Hierarchical Perspective Volume Rendering Using Triangle Fans

Greg Schussman and Nelson Max

Center for Image Processing and Integrated Computing (CIPIC)
Department of Computer Science, University of California
Davis, CA 95616-8562, USA
{schussma, max}@cs.ucdavis.edu

Abstract. We present a method of accelerated perspective volume rendering using cell projection, triangle fans, and a data hierarchy. The hierarchy allows mixed resolution rendering, greatly increasing speed. We utilize triangle fans for additional speed and texture mapped opacity for accuracy.

1 Introduction

Traditionally, volume rendering of regular cubical grids has used 3D texture mapping and compositing [1, 2] or ray tracing [4, 8] approaches, which take advantage of the regular distribution of the data. Polyhedron projection approaches [6, 9, 11, 12] have been used mainly for irregular meshes. These methods divide the projection of a cell by the projections of its edges into polygons which can be scan converted and composited using inexpensive and efficient polygon rendering hardware. They are particularly efficient at rendering large cells, where the polygon creation and set up costs are amortized over many pixels. Laur and Hanrahan [3] developed hierarchical Oct-tree approximations to cubical gridded data, which use large cells where appropriate. Their hexagon splatting approach was only an approximation, but can be made more precise using the techniques of [6, 10–12].

The polyhedron projection approach requires a global back-to-front visibility sort, which in general is difficult and time consuming [7, 10]. However, for regular cubical or oct-tree grids, the sort is trivial [3, 5].

In this paper, we optimize the polyhedron projection with the use of triangle fans. Wittenbrink [13] has applied triangle fans to the Shirley-Tuchman tetrahedron projection scheme [9], and gotten very good performance. For orthogonal projection of a cubical grid, Wilhelms and Van Gelder [11] have used a line sweep algorithm to construct the polygons. This is slower than the case-based approach of Shirley and Tuchman, but in the orthogonal case, all cube projections are congruent, so the polygons need to be constructed only once per frame. For perspective projection of arbitrary polyhedra, Max [7, 10, 12] has used incremental edge addition to construct the polygons, but this general construction is also slow. Here we use a case based approach to efficiently construct the polygons in the 10 topologically different perspective projections of a cube, and construct optimal triangle fans for each.

2 Overview

We render using alpha-blending in back-to-front order. Each cube is classified using simple dot products with its face normals. Then a lookup table provides a mapping of standard-case, hard-coded, optimal triangle fans onto the current cube.

A speedup to volume rendering is to build a multi-resolution hierarchy of the data, and render at a mixed resolution where only a specified amount of error is acceptable. By merging eight smaller cubes into a larger one, the number of triangles and CPU calculations needed to render the corresponding region of the volume is reduced by a factor of eight and the number of OpenGL fragments is halved. Linear interpolation across triangles of computed opacity vertex values is a reasonable approximation when the cubes are small [9]. However, this approximation becomes poor when the cube size grows. We utilize texture mapping to effectively compute the exponential per pixel needed for obtaining the correct opacity, while still reaping the benefits of hardware accelerated rendering [7, 10, 12].

This paper is organized as follows. Section 3 discusses cell classification. Our optimal standard-case triangle fans are presented in Section 4. The remapping of triangle fan vertices from standard cases is covered in Section 5. In Section 6 we discuss the mixed resolution data hierarchy. A description of our texture map appears in Section 7. Finally, Section 8 presents images and timing results.

3 Cell Classification

Under perspective projection, there are 10 topologically distinct cases. These are shown in Figure 1, which is arranged so that each case appears as a small rotation of the cases near it; rolling a row downward produces the next row down, and rolling a column to the left produces the next column to the left. There is no case directly below case 10 because rotating case 10 until the bottom face is not visible would produce a view that is topologically equivalent to case 11. This is also why there are no other cases shown in the lower left region of the figure.

The labels in the figure are based on the results of dot products described later in this section. The first digit is the number of faces with positive dot products and the second digit is the number of faces with dot products exactly equal to zero. In some cases an additional letter is used for further classification.

