Volume Rendering for Curvilinear and Unstructured Grids

Nelson Max, Peter Williams, Claudio Silva, and Richard Cook
Lawrence Livermore National Laboratory
{*max2, plw, rcook*}*@llnl.gov, csilva@cse.ogi.edu*

Abstract

We discuss two volume rendering methods developed at Lawrence Livermore National Laboratory. The first, cell projection, renders the polygons in the projection of each cell. It requires a global visibility sort in order to composite the cells in back to front order, and we discuss several different algorithms for this sort. The second method uses regularly spaced slice planes perpendicular to the X, Y, or Z axes, which slice the cells into polygons. Both methods are supplemented with anti-aliasing techniques to deal with small cells that might fall between pixel samples or slice planes, and both have been parallelized.

1. Introduction

In volume rendering for scientific visualization, a 3D scalar field is represented by a cloud of very small glowing particles, whose color and density depend on the scalar variable. The rendering then calculates how this cloud would appear in a given view (see [1]).

If the scalar field is sampled on a cubical or rectilinear grid, graphics hardware can render the volume in real time, using texture mapping [2]. However, at Lawrence Livermore National Laboratory we need to render volumes from physics simulations on curvilinear grids, or on unstructured finite element grids with mixed element types, currently with the topology of tetrahedra, cubes, triangular prisms, and square pyramids. Over the past 13 years, we have worked on two methods for directly rendering the cells of such general grids, without resampling the data onto a rectilinear grid.

The cell projection method divides the projection of each cell into polygons, and uses graphics hardware to draw them. This was first done for tetrahedra by Shirley and Tuchman [3]. Our extension to general polyhedra is described in section 2 below. In order for the hardware to composite these polygons correctly, the cell must sorted in back to front order, and section 3 describes several method for doing this. In our general grids, cells may have non-planar quadrilateral faces, which may cause problems for both the sorting and the polygon scan conversion. Section 4 describes a solution which selectively subdivides such cells into tetrahedra, only for views where they would cause problems.

The second method, described in section 5, involves slicing the cells into polygons by a collection of closely spaced parallel planes, as proposed by Yagel *et al*. [4], and rendering these polygons. In section 6, we describe how we slice only those cells above a specific volume threshold, and render the others, which might lie between the slice planes or between pixel samples in these planes, using the anti-aliased ellipsoidal splats of Zwicker *et al*. [5].

Large datasets require parallel processing for interactive visualization, and in section 7 we describe how we have parallelized parts of our two rendering methods.

2. Polyhedron Projection

The projections of the edges of a convex polyhedral cell divide the image plane up into polygons. Each of these polygons lies within the projection of a single front-facing face and a single back-facing face of the cell. Therefore, for an orthogonal projection, the length l of the viewing ray segment within the cell varies linearly across the polygon, and can be interpolated linearly by the hardware.

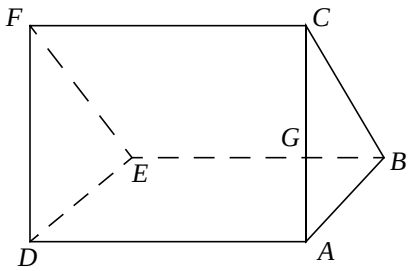

Figure 1. Projection of a triangular prism.

At the "thin" vertices A, *B*, *C*, *D*, and *F* in figure 1, the segment length, and therefore the compositing opacity, is zero, and the color is the one associated with the scalar value at the vertex. At the thick vertices, the length of the ray segment must be calculated geometrically, by finding the segment endpoints on the front and back faces which project to the vertex. We assume initially that the scalar field varies linearly across the cell, and that the *transfer*

functions that specify the particle color and density as a function of the scaler value are also linear. (See Max *et al*. [6] for a discussion of these assumptions.) Then the color at the ray segment endpoints can be obtained by interpolation across polygons (for the rear endpoint at *E*) or edges (for the endpoints of *G*). The densities κ for the cloud particles can similarly be interpolated. If κ_{avg} is average of κ at the segment endpoints, the opacity α for compositing the segment is

$$\alpha(\tau, l) = 1. - \exp(-\kappa_{avg} l) \quad \textbf{(1)}$$

as explained in [1]. Shirley and Tuchman [3] took the color for the thick vertex as the average of the colors at the segment endpoints. We have optionally used an analytic integration formula, derived in [7] and [8], which takes longer but more accurately accounts for the fact that the particles at the front of the segment partially occlude those at the rear. The color is then linearly interpolated across the polygon by the Gouraud shading hardware. This is only an approximation, but the results look good.

