# A Polygonal Approximation to Direct Scalar Volume Rendering

Peter Shirley *          Allan Tuchman†

Center for Supercomputing Research and Development
305 Talbot Lab
University of Illinois
Urbana, Illinois 61801

## Abstract

One method of directly rendering a three-dimensional volume of scalar data is to project each cell in a volume onto the screen. Rasterizing a volume cell is more complex than rasterizing a polygon. A method is presented that approximates tetrahedral volume cells with hardware renderable transparent triangles. This method produces results which are visually similar to more exact methods for scalar volume rendering, but is faster and has smaller memory requirements. The method is best suited for display of smoothly-changing data.

**CR Categories and Subject Descriptors:** I.3.0 [Computer Graphics]: General; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling.

**Additional Key Words and Phrases:** Volume rendering, scientific visualization.

## 1 Introduction

Display of three-dimensional scalar volumes has recently become an active area of research. A scalar volume is described by some function $f(x, y, z)$ defined over some region $R$ of three-dimensional space. In many scientific applications, $R$ will be some fairly simple region such as a cube or deformed cube, and $f$ will be defined at a finite set of points within $R$, with an interpolation function filling in the gaps between points. In many applications, such as those that employ finite element techniques, $R$ will be more complex, e.g. the interior of a mechanical part.

*Current Address: Department of Computer Science, Lindley Hall, Indiana University, Bloomington, IN 47405. Email: shirley@cs.indiana.edu.

†Email: tuchman@csrd.uiuc.edu.



The most intuitive strategy for displaying $f$ is to choose some particular value $k$ and display all points where $f(x, y, z) = k$. For continuous $f$ this will yield a set of well defined *iso-value surfaces* or *isosurfaces*[LC87]. Another method, the method of interest in this paper, is to display $f$ as a three-dimensional cloud. This idea of displaying volumes as clouds is commonly called *direct volume rendering*[Sab88, UK88].

To generate directly rendered images of $f$, two basic methods have been used: ray tracing[Bli82, KH84, Lev88, Sab88, UK88, SN89] and direct projection[FGR85, LGLD86, UK88, DCH88, Wes90]. Upson and Keeler discuss the relative merits of ray tracing and direct cell projection in their V-buffer paper[UK88]. In ray tracing, viewing rays are sent through each pixel and integrated through the volume. In direct projection, each cell of the volume is projected onto the screen. Because each cell is partially transparent, a painter's depth ordering algorithm is used for direct projection. Unfortunately, current graphics workstations do not support scan conversion of volumetric primatives.

Kaufman describes a hardware design that scan converts volume primatives into a three dimensional grid, and then performs ray tracing to produce an image[Kau87]. Kaufman's design has the advantage of implicitly correct depth ordering, so that unstructured grids may be rendered, and the further advantage that curvilinear cells are approximated by tricubic parametric volumes rather than polyhedrons, but such a system is not currently commercially available.

Hibbard and Santek used parallel stacks of transparent polygonal sheets to approximate volume cells, but their method is a 'quick and dirty' way to get pictures, and they reported noticeable errors for off normal viewpoints[HS89].

In this paper, we present the Projected Tetrahedra (PT) algorithm, a method of approximating directly projected volume cells with sets of partially transparent polygons that can then be rendered relatively quickly on a graphics workstation. These polygonal sets are recalculated for each new viewpoint, but are a more accurate approximation to direct projection volume than Hibbard and Santek's view independent technique.

## 2 Algorithm

The Projected Tetrahedra algorithm operates with any set of three-dimensional data that has been *tetrahedralated*, the three-dimensional analogue of triangulated data in the plane. Since a large class of data is sampled or computed on a lattice of six-sided cells or cubes, we include this decomposition in our description. The tetrahedra are ultimately described as partially transparent triangular elements for hardware rendering.

We chose to begin with tetrahedra both to demonstrate the high-quality images that can be produced as well as to accommodate volumes more general than rectilinear grids.

The algorithm proceeds as follows:

1. Decompose the volume into tetrahedral cells with values of $f$ stored at each of the four vertices. Inside each tetrahedron, $f$ is assumed to be a linear combination of the vertex values (Section 2.1).
2. Classify each tetrahedron according to its projected profile relative to a viewpoint (Section 2.2).
3. Find the positions of the tetrahedra vertices after the perspective transformation has been applied (Section 2.3).
4. Decompose projections of the tetrahedra into triangles (Section 2.4).
5. Find color and opacity values for the triangle vertices using ray integration in the original world coordinates (Section 2.5).
6. Scan convert the triangles on a graphics workstation (Section 2.6).

