposium on Volume Visualization*, pages 91–98. ACM SIGGRAPH, October 1994.
3. Timothy J. Cullip and Ulrich Neumann. Accelerating Volume Reconstruction With 3D Texture Hardware. Technical Report TR93-027, Department of Computer Science, University of North Carolina - Chapel Hill, May 1 1994.
4. George Eckel. *OpenGL Volumizer Programmer's Guide.* Silicon Graphics Computer Systems, Mountain View, CA,USA, 1998.
5. Robert Grzeszczuk, Chris Henn, and Roni Yagel. *SIGGRAPH '98 "Advanced Geometric Techniques for Ray Casting Volumes" course notes.* ACM, July 1998.
6. Charles D. Hansen and Paul Hinker. Isosurface extraction SIMD architectures. In *IEEE Visualization 92.* IEEE, October 1992.
7. Charles D. Hansen, Michael Krogh, James Painter, Guillaume Colin de Verdiere, and Roy Troutman. Binary-swap volumetric rendering on the T3D. In *Cray Users Group Conference*, Denver, Co., March 1995.
8. Charles D. Hansen, Michael Krogh, and William White. Massively parallel visualization: Parallel rendering. In *Proceedings of the 27th Conference on Parallel Processing for Scientific Computing*, pages 790–795, February 1995.
9. Eric LaMar, Bernd Hamann, and Kenneth I. Joy. Multi-Resolution Techniques for Interactive Hardware Texturing-based Volume Visualization. In *IEEE Visualization 99*, pages 355–362. IEEE, October 1999.
10. Marc Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, February 1987.
11. Tom McReynolds and Davis Blythe. *SIGGRAPH '98 "Advanced Graphics Programming Techniques Using OpenGL" course notes.* ACM, July 1998.
12. John S. Montrym, Daniel R. Baum, David L. Dignam, and Christopher J. Migdal. Infinite Reality: a Real-Time Graphics System. In *Proceedings of Siggraph 97*, pages 293–302. ACM, August 1997.
13. Frank A. Ortega, Charles D. Hansen, and James P. Ahrens. Fast Data Parallel Polygon Rendering. In *Supercomputing 93*, pages 709–718. IEEE Computer Society Press, November 1993.
14. Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 1.2).* Silicon Graphics Computer Systems, Mountain View, CA,USA, 1998.
15. Han-Wei Shen, Ling-Jan Chiang, and Kwan-Liu Ma. A Fast Volume Rendering Algorithm for Time-Varying Fields using a Time-Space Partitioning (TSP) Tree. In *IEEE Visualization 99*, pages 371–378. IEEE, October 1999.
16. Allen Van Gelder and Kwansik Kim. Direct Volume Rendering with Shading via Three-Dimensional Textures. In *1996 Volume Visualization Symposium*, pages 23–30. IEEE, October 1996. ISBN 0-89791-741-3.
17. Rüdiger Westermann and Thomas Ertl. Efficiently Using Graphics Hardware In Volume Rendering Applications. In *Proceedings of Siggraph 98*, pages 169–177. ACM, 19-24July 1998.
18. Orion Wilson, Allen Van Gelder, and Jane Wilhelms. Direct Volume Rendering via 3D Textures. Technical Report UCSC-CRL-94-19, University of California, Santa Cruz, June 29 1994.