Shirley and Tuchman [3] also linearly interpolate the compositing opacity α across the polygons in hardware. This is a more serious approximation, because of the non-linear exponential in equation (1). As shown in figure 2, it can cause derivative discontinuities in α, which result in disturbing Mach bands in the image.

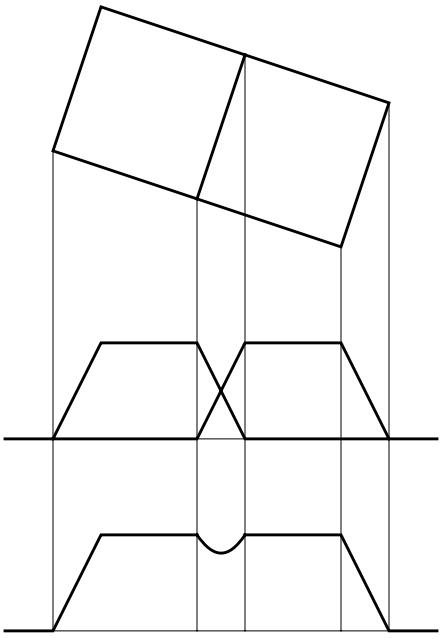

Figure 2. Top: cross section for a scan plane through two adjacent cubes. Middle: two curves linearly interpolating the α values for the two cubes. Bottom: the results of compositing these two curves.

To avoid this problem, we use a 2D texture table for alpha. The two texture coordinates are κ_{avg} and l, which do vary linearly across the polygons, and therefore can be correctly interpolated in the hardware. The texture is loaded with the values from equation (1), and then returns the correct exponential per pixel during rendering.

Shirley and Tuchman [3] calculated the topology for the projection of a tetrahedron into from one to four triangles using a collection of dot and cross products involving the vertices and the viewpoints to distinguish the different cases. Recently, these computations have been accomplished in hardware, using an advanced programable vertex “shader” engine [9].

For more general cells, we us an incremental method to build the image plane subdivision. The projected edges of the cell are added one by one to the subdivision, starting with one infinite region with no boundary. An edge may create a new boundary cycle (as the first one does), create a new polygon by closing a cycle, or subdivide existing polygons and edges by slicing across them.

This incremental method is slow, and not completely robust, since it uses floating point computations to determine the topology of the projection. In curvilinear grids, which are common in our simulations, the topology of the projections of most of the cells will agree with one of the three non-degenerate perspective projection topologies for a cube. We use a collection of tests involving the vertex projections to identify these cases, and look up the standard subdivisions for them. This speeds up our projections, even though the overhead of these tests is added to the cost of doing the general incremental subdivision when the tests fail. Schussman and Max [10] give a much faster method for doing the tests and look-up for the case of a cubical grid in perspective.

3. Visibility Sorting

For correct back-to-front hardware compositing, the cells must be *visibility sorted* into an ordered list. By definition, this means that if cell *A* partially occludes cell *B* from a particular viewpoint *V*, cell *B* must come before cell *A* in the list.

The Newell, Newell, and Sancha sort [11] for polygons was extended to the case of polyhedra in [12]. The cells are initially sorted from back to front by the depth coordinate of their rear-most vertex. The rearmost cell *A* from the list is then tested against all the other cells. If it does not occlude any of them, it is removed from the initial list and placed on the output list. On the other hand, if a cell *B* is found that is occluded by cell *A*, *B* is moved to the rearmost position on the initial list and the testing proceeds with it instead. When cell *B* is moved, it is marked as having been moved, and if there is a second attempt to move it, it is part of a *visibility cycle* of cells, each of

which occludes the next cell in the cycle. In this case, we know that no visibility sort is possible unless one of the cells in the cycle is subdivided.

Testing pairs of cells for mutual occlusion is the most expensive part of this algorithm, since it must be done $O(n^2)$ times, for a data volume of n cells. Therefore a sequence of tests of increasing difficulty are used, in the hope that the early tests in the sequence can eliminate the possibility of occlusion before the more difficult tests are needed.