The idea of Projected Tetrahedra is that the image composed of the triangles drawn in the last step will be similar in appearance to a full direct volume rendering of the input tetrahedra. Because the triangles are semi-transparent, they must either be rendered in depth order, or an enhanced frame buffer such as the A-buffer[Car84] must be used. Using an A-buffer may not be feasible for large volumes where each pixel might have hundreds of overlapping transparent polygons. Depth ordering for rectilinear grids is discussed by Frieder et al.[FGR85] and by Upson and Keeler[UK88], and for non-rectilinear grids is discussed by Williams and Shirley[WS90] and Max et al.[MHC90]. Unfortunately, a non-rectilinear mesh, even if its boundary is convex, may have cycles that make a correct depth ordering impossible[WS90]. The frequency of such cycles in computational meshes is unknown.

Throughout our discussion it is assumed that a perspective projection is used. An orthographic projection can be substituted by modifying step 2. Instead of using the viewpoint for classification, each face must be classified from a point on the view plane. This point can be the orthographic projection of any vertex on that face.

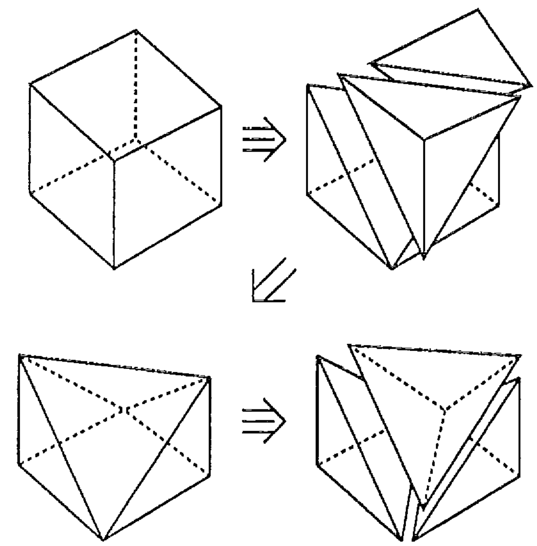

Figure 1: Decomposition of a cube into five tetrahedra

### 2.1 Decomposition into Tetrahedra

If the volume is rectilinear or curvilinear, then each rectilinear or curvilinear cell must be partitioned into tetrahedral elements with an original data point at each vertex. Figure 1 shows the decomposition of a cube into 5 tetrahedra, the smallest number of tetrahedra possible. This decomposition applies to any curvilinear cell (cube deformations without self intersections). There are only two rotational states of this decomposition. If two adjacent curvilinear cells are to be subdivided in this way, care must be taken to avoid the *cracking problem*, so that every point will be in *exactly* one tetrahedron. This problem is similar to the two-dimensional cracking problem encountered when spline surfaces are polygonalized. Problems can occur if the four vertices on a face of a six sided cell are not coplanar in a curvilinear mesh. In Figure 2, two curvilinear cells share a boundary surface which is not planer. If this surface is approximated by two triangles, then there are two possible options for which triangles are used, as shown on the bottom of the figure. The same pair of triangles must be used by each of the cells or cracking (an overlap or gap between cells) will occur. This implies that adjacent cells must use opposite rotational states of the decomposition shown in Figure 1. This will produce a three-dimensional checkerboard pattern of decomposition, with alternating rotational states, and thus no cracking can occur.