The following describes the computation of information used for selecting and indexing into lookup tables. For each face of a cube, the dot product of the face normal with the viewing vector is computed and saved as a pair, along with the associated face number. We denote this pair (d, f). The face normal is a unit vector oriented out of the cube. The viewing vector has unit length and points

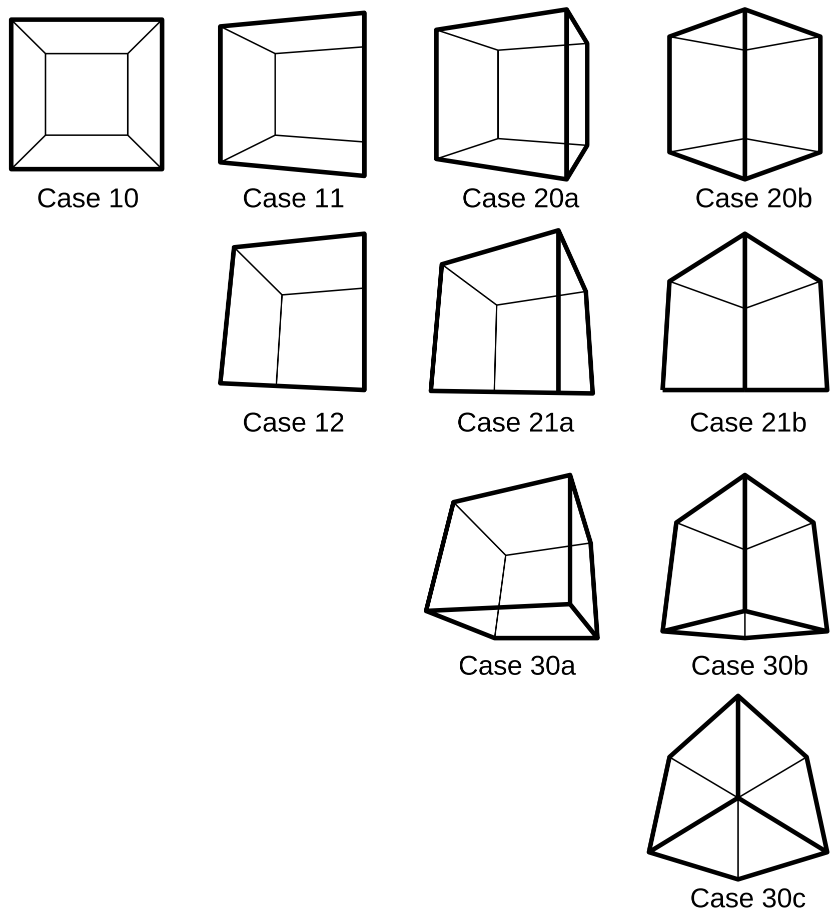


Fig. 1. These are the 10 topologically distinct perspective projections of a cube. The cases can be grouped by class, which is given by the first digit of the case label.

toward the eye from the face center. Our face numbering convention is shown in Figure 2a.

Conceptually, the six (d, f) pairs are sorted in descending order. That is, $d_0 \geq d_1 \geq \ldots \geq d_5$. In practice, we are only interested in the first three pairs, so it is not necessary to store and sort all six pairs. We denote the corresponding faces by using the same subscripts. For example, f_0 is the face corresponding to the most positive dot product.

Figure 3 shows how the values for d_0, d_1, and d_2 relate to the 10 different cases of Figure 1. The table in Figure 3 is presented to facilitate discussion. Classification is not done by evaluating the entries in the table. Instead, the d values are used to select a lookup table, and the f values are used to index into that table.