There is a faster $O(n)$ algorithm that works for a convex grid of convex cells. It is based on pairwise ordering relations between cells that share a common face. If the viewpoint V is on the same side of the face F between cells A and B as cell A is, then cell A occludes cell B, but cell B cannot occlude cell A. In this situation, cell B must come before cell A in the visibility sort for viewpoint V, and we write $B <_{vp} A$. This $<_{vp}$ relation defines a partial order on the cells, and any (total) sorted order consistent with this partial order is a visibility ordering. The reason is that if cell A occludes cell B, there is a viewing ray R from V which intersects cell A and then cell B. Since the grid volume is convex, the ray R does not leave the mesh between cells A and B; instead it passes through a sequence of intervening cells $C_1, C_2, ..., C_k$. Each pair of consecutive cells in the list $A, C_1, C_2, ..., C_k, B$ is separated by a common face, so we have the sequence of relations $B <_{vp} C_k <_{vp} C_{k-1} <_{vp} ... <_{vp} C_1 <_{vp} A$. Thus cell B must come before cell A on the sorted list.

We represent the relations $<_{vp}$ as a directed graph, whose nodes correspond to the cells. There is a directed edge from B to A whenever $B <_{vp} A$. A topological sort of this directed graph will produce a visibility ordering. It works as follows.

For all cells C, set C.incount = 0. For all directed edges corresponding to a relation $B <_{vp} A$, increment A.incount. For each cell A, A.incount now counts the number of cells that A directly occludes across a common face. For all cells C, if C.incount is zero, put C on a queue of cells that can be added to the output list at any time, because they do not occlude any other cells.

While the queue is non-empty, remove a cell C from the queue, and for all directed edges from C to a cell B, decrement B.incount. If any such B.incount reaches zero, put the cell B on the queue. Then add cell C to the next position on the visibility sorted output list.

The algorithm terminates when the queue becomes empty. If this happens before all cells are added to the output list, there is a visibility cycle. The steps in the initialization of the C.incount values treat each cell twice, and each directed edge once. Then in the while loop, each cell and each edge are again treated once. Our cells have only a finite number of allowed topological types, and the maximum number of faces (generating directed edges) per cell is six. Therefore the algorithm takes time $O(n)$.

This algorithm will not necessarily be correct for a non-convex grid, because the viewing ray R may leave the grid volume through an external face, cross a gap of space outside the grid, and then re-enter another cell. Cells with external faces are called boundary cells; suppose there are b of them. If we can add extra relations between these boundary cells, corresponding to the ray segments across the gaps, the topological sort will again produce a visibility ordering.

If we perform pairwise occlusion tests between all $O(b^2)$ pairs of boundary cells, we can find these extra relations in time $O(b^2)$, so the algorithm will cost $O(n + b^2)$. In Comba *et al*. [13] we combine a BSP-tree sort of the external faces and a brute force comparison of each cell having an external face with a small list of p "partially projected" cells. Takes time $O(n + b\ p)$ when BSP trees are balanced, but this is not always the case. There is also significant preprocessing overhead to create the BSP trees.

Recently, we have tried a new approach [14]. We scan convert all the exterior faces into a software A-buffer, which maintains a sorted list of all the viewing-ray / exterior-face intersections per pixel. Then we extract relations for the ray segments across gaps by looking at successive pairs of intersections in the A-buffer. By sorting the exterior faces initially by depth of their centroids, we insure that a new intersection almost always occurs at the head of the A-buffer list for each pixel, so maintaining the list per pixel is inexpensive.

This algorithm usually takes time $O(b \log b + w\ h + a + n)$. The first term is for the sort of exterior faces, the second term is the number of pixels in the final rendered image, and represents the overhead in setting up and checking the A-buffer lists. The third term represents the cost of scan converting the exterior faces, extracting from the A-buffer lists the extra relations across the ray gaps, and processing them during the topological sort. This is proportional to the projected area a of the external faces, measured in pixels. The last term is for processing the relations across shared faces, as in the case of a convex grid.

This algorithm does not produce a true visibility sort, because cells may occlude each other along viewing rays which do not pass through pixel centers. However, since our A-buffer rays are the same as the pixel sample rays used in the final hardware rendering, the image will be correct. This sorting method is faster than any of the others we have tried for general non-convex grids.

4. Non-Planar Faces

In a curvilinear grid, the four vertices of a quadrilateral do not in general lie in the same plane. If a cell has a non-planar quadrilateral face, a viewing ray can leave the

cell and enter again across the same face, so a visibility sort is impossible. In addition, such a face can project onto the image plane with two of its edges crossing, a so-called "bow tie" polygon that also presents problems for rendering. Whether or not a given quadrilateral presents such a problem depends on the viewpoint. Therefore we have developed a viewpoint-dependent method of dividing the problem quadrilaterals into two triangles, and the problem cells into tetrahedra.