### 2.2 Classification

A tetrahedron may have any of four silhouettes depending on the viewpoint and the orientation of the tetrahedron. Since the goal is to approximate this volume element with triangles, we first classify the tetrahedron based on its projected shape. Figure 3 enumerates the four possible projected

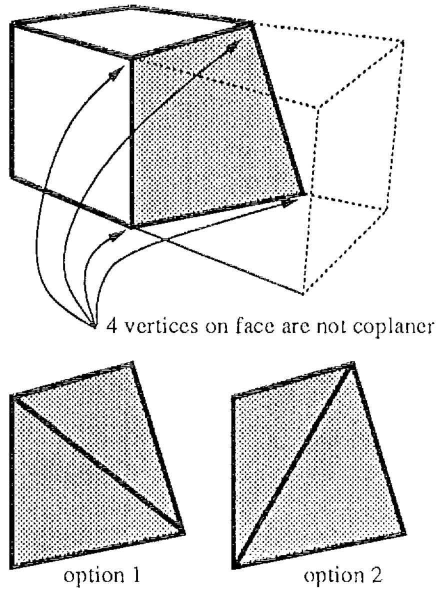


Figure 2: Two adjacent curvilinear cells sharing a non-planer boundary should have the same polygonalization of the boundary to avoid the three-dimensional cracking problem.

shapes arising from six possible cases. We note that each case can be distinguished by examining the surface normal vectors of each face and comparing them with the viewer's eye position or viewpoint. We only care whether the surface normal points toward, points away from, or is perpendicular to the view vector, so we use the notation '+', '-', or '0' to mark each face. Classes 1a and 1b have the same shape, but in one case (1a) the eye looks directly at three faces and away from one, so is marked '+++-', whereas in the other (1b), three faces are not visible to the eye and it is marked '---+'. The number of '+ ', '-' and '0' faces are counted and these values used as a table index to classify each tetrahedron into one of the 6 classes shown in Figure 3.

Clearly, classes 3 and 4 are degeneracies class 1 or class 2. We treat them as separate since they are easy to identify during this classification step and less efficient to test for later. By doing so we are also able to avoid generating degenerate polygons.

In some cases, the surface normal may not be immediately available or its direction may be ambiguous (since each face will have two opposing normal vectors). Therefore we use the plane equation $F$ for each face which is directly available from the 3 vertices that make up the plane. If tetrahedron T is defined by vertices $P_1$, $P_2$, $P_3$, and $P_4$, then find the equation of plane $P_1P_2P_3$. If $F(eye)$ is zero, then the eye is collinear with the plane and allows that face to be marked '0'. If non-zero, the value of $F(P_4)$ is computed. If this value has the same sign as the $F(eye)$, then the plane points away, otherwise it points toward the view point.

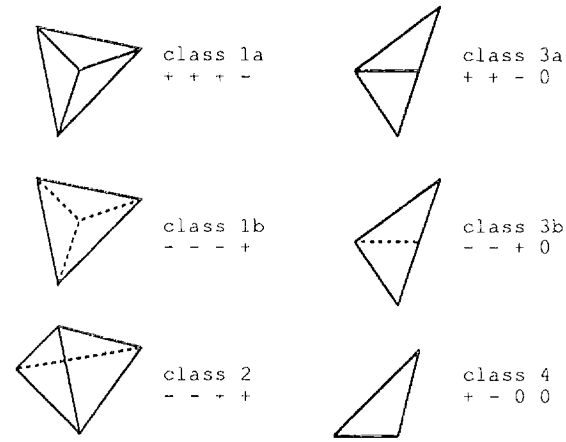


Figure 3: Classification of Tetrahedra Projections

### 2.3 Projection

The tetrahedra must be decomposed into triangles, in essence triangulating the projection of the tetrahedron. To do this we define the transformation from the 3-dimensional viewing frustrum to a 3-dimensional rectangular parallelepiped as a mapping from world coordinates to perspective coordinates. For an orthographic projection this transformation is the identity. It preserves the relative distance of points along the axis of the viewing coordinate system which is aligned with the view direction vector.

It is easier to transform each tetrahedron to the perspective (or orthographic) viewing coordinate system and then intersect 2-dimensional lines (formed by discarding the Z coordinate) than to do similar calculations in the original world coordinate system. Also, this transformation must be performed anyway.

The viewing transformation is a simple one composed of a translation to the origin, a rotation from the world coordinate system to the viewing coordinates, and an optional perspective transformation. The viewing transformation is applied to each vertex of the tetrahedron.

In this step and the decomposition step described in the next section, both the viewing matrix and its inverse are needed. The matrices that perform these transformations are described in the Appendix of this paper.

### 2.4 Decomposition into Triangles

Each projected tetrahedron is decomposed into one to four triangles. The projection may be used to find the coordinates of each triangle, as shown in Figure 4.