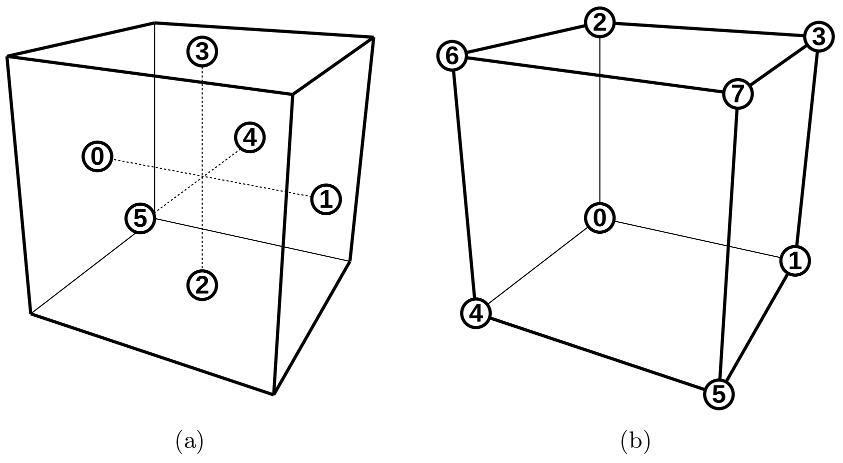


Fig. 2. The numbering conventions are shown for (a) faces and (b) vertices. Both cubes have the same orientation.

Dot Products	Case	Class
$d_0 > 0 \quad > d_1 \geq d_2$	10	1
$d_0 > 0 \quad = d_1 > d_2$	11	1
$d_0 > 0 \quad = d_1 = d_2$	12	1
$d_0 > d_1 > 0 \quad > d_2$	20a	2
$d_0 = d_1 > 0 \quad > d_2$	20b	2
$d_0 > d_1 > 0 \quad = d_2$	21a	2
$d_0 = d_1 > 0 \quad = d_2$	21b	2
$d_0 > d_1 \geq d_2 > 0$	30a	3
$d_0 = d_1 > d_2 > 0$	30b	3
$d_0 = d_1 = d_2 > 0$	30c	3

Fig. 3. The 10 distinct cases can be identified by the dot product results.

When two or more of $\{d_0, d_1, d_2\}$ are exactly equal to one another, or exactly equal to zero, we call the associated case degenerate. Graphically, this corresponds to a vertex projecting exactly onto an edge or onto another vertex. Of all possible projected orientations, the degenerate ones are quite rare. The reason for calling these cases degenerate is that triangle fans for non-degenerate cases can still be used even though a few of the triangles will project to have zero area. For example, cases 11 and 12 are degenerate versions of case 10.

There is no need to compute optimal triangle fans for degenerate cases. The degenerate cases do not cause slowdown in practice; because they occur so rarely, optimizing them produces no noticeable speedup.

We combine cases 10, 11, and 12 to be class 1, cases 20a, 20b, 21a, 21b to be class 2, and cases 30a, 30b, and 30c to be class 3. The class is given by the number of $d > 0$. Optimal triangle fans are constructed for each class.

4 Optimal Triangle Fans

Optimal triangle fans are constructed as templates for default orientations of the non-degenerate projection cases. The information computed in Section 3 is used to consult lookup tables for mapping the indices of the actual projected cube vertices to the standard template. In this way, the templated triangle fans conform to the projected cube.

Our notion of "optimal" should be clarified. We assume implementation in OpenGL, which does not support the swaptmesh() call of IrisGL. A commonly suggested workaround is to repeat a vertex. However this forces two zero-area triangles and also requires re-issuing the vertex of the swapped edge. So simply repeating a vertex in OpenGL is not equivalent to issuing the swaptmesh() command. We compared performance of the vertex repeating method versus simply starting a new triangle fan, and found no significant difference. For clarity, we describe our solution in terms of pairs of triangle fans. Because none of the non-degenerate cases can be covered with a single triangle fan or single triangle strip (without resorting to repeating vertices for edge flipping), our two-fan solution is optimal.

The triangle fans for class 1 are shown in Figure 4, and those for class 2 are shown in Figure 5. We describe a triangle fan by the indices of the vertex points. Our vertex numbering convention is shown in Figure 2b. Class 1 is handled by two triangle fans: $fan(2, 3, 7, 6, 4, 0, 3)$, and $fan(1, 3, 0, 4, 5, 7, 3)$. In this notation, the vertices correspond to the order expected by OpenGL; the first vertex is the center of the triangle fan, and then the remaining vertices lie on the perimeter of the fan. Class 2 is handled by a triangle fan and a triangle strip: $fan(1, 7, 3, B, A, 4, 5, 7)$, and $strip(7, 3, 6, B, 2, A, 0, 4)$. The vertices A and B are intersections of projected edges.