Our grids are defined by an ordered list of vertices with 3D locations and scalar field values, and a list of cells of each topological type, with index pointers for their vertices. To subdivide a quadrilateral face into triangles, we draw a diagonal from its vertex of lowest index. (For more general cell topologies, whose faces could have more than four vertices, we might draw more than one diagonal from the lowest index vertex.) This choice of diagonals is consistent for the two cells sharing a face, since it depends only on the indices of the vertices on that face. It is also compatible with subdividing any or all cells into tetrahedra, by connecting with tetrahedra the vertex of lowest index in the whole cell to all triangles from faces not sharing this vertex. This is because the lowest index vertex of the whole cell is also the lowest index vertex of any cell face sharing it.

Once we decide on the diagonals, there are straightforward tests to detect whether a cell with its faces so triangulated can intersect a viewing ray in more than one segment. For example, in figure 3, cell *A* is a problem cell for the visibility sort if a viewing ray can intersect both triangles *EFG* and *EGH*. The presence of this problem depends on the viewpoint *V,* and can be detected using the plane equations of the triangles, and the locations of *V, H*, and *F.* In the figure, cell *A* is a problem cell and would be subdivided into tetrahedra for this viewpoint. However, cell *B* is not a problem cell, and would not be subdivided.

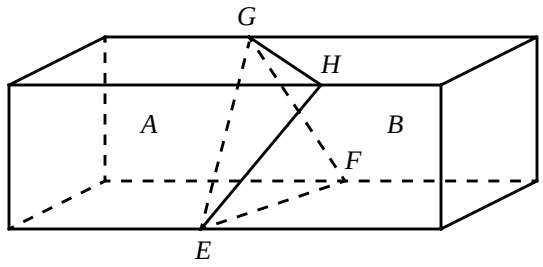

Figure 3. Two cells sharing a non-planar face.

We also subdivide into tetrahedra any cells that contain contour surfaces. Since the scalar field is interpolated linearly across tetrahedra, the contour surfaces intersect a tetrahedron in a collection of parallel polygons, which subdivide the tetrahedron into polyhedral slabs. The contour polygons and volume slabs are sorted from back to front, based on the position of the viewpoint with respect to the contour planes, and rendered in hardware, using the general polyhedron projection method for the slabs.

This method assumes that the transfer functions specifying the particle density and color are linear, but our system also supports piecewise linear transfer functions, with scalar value breakpoints separating the linear pieces. Any cell whose scalar range contains one or more of these breakpoint values is also subdivided into tetrahedra and then into slabs by the breakpoint contours. Then the transfer functions are linear each slab.

5. Plane Slicing

In our second method, described in detail in [15], we take several hundred evenly spaced slicing planes perpendicular to each of the three X, Y, and Z axes, and slice each cell incrementally into polygons on these three planes. We use a variant of the marching cubes algorithm to slice the cells. The polygons for each slice are grouped into an array for efficient OpenGL rendering. The slicing phase is done once in a preprocessing step. The scalar values for the polygon vertices are normalized so they can used as 1D texture coordinates to access the transfer functions which are stored in a texture map.

Keeping the transfer functions in texture allows several optimizations, including an alpha dithering technique [16]. This technique expands the range of allowable opacity values on graphics hardware that uses only 8 bits per channel. This is important because the majority of polygons in typical scientific volume renderings have a very low opacity in order to create sufficient transparency to give a penetrating view of the object, and as the slice density increases, the per-slice opacity must be reduced. Frequently, this means the opacity becomes too small to register. This texture map is adjusted for the appropriate per-slice opacity whenever the slice plane spacing is changed. In the future, we hope to add opacity corrections per pixel to account for the varying length of the viewing rays between slanted slice planes in a perspective view.

At rendering time, the set of slices whose axis is closest to the viewing direction is rendered from back to front with opacity blending using graphics hardware, if available. Images may be generated using progressive refinement: initially a few slices are rendered, then as time permits the resolution of the image is improved by rendering it with more slices. For any viewing direction, the user may request that a new set of slices be generated perpendicular to that direction.

6. Anti-aliasing for Small Cells

Small cells may be missed between slice planes, or there may be no pixel centers inside their slice polygons,

so that they are missed during scan conversion. Keeping such small polygons wastes space, communication bandwidth, vertex transformations, and polygon set-up.