For each triangle, the tetrahedron has zero thickness (and therefore opacity) around its outline. The maximum brightness and opacity occur where the tetrahedron is thickest. This thickest point and its attributes must be computed. In

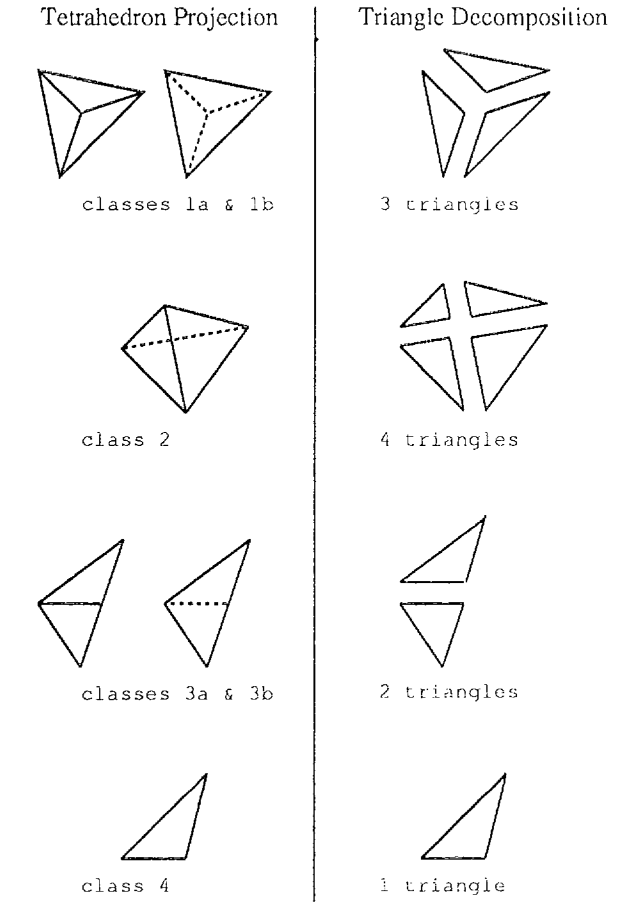


Figure 4: Decomposition of Projected Tetrahedra into Triangles

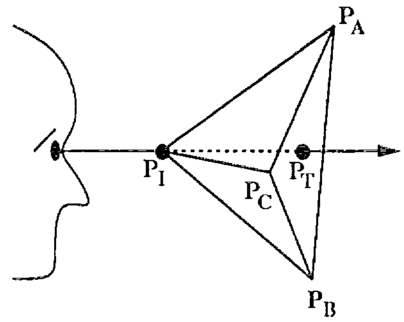


Figure 5: Example of Class 1 Decomposition.

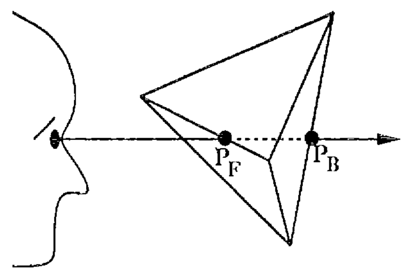


Figure 6: Example of Class 2 Decomposition.

class 4, the point is just the original two vertices collinear with the view point. In classes 2 and 3, a line intersection is performed, and in class 1 a bilinear interpolation of the near or far point on the opposite face is used. In each case, the intersection point is mapped by the inverse viewing transformation, $V^{-1}$, to find the resulting decomposed triangle vertices. The thickness of the tetrahedron is determined at this intersection point by the Euclidean distance formula. The opacity at this vertex is obtained from the thickness of the tetrahedron and the scalar values at the vertex and the intersection point.

For example, a class 1 projection has one interior point, $P_I$ and three boundary points, $P_A$, $P_B$, and $P_C$ (all in the perspective coordinate system). As shown in Figure 5, the vector from the eye through $P_I$ pierces the plane $P_A P_B P_C$ at point $P_T$. The $x$ and $y$ coordinates of $P_T$ are the same as those of $P_I$. A bilinear interpolation is used to find the $z$ coordinate. We define

$$P_T = P_A + u(P_B - P_A) + v(P_C - P_A)$$

and solve this vector equation in $x$ and $y$ for the parameters $u$, $v$. From $u$ and $v$ we can solve for both the $z$-coordinate of $P_T$ and the interpolated value of the the scalar function. $P_T$ is then mapped via $V^{-1}$ to world coordinates and its distance from $V^{-1}P_I$ (the untransformed $P_I$) is computed. Figure 6 shows a ray passing through a class 2 tetrahedron. In this case we need to find $P_F$, the point on the front-facing edge, and $P_B$, the point on the back facing edge.