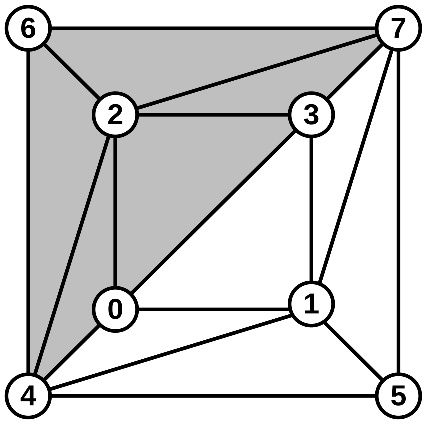


Fig. 4. Class 1 has two triangle fans: $fan(2, 3, 7, 6, 4, 0, 3)$ shown in grey and $fan(1, 3, 0, 4, 5, 7, 3)$ shown in white.

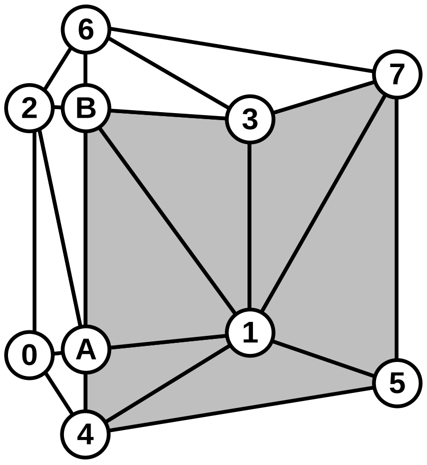


Fig. 5. Class 2 consists of $fan(1, 7, 3, B, A, 4, 5, 7)$ shown in grey, and $strip(7, 3, 6, B, 2, A, 0, 4)$ shown in white. A and B are not original cube vertices; instead, they are edge intersections which must be computed.

The vertices that lie on the outer edge of the projected cube all have thickness zero. For the other vertices or intersections of projected edges, the thickness must be computed. Thickness and density are used as texture coordinates, as discussed in Section 7.

Class 3 can appear in two ways, each a mirror image of the other. Both versions 3a and 3b require their own pair of triangle fans, which are illustrated in Figure 6. The class 3a solution consists of $fan(A, 6, 2, 0, 4, 5, 1, B, 6)$ and $fan(B, 1, 5, 7, 3, 2, 6)$. Class 3b uses $fan(A, 6, B, 1, 0, 4, 5, 7, 6)$ and $fan(B, 6, 7, 3, 2, 0, 1)$. In both versions, vertex 6 is nearest to the eye. However, point A is at the intersection of E_{46} and E_{01} in class 3a, and at the intersection of E_{46} and E_{15} in class 3b. Here E_{ij} means the edge corresponding to the line segment joining vertex i and vertex j. The edges whose intersection yields point B are also different for class 3a and class 3b. Classes 1 and 2 do not have this problem because for them a reflection is topologically equivalent to a rotation in the projected plane.

5 Mapping from Standard Orientation

In Section 4, optimal triangle fans were computed for standard cube orientations. In this section, we describe how to remap the standard vertices onto the specific cube being rendered. This mapping makes use of the d and f values computed in Section 3. Counting the number of strictly positive d provides the cube's class. For each of the three classes, lookup tables are used.

For class 1, f_0 is used to index into the lookup table shown in Figure 7. For example, say face 3 is oriented directly toward the eye, so f_0 would have been 3.