An adaptive mesh is designed to concentrate small cells in regions of complex geometry, high gradients, shocks, or other potentially important regions in the simulation. If these small cells do not contribute to the image, important details in the volume rendering may be absent.

Such missed data is caused by inadequate sampling, and the standard solution is to apply a pre-sampling filter to remove high frequencies and produce anti-aliased output after sampling. The filtered version of a cell or polygon is a complex entity which is difficult to render. Therefore we have used splatting (see Westover [17]) to do the anti-aliasing.

Our first approach was described in Williams *et al*. [8]. Small cells were detected by counting the pixels in a software scan conversion of their projections, and were rendered using a 3 by 3 pixel square piecewise quadratic spine splat. The subpixel location of the center of gravity of the cell was used to analytically compute the splat weights. These were multiplied by the cell volume and the particle density from the transfer function to find the compositing opacity. Compositing was done in software, but the whole process could also be done in hardware, using the subpixel location to determine the vertex texture coordinates for a small textured square, and storing the splat weights in a texture.

In our slicing implementation, we compared the volume of the cell to a threshold, to select small cells that should be splatted instead of sliced. This test will detect cells that may fall between slices, as well as ones whose slice polygons may be missed by pixel samples. Each small cell is assigned to the closest slice plane, and fit by a guassian ellipsoid which approximated its shape. In a curvilinear grid with a slowly varying Jacobian derivative matrix for the mapping from computational to physical coordinates, such splats will sum to near unity at any point in the volume, and thus smoothly interpolate the sampled scalar field.

As explained in Zwicker et al. [5], the “footprint” projection of each splat is convolved with a gaussian presampling filter in the image plane, to give an enlarged 2D splat for anti-aliasing. The elliptical footprint of the enlarged splat is enclosed in a rectangle, which is rendered in texture mapping hardware by multiplying a 2D circular gaussian texture by a polygon RGBA color determined from the transfer function, the cell volume, and the enlarged footprint size.

In a preprocess, the small cells closest to each slice are sorted by the depth of their centers of gravity, and splatted into an image in back to front order. The image is read back, and a rectangle enclosing its non-zero values is determined. Then a sequence of texture maps is created per slicing direction, with each map containing as many of the textured rectangles from consecutive slices as would fit.

During the interactive rendering, the texture maps for the current slicing direction are loaded into texture objects, which are bound one by one as the slices are rendered. When the polygons for the larger cells in a slice are rendered, their presence is recorded in a stencil buffer. Then the rectangle representing the splats is positioned on the slice plane, and composited using texture mapping. A stencil test restricted its effect to the region where polygons are absent. Thus the splats, which are enlarged for anti-aliasing and would otherwise overlap the polygons, do not contribute to regions where the full opacity for the ray segment between slices is already accounted for.

7. Parallelization

We have partially parallelized both the cell projection and the slicing code. For the cell projection, the slow step is computing in software the polygons in the projection. Therefore we assigned several threads on our 64 processor SGI Onyx2 to do the projection, and load the resulting polygonal information into vertex arrays. One thread does the sorting, and one thread makes the OpenGL calls on the full vertex arrays. For details of this implementation, see Bennett *et al*. [18]. For tetrahedral cells with the Shirley - Tuchman projection [3], the speed-up curve leveled out after about 5 projection threads, while for the general grids, where the cell projection code is more time consuming, it leveled out after 18 projection threads. We believe this leveling out is due to communications overhead.

For the slicing method, the Scalable Distributed Volume Rendering (SDVR) System [15] for unstructured data is targeted to large (16-1000+ node) PC clusters some of whose nodes have graphics cards. The system runs under LINUX/UNIX and uses OpenGL and MPI. The primary goal of the system is scalability: as the data set size increases, if additional computational nodes are provided, rendering time remains constant. In addition, the system is designed to run with or without graphics hardware, and on any platform, although not at peak performance.

For distributed rendering, the data set is first partitioned into small load-balanced chunks in a preprocessing step using an out of core algorithm based on a modified k-d decomposition. These brick-shaped chunks are distributed to the nodes of the cluster. Each node then slices, clips, and renders its chunks, and the resulting subimages are gathered and accumulated over the interconnect system. A prototype of the system has been constructed and has run successfully at interactive rates on up to 32 nodes of a PC cluster all of which had Nvidia GeForce-3 graphics cards.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract no. W-7405-Eng-48. We wish to thank Randall Frank for helpful conversations, Mark Duchaineau for the cell slicing code, and our co-authors in the papers below, whose work we have summarized.