## 2.5 Ray Integration

We next describe the rules for ray integration at the thickest point of a tetrahedron. We assume that linear interpolation of the brightness and opacity across each triangle decomposed from a tetrahedron is acceptable, so a ray integration is needed only at the thickest point. This approximation is reasonable for tetrahedra with small opacity.

To develop the rules for direct volume rendering, we assume a density volume scatterplot model that uses particles of cross sectional area $A_p$. We also assume that for each scalar value $\rho$ that $f$ might take, there is a corresponding particle number density $N_p(\rho)$, and a corresponding particle color $C_p(\rho)$. The particle color $C_p(\rho)$ is assumed to be the same for all viewing angles. Typically, $N_p$ and $C_p$ are stored as tables.

To determine how the small particles change the color seen along a viewing ray, consider the color of the ray as a function $C(t)$ of the distance $t$ along the ray, where $t$ increases as we advance along the ray toward the viewer. To generate a differential equation for how the particles interact with a ray, consider the change in color that occurs as we advance a small distance $\Delta t$ toward the viewer:

$$C(t+\Delta t) = \underbrace{(1 - N_p(t)A_p\Delta t)C(t)}_{old} + \underbrace{N_p(t)A_p\Delta t C_p(t)}_{new} \quad (1)$$

The subexpression marked as *old* is the color at $t$ attenuated by the opaque particles between $t$ and $t+\Delta t$. The amount of attenuation $N_s(t)A_s\Delta t$ is simply the fraction of area that is covered by particles as opposed to background. The subexpression 'new' is the color contributed by the particles between $t$ and $t+\Delta t$. Taking $\Delta t$ to be differential yields:

$$\frac{dC(t)}{dt} + N_p(t)A_p\left[C(t) - C_p(t)\right] = 0$$

This differential equation cannot be solved in closed form for arbitrary $N_p$ and $C_p$. However, if we assume constant particle color $C_0$, a known value for $C(t_0)$, and that $N_p$ varies linearly between known values $N_0$ and $N_1$ at $t_0$ and $t_1$, we can solve for $C(t_1)$:

$$C(t_1) = C(t_0)e^{-A_p(t_1-t_0)\frac{N_0+N_1}{2}} + C_0\left[1 - e^{-A_p(t_1-t_0)\frac{N_0+N_1}{2}}\right]$$

We can view this equation as stating that the region along the ray between $t_0$ and $t_1$ has a color $C = C_0$ and an opacity $\alpha$ defined by:

$$\alpha = 1 - e^{-A_p(t_1-t_0)\frac{N_0+N_1}{2}}$$

If even less accuracy is acceptable, then Equation 1 can be used directly for alpha:

$$\alpha = A_p(t_1 - t_0)\frac{N_0 + N_1}{2}$$

Approximating $C_0$ by the average particle color between $t_0$ and $t_1$, color can be calculated as:

$$C(t_1) = \alpha\frac{C_p(t_0) + C_p(t_1)}{2} + (1-\alpha)C(t_0)$$

Note that this is just alpha compositing as described by Porter and Duff[PD84] (it is the *atop* operation in their terminology). The preceding discussion shows the motivation for the use of alpha compositing in direct volume rendering.

### 2.6 Rendering

Once the triangles are generated, with each vertex having an associated color $C$ and opacity $\alpha$, they can be rendered back to front in painter's algorithm order with $C$ and $\alpha$ being linearly interpolated in between the vertices (similar to Gouraud shading). The opacity at the zero thickness vertices will be zero, but the color will be determined from the color function $S(\rho)$ discussed in Section 2.5. Each pixel value in the frame buffer will change according to the rule:

$$C_{new} = \alpha C + (1-\alpha)C_{old}$$

where $C_{new}$ is the new pixel color, $C$ is the interpolated color of the polygon at that pixel, $\alpha$ is the interpolated opacity, and where $C_{old}$ is the current pixel value, originally just the background color.