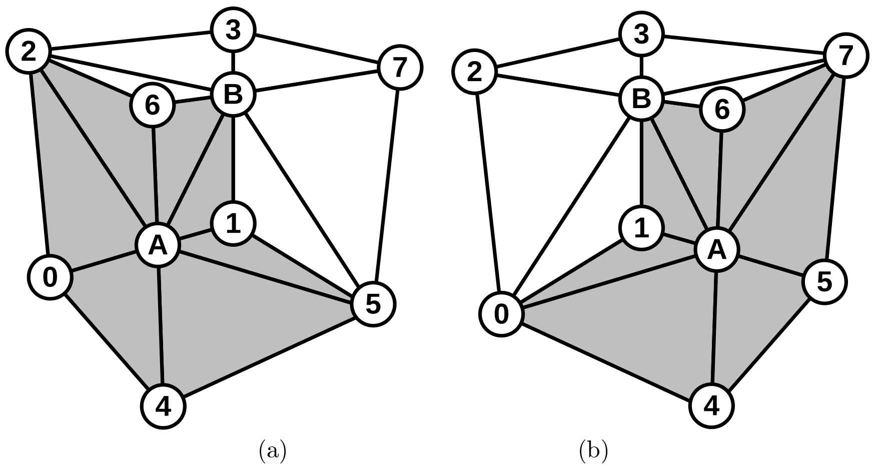


Fig. 6. Class 3a (a) and class 3b (b) are mirror images of each other and require separate triangulations. Class 3a consists of $fan(A, 6, 2, 0, 4, 5, 1, B, 6)$ shown in grey, and $fan(B, 1, 5, 7, 3, 2, 6)$ shown in white. Class 3b consists of $fan(A, 6, B, 1, 0, 4, 5, 7, 6)$ and $fan(B, 6, 7, 3, 2, 0, 1)$.

Then, according to row $f_0 = 3$ of the table, the vertices of the first triangle fan of class 1 get remapped from the standard $fan(2, 3, 7, 6, 4, 0, 3)$ to the specific $fan(0, 1, 3, 2, 6, 4, 1)$.

The table shown in Figure 8 is used for Class 2. Because Class 2 has two faces oriented toward the eye, both f_0 and f_1 are used to index into this table. Vertex remapping proceeds as in the previous example.

It should be noted that in an actual implementation, there would be unused rows in the table. These unused rows are not shown in the figure. There are two reasons for unused rows. First, no face is ever repeated in the f values, so the $f_0 = 0$, $f_1 = 0$ row is not shown in the table. Second, faces on opposite sides of a cube can never be both oriented toward the eye at the same time, which is why the $f_0 = 0$, $f_1 = 1$ row is not shown in the table.

Figure 8 can also be used for class 3, which has 3 faces oriented toward the eye. A separate table could handle both class 3a and 3b together, but this table

f_0	V_0	V_1	V_2	V_3	V_4	V_5	V_6	V_7
0	1	5	3	7	0	4	2	6
1	4	0	6	2	5	1	7	3
2	2	3	6	7	0	1	4	5
3	4	5	0	1	6	7	2	3
4	6	7	4	5	2	3	0	1
5	0	1	2	3	4	5	6	7

Fig. 7. Case 10 vertex remapping lookup table

would be $6 \times 6 \times 6 \times 8$, which is 1728 entries, most of which would never be used. Instead, we use a smaller table that is $6 \times 6 \times 6$ (216 entries) to look up whether the current cube is class 3a or class 3b. For class 3a the consecutive digits of f_0, f_1, f_2 will be one of {024, 052, 043, 035, 142, 153, 134, 125, 240, 214, 205, 251, 350, 315, 341, 304, 413, 430, 421, 402, 503, 512, 520, 531}, and all other possible combinations of f_0, f_1, f_2 will belong to class 3b.

We can re-use the table for class 2 as follows. If this is class 3a, index into the table using f_0, f_1, and use that row to remap the vertices of the triangle fans from Figure 6a. Otherwise swap the order of the indices, indexing into the table with f_1, f_0, and use that row to remap the vertices of the triangle fans from Figure 6b.

f_0	f_1	V_0	V_1	V_2	V_3	V_4	V_5	V_6	V_7
0	2	5	7	1	3	4	6	0	2
0	3	3	1	7	5	2	0	6	4
0	4	1	5	3	7	0	4	2	6
0	5	7	3	5	1	6	2	4	0
1	2	0	2	4	6	1	2	5	7
1	3	6	4	2	0	7	5	3	1
1	4	2	6	0	4	3	7	1	5
1	5	4	0	6	2	5	1	7	3
2	0	2	3	6	7	0	1	4	5
2	1	7	6	3	2	5	4	1	0
2	4	3	7	2	6	1	5	0	4
2	5	6	2	7	3	4	0	5	1
3	0	4	5	0	1	6	7	2	3
3	1	1	0	5	4	3	2	7	6
3	4	0	4	1	5	2	6	3	7
3	5	5	1	4	0	7	3	6	2
4	0	6	7	4	5	2	3	0	1
4	1	5	4	7	6	1	0	3	2
4	2	4	6	5	7	0	2	1	3
4	3	7	5	6	4	3	1	2	0
5	0	0	1	2	3	4	5	6	7
5	1	3	2	1	0	7	6	5	4
5	2	1	3	0	2	5	7	4	6
5	3	2	0	3	1	6	4	7	5