References

[1] Nelson Max, “Optical Models for Direct Volume Rendering”, IEEE Transactions on Visualization and Computer Graphics, 1(2) 1995, pp. 99-108.

[2] Brian Cabral, Nancy Cam, and Jim Foran, “Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware”, 1994 Volume Visualization Symposium, ACM Press, pp. 91 - 98.

[3] Peter Shirley and Allan Tuchman, “A Polygonal Approximation to Direct Scalar Volume Rendering”, Computer Graphics 24(5) 1990, pp. 63 - 70.

[4] Roni Yagel, David M. Reed, Asish Law, Po-Wen Shih, and Naeem Shareef, “Hardware Assisted Volume Rendering of Unstructured Grids by Incremental Slicing,” 1996 Volume Visualization Symposium, IEEE Computer Society Press, pp. 55-62.

[5] Matthias Zwicker, Hanspeter Pfister, Jeroen van Barr, and Markus Gross, “EWA Volume Splatting”, Proceedings of Visualization 2001, IEEE Computer Society, pp. 29 - 36.

[6] Nelson Max, Peter Williams, and Claudio Silva, “Cell Projection of Meshes with Non-Planar Faces”, in *Data Visualization: The State of the Art*, Fritz Post, Gregory Nielson, and Georges-Pierre Bonneau, editors, Kluwer Academic Publishers, Boston, 2003, pp. 157 - 168.

[7] Peter Williams and Nelson Max, “A Volume Density Optical Model”, 1992 Workshop on Volume Visualization, ACM Press, pp. 61 - 68.

[8] Peter Williams, Nelson Max, and Clifford Stein, “A high accuracy volume renderer for unstructured data”, IEEE Transactions on Visualization and Computer Graphics, 4(1) 1998 pp. 37 - 54.

[9] Brian Wylie, Kenneth Morland, Le Anne Fisk, and Patricia Crossno, “Tetrahedral Projection using Vertex Shaders”, Volume Visualization and Graphics Symposium 2002, Chris Johnson and Klaus Mueller, editors, ACM Press, pp. 7 - 12.

[10] Greg Schussman and Nelson Max, “Hierarchical Perspective Volume Visualization usingTriangle Fans”, International Workshop on Volume Graphics 2001, Stoney Brook, NY, (Klaus Mueller, Editor), pp. 195 - 200.

[11] M. Newell, R, Newell, and T. Sancha, “Solution to the Hidden Surface Problem”, Proceedings of the ACM National Conference, 1972, pp. 443 - 450.

[12] Clifford Stein, Barry Becker, and Nelson Max, “Sorting and Hardware Assisted Rendering for Volume Visualization”, Proceedings of the 1994 Symposium on Volume Visualization, (Arie Kaufman and Wolfgang Krueger, editors), ACM Press, pp. 83 - 89.

[13] Jao Comba, James Klosowski, Nelson Max, Joseph Mitchell, Claudio Silva, and Peter Williams, "Fast Polyhedral Cell Sorting forInteractive Rendering of Unstructured Grids", Proceedings of Eurographics 1999, pp. C-369 - C-376.

[14] Richard Cook, Nelson Max, Claudio Silva, and Peter Williams, “Image-Space Visibility Ordering for Cell Projection Volume Rendering of Unstructured Data”, Lawrence Livermore National Laboratory Technical Report UCRL-JC-146582-REV-1, submitted to IEEE TVCG, 2003.

[15] Peter Williams, Mark Duchaineau, Randall Frank, and Nelson Max, "A Scalable Distributed Volume Rendering System," Lawrence Livermore National Laboratory Technical Report UCRL-JC-152158-EXT-ABS, 2003.

[16] Peter Williams, Randall Frank, and Eric LaMar, ‘‘Alpha Dithering to Correct Low-Opacity 8 Bit Compositing Errors,’’ Lawrence Livermore National Laboratory Technical Report UCRL-JC-147797, 2003.

[17] Lee Westover, “Interactive Volume Rendering”, Proceedings of the Chapel Hill Workshop on Volume Visualization, May 1989, pp. 9 - 16.

[18] Janine Bennett, Richard Cook, Nelson Max, Deborah May, and Peter Williams, “Parallelizing a High Accuracy Hardware_assisted Volume Renderer for Meshes with Arbitrary Polyhedra”, 2001 Symposium on Parallel and Large-Data Visualization and Graphics, ACM, pp. 101 - 106.