Since this process will generate a large number of adjacent partially transparent polygons, the graphics engine should be of a type that will not duplicate edges for adjacent polygons, or visual artifacts may occur at every shared edge.

Another possible problem can arise when the frame buffer uses one byte to store $\alpha$. If $\alpha$ is reasonable small, precision errors could greatly damage image quality.

## 3 Results

The Projected Tetrahedra algorithm has been implemented in C and runs on several different workstations. With an initial version of the program running on a Sun 4/490 workstation we process about 3900 tetrahedra per second into triangles. The number of triangles created varies with the view point, but is close to 13,000 triangles per second. For our timings we include the time to input the tetrahedra, since they are likely to be produced in the proper back-to-front order by another program. We do not include the time to output the triangles since we will generally pass them directly to a rendering library.

A medium sized volume from a simulation of a binary star formation is defined on a rectilinear grid of 33 × 33 × 15 nodes, or 32 × 32 × 14 cells, giving 14,336 voxels or 71,680 tetrahedra. We used this volume on a Sun 4/490 (Sparc) to create our color image. Color Plate 1 (shown in grey level in Figure 7) compares the Projected Tetrahedra algorithm to more exact volume rendering. Both images in the color figure (and grey version) were rendered at 256 × 192 pixel resolution. The upper image in Figure 7 was generated with the PT method in about 19 seconds plus rendering time, including input and all steps described in Sections 2.1 through 2.5. The timing is independent of image size. The lower image in Figure 7 was rendered on the same computer in about 7 minutes with volumetric ray tracing program using techniques similar to those in [UK88]. The ray traced image time is directly proportional to the number of pixels in the resulting image. The MPDO algorithm[WS90] was used to generate the back to front ordering of tetrahedra for the PT version. This step took approximately 3 seconds including output of the tetrahedra.

Also note the background white lines equally spaced both horizontally and vertically in Figure 7. The image buffer was initialized to this pattern of lines on a black background. The lines accentuate the transparent components of the rendered image. This is a computationally free way to highlight the areas of low opacity but high brightness and to distinguish them from areas of high opacity but low brightness. The initial pattern can be defined procedurally or by loading a pre-computed image.

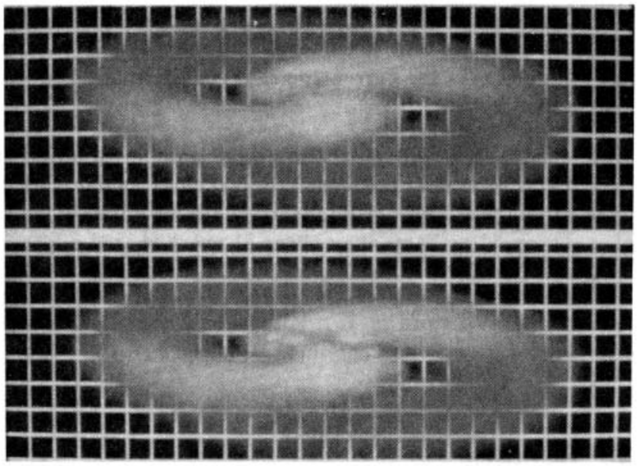

Figure 7: Binary Star Formation Simulation. Top: ray traced image. Bottom: Projected Tetrahedra image

*Editor's note: This image is reproduced in color on front cover.*

# 4 Future Work

It is sometimes useful to embed opaque geometric primitives in the volume. We have found it helpful for visual interpretation to place *grid bars* within a ray traced cloud-like volume. Grid bars are long thin opaque cylinders or rectangular parallelepipeds embedded in a volume. A few bars are usually defined evenly spaced along each axis of the cartesian coordinate system, forming a bounding box of the original grid. The images would provide more visual cues than those obtained using the background grid described in Section 3.

There are several examples of more important applications for embedded opaque geometry primitives. The geometric model of a wing or other structure may be embedded in a volume computed by a computational fluid dynamics simulation to aid visualization of the flow. Flow ribbons tracking a vector-valued function in the same region as the scalar may be shown by calculating the ribbon paths and rendering these polygons in the volume[SN89].