Fig. 8. Vertex remapping lookup table for classes 2 and 3a. For class 3b, f_0 and f_1 are swapped.

6 Mixed Resolution Data Hierarchy

Our data hierarchy is like that of an oct-tree, which is constructed from the bottom up. We call the nodes of the tree cubes. Each cube has the following information: i, j, k cell indices, a level number (0 is the finest level) a scalar value, an error, and pointers for 8 children. Minimum and maximum extent in x, y, and z can be calculated from the cell indices and the level. The order of these pointers is the same as the ordering of vertices in Figure 2b.

Given an initial dataset with $l \times w \times h$ voxels, a cube is created for each voxel. The initial high resolution cubes are the leaves of the tree. Because scalar value of the voxel is represented exactly in these leaves, the error is exactly zero.

The cubes for the next level in the tree are created by merging (up to) 8 of the initial cubes. So, in the 8 cube case, the parent cube $C_{i,j,k}$ in a coarser level is the result of merging $C_{2i,2j,2k}$, $C_{2i+1,2j,2k}$, $C_{2i,2j+1,2k}$, $C_{2i+1,2j+1,2k}$, $C_{2i,2j,2k+1}$, $C_{2i+1,2j,2k+1}$, $C_{2i,2j+1,2k+1}$, and $C_{2i+1,2j+1,2k+1}$ from the next finer level. The parent cube's value is the average of the values of the children. The parent's error is the maximum of the absolute values of all differences between the parent's value and the value of all leaves descended from the parent.

Once all cubes from one level of the tree have been merged to produce a smaller, coarser level, the process is repeated recursively, using the coarse level as the fine level input for the next iteration.

Selecting cells for rendering is a recursive procedure based on an error tolerance. The tree is traversed from the root in depth-first order, until the error of a cell is acceptable, at which point it is selected for rendering. Rendering uses a recursive back-to-front traversal of the oct-tree.

7 Texture Map

Section 3 discussed the correct order of vertices for triangle fans. In Figures 5 and 6, the vertices A and B were simply called the intersection of the projected edges of the cube. In three space, the line segments corresponding to cube edges are skew; that is, they do not actually intersect. We are interested in two points, one at the front of the cube and one at the back. These points are where the viewing ray intersects each of the line segments. The same figures that provide the standard triangulation for the fans also show which edges must be used for computing the front and back intersection points. Although front and back points will project to the same place in the viewed image, the 3D distance between them is the length l of the viewing ray segment. This length l must also be computed for viewing rays that enter the cube through a vertex (like vertex 6 in Figure 6) and then exit through a face, or vice-versa (like vertex 0 in Figure 4). Suppose the average of extinction coefficients at the ray segment's endpoints is a. Then the correct alpha to use in compositing is $\alpha = 1 - e^{-al}$, as discussed in [7, 10–12].

If we were to calculate colors and opacities for each vertex of the triangle fans, linear interpolation across the face of the triangles would be an approximation to evaluating the exponential opacity function for each pixel. The approximation is only good when for cells that are relatively small and transparent. We do not assume all cells are relatively transparent. Also, when rendering at mixed resolution, relatively large cells could be selected for rendering. We use the texture map of [7, 10, 12] to achieve the correct opacities.

The texture map at texture coordinates (u, v) is loaded with $1 - e^{-uv}$. Then the ray length l and average extinction coefficient a are used as the (u, v) texture coordinates for each vertex in a triangle, triangle fan, or triangle strip. Since each triangle lies in the projection of a single front and back face of the cube, it is a very good approximation to let the graphics hardware linearly interpolate these texture coordinates across the triangle, and then look up the effect of the exponential in the texture table to get the alpha for compositing. (If the projection were orthogonal instead of perspective, this approximation would be exactly correct.)

8 Results

We applied our method to a $256 \times 256 \times 110$ CT dataset of an engine block. Computation and rendering was performed on a PC running Linux on a 1.4 GHz Pentium 4 and using an nVIDIA GeForce 2 graphics processor.

Figure 9 summarizes the parameters and timing for the images in Color Plate 1. The images correspond to no allowed error, some allowed error, and a lot of allowed error, respectively. Selection is the time it takes to traverse the hierarchy in view-dependent back-to-front order, selecting cells which satisfy the error tolerance. The selection time is not included in the rendering time. Cell classification was used to skip cells which were completely transparent, so the number of cubes reflects only those which contribute to the image. It took 4 seconds to construct the hierarchy.

Color Plate 1b does not look significantly different from Color Plate 1a, yet it renders 39 times faster. Color Plate 1c is noticeably poorer in quality, although it renders at about 14 frames per second. It still shows many of the essential features of the data, allowing meaningful interactive data navigation where detail can be added progressively when navigation stops.

Error	Cubes	Triangles	Selection	Rendering
0%	5.20 M	62 M	0.85 sec	37.5 sec
18%	0.19 M	2.3 M	0.036 sec	0.95 sec
48%	15.8 K	0.19 M	0.0019 sec	0.070 sec

Fig. 9. Rendering statistics for the engine block dataset

9 Conclusion

We have demonstrated a fast method of volume rendering in perspective. The method utilizes a hierarchical data structure for mixed resolution rendering, which, along with triangle fans, minimizes traffic to the graphics hardware subsystem. Cell projections come from a few sets of standard triangle fan cases that are easily identified by computing simple dot products and using fast table lookups.

10 Acknowledgments

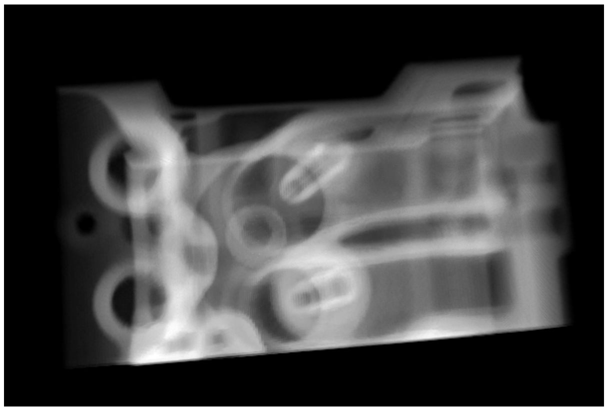
This work was supported by the National Science Foundation through the Large Scientific and Software Data Set Visualization (LSSDSV) program under contract ACI 9982251, and through the National Partnership for Advanced Computational Infrastructure (NPACI); the Office of Naval Research under contract N00014-97-1-0222; and the Lawrence Livermore National Laboratory under ASCI ASAP Level-2 Memorandum Agreement B347878 and under Memorandum Agreement B503159. We also acknowledge the support of the Stanford Linear Accelerator Center(SLAC), General Atomics, the Digital Media Innovation Program (DiMI), ST Microelectronics, ALSTOM Schilling Robotics and SGI. We thank the members of the Visualization and Graphics Research Group at the Center for Image Processing and Integrated Computing (CIPIC) at the University of California, Davis.

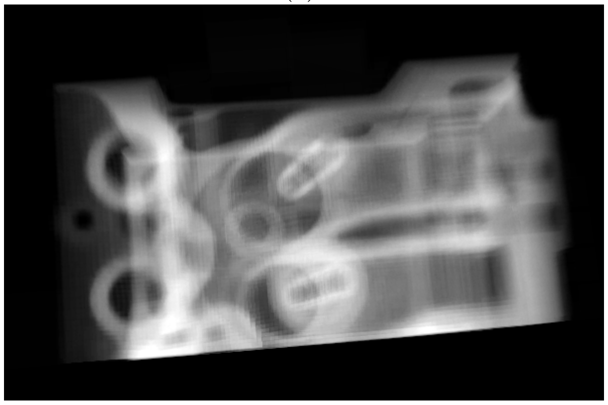
We would also like to thank Stanford Computer Graphics Laboratory for making the engine block dataset available.