One possible way to combine the Projected Tetrahedra algorithm with opaque surface primitives would be first to render the opaque surfaces with a Z-buffer[FvDFH90], and then to render each transparent polygon in depth order, omitting any contribution to a pixel if the z-value of the triangle at that pixel is deeper than the z-value stored in the Z-buffer. This would introduce additional error only for tetrahedra that contain surfaces.

Embedded transparent iso-surfaces are sometime useful. Once the cells have been divided into tetrahedra, it would be possible to extract isovalued surfaces in a manner similar to the marching cubes algorithm[LC87]. Such a *marching tetrahedra* algorithm would generate three and four-sided polygons that could be rendered separately or embedded in the volume. The surfaces can be rendered with any degree of transparency.

The greatest promise of the Projected Tetrahedra technique is its potential for faster volume rendering useful in previewers and interactive systems. As the rendering can be done in hardware, the bottlenecks will be either the conversion of volume elements to polygonal approximations, described in this paper, or the generation of the tetrahedra in a back-to-front order. Our current implementation certainly cannot generate triangles fast enough to keep up with a 100,000 polygon per second graphics workstation, but is reasonably fast and is linear in the number of input tetrahedra.

# 5 Conclusion

The Projected Tetrahedra algorithm presented in this paper approximates the volume rendering of tetrahedral cells by hardware renderable partially transparent triangles. This approximation is accomplished by finding triangles with the same silhouette as a tetrahedron, and linearly interpolating color and opacity on the triangles from the fully transparent silhouette edges to the values at the thickest part of the tetrahedron, as seen by the viewer.

The strengths of the method are that the triangles can be rendered in hardware, that perspective or orthographic viewing can be used, and that unstructured mesh geometry can be used provided that a painter's depth ordering is known (or an A-buffer with sufficient levels of transparency is available). The voxels and thus the tetrahedra can be processed independently, so the Projected Tetrahedra algorithm may be implemented in parallel. This algorithm also has very small memory requirements since the only data needed are for the tetrahedron currently being processed.

The weaknesses of the Projected Tetrahedra method include the restriction to tetrahedral cells and possible precision problems in the scan conversion. In Section 4 we mentioned that conventional geometric primitives could be embedded in the volume with an appropriate depth sort. Even with such a sort there would be visibility errors wherever the geometric primitive intersected a volume element. This would at best limit the number and complexity of embedded primitives. The visibility errors would be more pronounced than the other approximation artifacts introduced by the PT algorithm. The primary approximation used in the Projected Tetrahedra method is the low particle number density assumption of the ray integration section (which implies the linear interpolation used on the triangles). This approximation can cause problems in datasets where the particle density is high. An example of such a high particle density dataset is the medical dataset used by Levoy[Lev88], where pseudo-surfaces are generated. The PT method is also suspect for very large datasets, because one of the primary reasons for generating the triangles is to avoid having to perform ray integration at every pixel covered by a tetrahedron. For very large datasets most tetrahedra would cover at most a few pixels, so the triangles would not yield much time savings.

The limitations above seem to indicate that the Projected Tetrahedra method presented in this paper is primarily useful for medium size datasets (on the order of millions of cells or fewer) that have reasonably smooth variations (e.g. no surface-bone interfaces). Such data include many fluid flow and stress analysis calculations. With hardware triangle rendering this algorithm for direct volume rendering should also find a place in interactive volume visualization environments.

## 6 Acknowledgements

We are very grateful to Peter Williams for developing an algorithm for producing a back-to-front traversal of an arbitrary three-dimensional volume of data. Thanks also to Dennis Gannon and Henry Neeman for their help and encouragement. We would like to thank Richard Durison, Department of Astronomy, Indiana University, Bloomington for the star simulation data. We would also like to thank the anonymous reviewer for the careful reading of this paper and the suggestions for improvement. This work was partially supported by the Air Force Office of Scientific Research Grant AFOSR-90-0044.

## Appendix: Viewing Matrices

If $u$, $v$, and $w$ are the normalized orthogonal basis vectors for the viewing coordinate system in $(x, y, z)$ world coordinates, then the rotation matrix is straightforward to compute. The perspective transformation is not a projection to a two-dimensional plane, but to 3-dimensions. Thus the inverse transformation can be applied to the intersection points we compute during the triangle decomposition. The projection matrix $P$ (below) transforms the viewing frustrum aligned with the $z$-axis to a rectangular parallelepiped. We later discard the $z$ component of the transformed coordinate to project to the view plane. The three matrices are determined from initial view data and are multiplied to produce a viewing transformation matrix as follows:

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -x_{eye} & -y_{eye} & -z_{eye} & 1 \end{bmatrix}$$

$$R = \begin{bmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & (n+f)t & t \\ 0 & 0 & -fnt & 0 \end{bmatrix}$$

where $n$ is the distance from the eye to the near clipping plane, $f$ is the distance to the far clipping plane, $\theta$ is the field of view, and $t = tan(\theta)/2$. The viewing transformation is the product of these matrices

$$V = TRP$$

The inverse of this matrix is needed and is easily determined also: the inverse of the orthogonal matrix $R$ is its transpose and $P$ is composed of a $2 \times 2$ identity and a $2 \times 2$ block.

$$T^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x_{eye} & y_{eye} & z_{eye} & 1 \end{bmatrix}$$

$$R^{-1} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$P^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1/fnt \\ 0 & 0 & 1/t & (n+f)t/fnt^2 \end{bmatrix}$$

The inverse viewing transformation is the product of these matrices

$$V^{-1} = P^{-1}R^{-1}T^{-1}$$

## References

[Bli82] James F. Blinn. Light reflection functions for simulation of clouds and dusty surfaces. *Computer Graphics*, 16(3):21–30, July 1982. ACM Siggraph '82 Conference Proceedings.

[Car84] Loren Carpenter. The A-buffer, an antialiased hidden surface method. *Computer Graphics*, 18(3):103–108, July 1984. ACM Siggraph '84 Conference Proceedings.

[DCH88] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. *Computer Graphics*, 22(4):65–74, July 1988. ACM Siggraph '88 Conference Proceedings.

[FGR85] Gideon Frieder, Dan Gordon, and Anthony Reynolds. Back-to-front display of voxel-based objects. *IEEE Computer Graphics and Applications*, 5(1):52–60, January 1985.

[FvDFH90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, MA, second edition, 1990.

[HS89] William Hibbard and David Santek. Interactivity is the key. In *Proceedings of the Chapel Hill Workshop on Volume Visualization*, pages 39–43, May 1989.

[Kau87] Arie Kaufman. Efficient algorithms for 3d scan-conversion of parametric curves, surfaces, and volumes. *Computer Graphics*, 21(4):171–179, July 1987. ACM Siggraph '87 Conference Proceedings.

[KH84] James T. Kajiya and B. P. Von Herzen. Ray tracing volume densities. *Computer Graphics*, 18(4):165–174, July 1984. ACM Siggraph '84 Conference Proceedings.

[LC87] 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.

[Lev88] Mark Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988.

[LGLD86] Reiner Lenz, Bjorn Gudmundsson, Bjorn Lindskog, and Per Danielsson. Display of density volumes. *IEEE Computer Graphics and Applications*, 6(7), July 1986.

[MHC90] Nelson Max, Pat Hanrahan, and Roger Crawfis. Area and volume coherence for efficient visualization of 3d scalar functions. *Computer Graphics*, 24(5), December 1990. San Diego Volume Visualization Conference Proceedings.

[PD84] Thomas Porter and Tom Duff. Compositing digital images. *Computer Graphics*, 18(4):253–260, July 1984. ACM Siggraph '84 Conference Proceedings.

[Sab88] Paolo Sabella. A rendering algorithm for visualizing 3d scalar fields. *Computer Graphics*, 22(4):51–58, July 1988. ACM Siggraph '88 Conference Proceedings.

[SN89] Peter Shirley and Henry Neeman. Volume visualization at the Center for Supercomputing Research and Development. In *Proceedings of the Chapel Hill Workshop on Volume Visualization*, pages 17–20, May 1989.

[UK88] Craig Upson and Micheal Keeler. V-buffer: Visible volume rendering. *Computer Graphics*, 22(4):59–64, July 1988. ACM Siggraph '88 Conference Proceedings.

[Wes90] Lee Westover. Footprint evaluation for volume rendering. *Computer Graphics*, 24(4):367–376, August 1990. ACM Siggraph '90 Conference Proceedings.

[WS90] Peter L. Williams and Peter Shirley. An a priori depth ordering algorithm for meshed polyhedra. Technical Report 1018, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, September 1990.