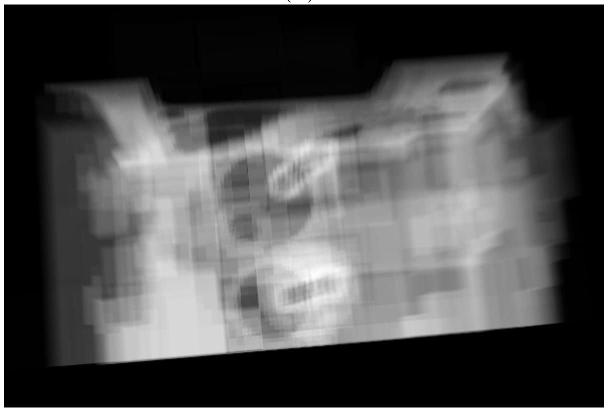
References

1. Brian Cabral, Nancy Cam, and Jim Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. *1994 Symposium on Volume Visualization*, pages 91–98, October 1994. ISBN 0-89791-741-3.
2. Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. *Computer Graphics* (*Proceedings of SIGGRAPH 88*), 22(4):65–74, August 1988. Held in Atlanta, Georgia.
3. David Laur and Pat Hanrahan. Hierarchical splatting: A progressive refinement algorithm for volume rendering. *Computer Graphics* (*Proceedings of SIGGRAPH 91*), 25(4):285–288, July 1991. ISBN 0-201-56291-X. Held in Las Vegas, Nevada.
4. Marc Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, May 1988.
5. Nelson Max. *Sorting for Polyhedron Compositing in "Focus on Scientific Visualization"*, pages 259–268. H. Hagen, H. Müller, and G. Nielson, editors, Springer-Verlag, Berlin, Germany, 1993.
6. Nelson Max, Pat Hanrahan, and Roger Crawfis. Area and volume coherence for efficient visualization of 3D scalar functions. *Computer Graphics* (*San Diego Workshop on Volume Visualization*), 24(5):27–33, November 1990.

7. Nelson Max, Peter Williams, and Claudio Silva. Approximate volume rendering for curvilinear and unstructured grids by hardware-assisted polyhedron projection. *International Journal of Imaging Systems and Technology*, 11:53–61, 2000.
8. Hanspeter Pfister, Jan Hardenbergh, Jim Knittel, Hugh Lauer, and Larry Seiler. The volumepro real-time ray-casting system. *Computer Graphics* (*Proceedings of SIGGRAPH 99*), pages 251–260, August 1999. ISBN 0-20148-560-5. Held in Los Angeles, California.
9. Peter Shirley and Allan Tuchman. A polygonal approximation to direct scalar volume rendering. *Computer Graphics* (*San Diego Workshop on Volume Visualization*), 24(5):63–70, November 1990.
10. Clifford Stein, Barry Becker, and Nelson Max. Sorting and hardware assisted rendering for volume visualization. *1994 Symposium on Volume Visualization*, pages 83–90, October 1994. ISBN 0-89791-741-3.
11. Jane Wilhelms and Allen Van Gelder. A coherent projection approach for direct volume rendering. *Computer Graphics* (*Proceedings of SIGGRAPH 91*), 25(4):275–284, July 1991. ISBN 0-201-56291-X. Held in Las Vegas, Nevada.
12. Peter L. Williams, Nelson L. Max, and Clifford M. Stein. A high accuracy volume renderer for unstructured data. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):37–54, January-March 1998. ISSN 1077-2626.
13. Craig Wittenbrink. Cellfast: Interactive unstructured volume rendering. *Proceedings IEEE Visualization 99, Late Breaking Hot Topics*, pages 21–24, October 1999.


(a)


(b)


(c)

Color Plate 1: The engine block dataset at full resolution (a) takes 38.3 seconds to render. With an 18% error tolerance (b), it takes 0.986 seconds to render, and with a 43% error tolerance (c), it renders in 0.072 seconds, (or about 14 frames per second).